Abstract

Optimizing Memory Management for Disaggregated Architectures

Yupeng Tang

2024

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

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

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

Optimizing Memory Management for Disaggregated Architectures

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

by
Yupeng Tang

Dissertation Director: Anurag Khandelwal

Dec, 2024

Contents

List of Figures

List of Tables

Acknowledgements

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

Chapter 1

Introduction

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

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

Memory disaggregation is the first step toward realizing the full potential of resource disaggregation, enabling data centers to efficiently allocate and utilize resources based on dynamic application needs. This ultimately leads to improved performance and better resource utilization.

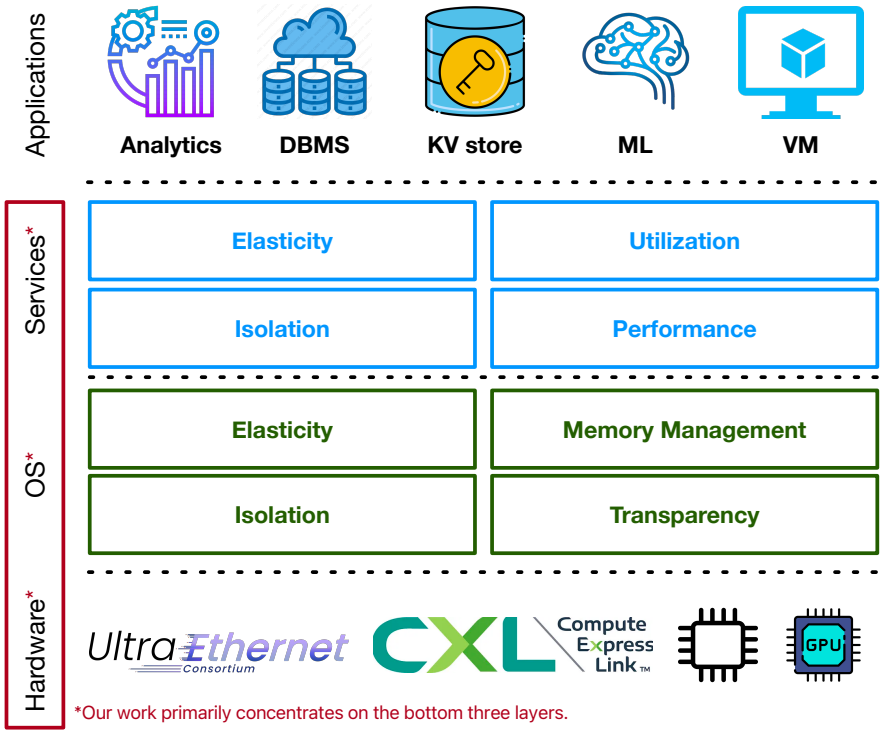


Fig. 1.1: **Cloud Stack of Disaggregated Architecture.**

1.1 Limitations of Existing Approaches

While resource disaggregation offers numerous advantages, transitioning existing applications to a disaggregated architecture is far from straightforward. Recent research has explored various approaches to address this challenge. Some efforts focus on adapting applications to optimize their use of disaggregated memory [27–30], while others aim to transparently port applications, shifting the responsibility of mitigating performance penalties—arising from the mismatch between disaggregated architectures and traditional software interfaces—to the service or operating system layer [1, 2, 31–34].

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

Application interface. In disaggregated architectures, applications 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 nodes [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 [35] and addressing applications with irregular memory access patterns [36]. 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) [10, 11, 37] offer promising enhancements with faster access times and hardware-supported cache coherence. Yet, the current state of hardware prototypes and software support for these technologies remains limited.

1.2 Thesis Overview

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

1.2.1 Memory management as a Service

With least modification to lower layers such as OS/Hardware, we explore the design requirement and challenges in provoding memory management as a service. We proposed an end-to-end system design called Jiffy, which enables multiple application/tasks multiplex memory in a elastic manner. Jiffy also provides multiple popular data structure interface and can be easily applied to existing cloud applications.

1.2.2 In-network memory management OS-design

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

1.2.3 Memory management adaptation for new-generation interconnects

In prior work [1, 2], ethernet is considered as the most popular interconnect for disaggregated data centers. However, as new memory interconnects are emerging, such as Compute Express Link(CXL), new adaptation of memory management needs to be made regarding the new interconnect interface. Within the context of disaggregated architecture, new problems arises such as how can the applications leverage multiple tiers of memory. Therefore, we start with a perfomance analysis on CXL 1.1 single host extended memory, and then we propose a new system design that integrates disaggregated CXL memory pool with today's emerging popular application - LLM inference.

1.3 Outline and Previously Published Work

This dissertation is organized as follows. Chapter 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 [35][1]. Chapter 3 revises material from [1][2] and [36][3]. Finally, Chapter 4 revises material from [38][4].

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

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

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

4. Work done in collaboration with the Bytedance Infrastructure team

Chapter 2

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 appli-

cations experiencing fluctuating memory demands [35]. 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 [39] 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 [35]. 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

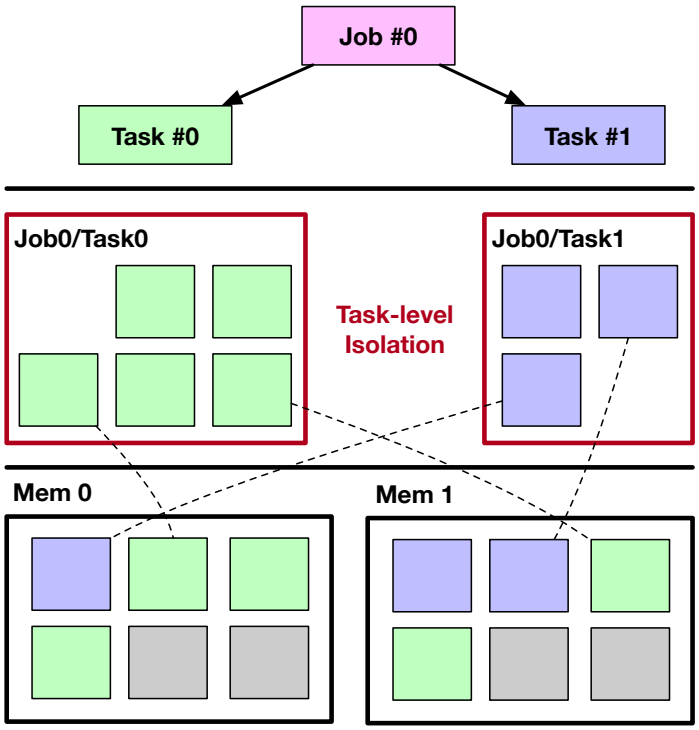


Fig. 2.1: **Jiffy Overview**.Jiffy allocates memory resources individually for each task within a job. Memory is allocated in small, fixed-sized blocks to ensure elastic scaling of memory according to demand.

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

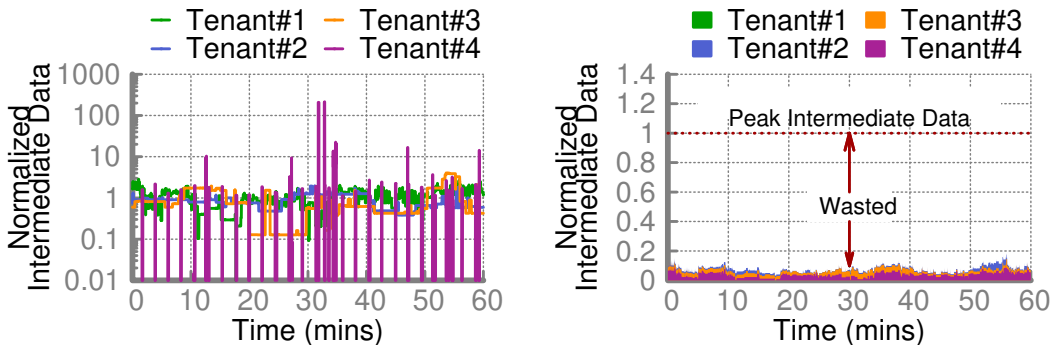
Data analytics applications, which utilize disaggregated memory for inter-task communication and intermediate data storage, are becoming increasingly common. As discussed in [39–42], these applications handle user requests in the form of jobs, each defining its memory needs upon creation. The dilemma of balancing performance with resource efficiency for job-level memory allocation has been extensively studied [43, 44]. 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 [43], 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 [35], an elastic memory service tailored for disaggregated architectures. As shown in Figure 2.1, 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.

Serverless architectures offer on-demand elasticity of compute and persistent storage, while charging users for resources consumed by their jobs at fine-grained timescales [45–47]. While originally deemed useful only for web microservices, IoT and ETL workloads [48, 49], recent work on serverless analytics has demonstrated the benefits of serverless architectures for resource- and cost-efficient data analytics [39, 40, 42, 50–62].

The core idea in serverless analytics is to use a remote low-latency, high-throughput *ephemeral storage system* for: (1) inter-task[1] communication via shared memory; and (2) for multi-stage jobs, storing intermediate data beyond the lifetime of the task that produced the data (until it is consumed

1. Existing distributed programming frameworks, while different in underlying programming models and semantics, share a common structure (Figure **??**, Figure 2.3) — the job is split into multiple *tasks*, possibly organized along multiple stages or a directed acyclic graph. Each task generates *intermediate* data during its execution; upon completion, each task partitions its intermediate data and exchanges it with tasks in the next stage.



(a) Intermediate data (normalized by mean usage)

(b) Cumulative intermediate data (normalized by peak usage)

Fig. 2.2: Analysis of production workloads from Snowflake [61] for four tenants over a 1 hour window: (a) the ratio of peak to average storage usage for a job can vary by an order of magnitude during its execution; and (b) provisioning for peak usage results in average utilization $< 10\%$. Across all tenants, the average utilization is 19%.

by downstream tasks). Ephemeral storage systems thus allow decoupling storage, communication and lifetime management of intermediate data from individual compute tasks, enabling serverless analytics frameworks to exploit the on-demand compute elasticity offered by serverless architectures.

Existing ephemeral storage systems [39, 50], however, suffer from a fundamental limitation: they allocate storage resources at the job granularity. That is, jobs specify their storage demands at the time of the submission; and, the system allocates and reserves storage resources equal to the job's demand (potentially by elastic scaling of total ephemeral storage system capacity) for the entirety of its lifetime [39, Figure 1].

The problem of performance degradation and/or resource underutilization for such job-level resource allocation in storage systems is well-understood [44, 61]. On the one hand, if jobs specify their average demand, their performance degrades when instantaneous demand is higher than their average demand (due to read/write requests being executed on slower secondary storage, *e.g.*, S3), as shown in Figure 2.2(a). On the other hand, if jobs specify their peak demand, the system suffers from resource underutilization when their instantaneous demand is lower than the peak demand, as shown in Figure 2.2(b). Indeed, the problem worsens as the difference between the peak demand and the average demand increases. Unfortunately, the target use case of ephemeral storage systems — intermediate data — is a bad-case scenario for the difference between peak and average demands: recent deployment studies have reported that intermediate data sizes can vary over multiple orders

of magnitude during the lifetime of the job [61]. For instance, Figure 2.2 presents our analysis of the publicly-released dataset of > 2000 tenants from Snowflake [61]: it shows that the ratio of peak to average demands in Snowflake production workloads can vary by two orders of magnitude over a period of minutes! As a result, job-level resource allocation in existing ephemeral storage systems can lead to significant performance degradation and/or resource underutilization (our evaluation results in §3.14 show as much as $4.1\times$ performance degradation and 60% resource underutilization for production workloads).

We present Jiffy, an ephemeral storage system for serverless analytics that allocates storage resources at the granularity of small fixed-size storage blocks — multiple storage 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 processes at the granularity of fixed-size memory blocks (pages); indeed, Jiffy adapts this design to ephemeral storage systems for serverless analytics. Performing resource allocation at the granularity of small storage blocks allows Jiffy to elastically scale storage 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 ephemeral storage capacity across concurrently running jobs, thus minimizing the overheads of reads and writes to slower secondary storage (*e.g.*, S3). We show that such fine-grained resource allocation allows Jiffy to achieve $1.6-2.5\times$ improvement in application-level performance compared to Pocket [39], across a variety of workloads and cluster configurations.

Enabling fine-grained resource allocation requires resolving four unique challenges introduced by serverless analytics:

- First, each serverless analytics job can be organized around multiple stages (or a directed acyclic graph), with tens to thousands of individual tasks in each stage [39, 40, 42, 50–61]. Thus, performing fine-grained resource allocation requires an efficient mechanism to keep an up-to-date mapping between tasks and storage blocks allocated to individual tasks.
- Second, the number of tasks reading and writing to the shared ephemeral storage can change rapidly in serverless analytics. Thus, task-level isolation becomes critical: arrival and departure of new tasks should not impact the performance of existing tasks.

- Third, decoupling of serverless tasks from their intermediate data means that the tasks can fail independent of the intermediate data. Thus, we need mechanisms for explicit lifetime management of intermediate data.
- Fourth, decoupling of tasks from their intermediate data also means that data partitioning upon elastic scaling of storage capacity becomes challenging, especially for certain data types used in serveless analytics (*e.g.*, key-value stores [39,40,40,42,50,54,56,60]). Indeed, naïvely delegating this to applications would require large network transfers (between compute tasks and ephemeral storage system) and data read/write operations every time the capacity is scaled (§**??**). Thus, we need new mechanisms to efficiently enable data partitioning within the ephemeral storage system.

Jiffy resolves these challenges by integrating several mechanisms into an end-to-end system. First, in a sharp contrast to classical distributed shared memory systems [63–67] and recent in-memory stores [24, 25, 27, 68] that use a global address space, Jiffy exposes a hierarchical address space that captures the structure of the analytics job (*e.g.*, directed-acyclic graphs with individual tasks) [44, 61]. Such a hierarchical addressing mechanism allows Jiffy to both efficiently manage the mapping between storage blocks and tasks, and provide task-level isolation. Second, Jiffy ties the hierarchical addresses with a lease-based mechanism for efficient lifetime management of intermediate data. Finally, similar to function shipping, Jiffy supports partition-function shipping — analytics jobs can offload data repartitioning upon resource allocation/deallocation to Jiffy, that performs seamless data repartitioning. We discuss in §**??** how, for each of these techniques, the aforementioned unique challenges introduced by serverless architectures require Jiffy to make different design decisions than original realizations of these mechanisms. Jiffy integrates these mechanisms into an end-to-end ephemeral storage system that provides resource elasticity at the granularity of seconds, matching the compute elasticity timescales of serverless architectures.

We have realized an end-to-end implementation of Jiffy, which we will open-source along with the paper. Jiffy's data plane enables compute tasks to read/write intermediate data to their blocks via an intuitive, programmable, API (§**??**). We demonstrate the expressiveness of Jiffy's API by realizing serverless incarnations of several powerful distributed programming frameworks atop Jiffy (§**??**): MapReduce [69], Dryad [70], StreamScope [71] and Piccolo [72]. We compare Jiffy against five state-of-the-art ephemeral storage systems over a variety of workloads. Our evaluation suggests

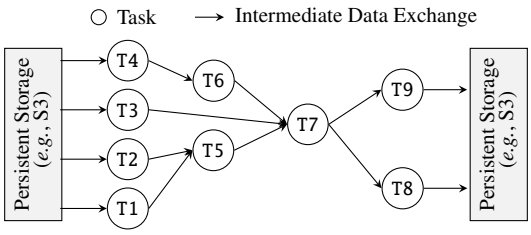


Fig. 2.3: **Execution DAG example for a typical analytics job.** Intermediate data exchange across tasks occurs via Jiffy.

that fine-grained resource allocation in Jiffy allows it to improve resource utilization by as much as $3\times$ and distributed analytics performance by a factor of $1.6 - 2.5\times$ compared to peak provisioning.

2.2 Jiffy Design

Jiffy enables fine-grained sharing of ephemeral storage capacity across concurrently running serverless analytics jobs. Inspired by virtual memory, Jiffy partitions the storage capacity into fixed-sized blocks (akin to virtual memory pages), and performs storage allocations at the granularity of these blocks. This allows Jiffy to achieve two desirable properties. First, multiplexing the available capacity at block granularity allows Jiffy to match instantaneous job demands at seconds timescales. Second, Jiffy does not require jobs to know (even an estimate of) intermediate data sizes a priori; as tasks write/delete data, Jiffy dynamically allocates/deallocates resources at block granularity.

Remark. Multiplexing available storage capacity is different from scaling the capacity of the storage pool. Prior systems, including Pocket, focus on the latter: since resource allocation is done at job granularity, as jobs arrive or finish, these systems add and remove the storage servers to elastically scale the capacity of the ephemeral storage system. However, existing capacity may be underutilized since a job may not be using the storage allocated to it. Jiffy focuses on the former: efficiently sharing the capacity available at any given instant of time across concurrently running jobs. When the ephemeral storage capacity utilization is high (i.e., many jobs are actually using the storage capacity), Jiffy can add storage servers to scale up the capacity similar to Pocket. Interestingly, by efficiently multiplexing the available storage capacity across concurrently running jobs, Jiffy also reduces the frequency at which storage servers need to be added/removed from the pool.

As discussed in §4.2, enabling fine-grained resource allocation requires resolving four unique

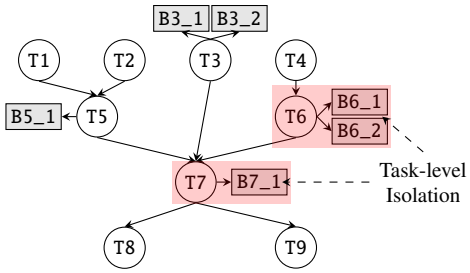


Fig. 2.4: **Hierarchical addressing** for the job in Figure 2.3. Jiffy provides task-level resource isolation for ephemeral storage under each task address-prefix (§2.2.1). Note that block addresses are only assigned to address-prefixes with currently allocated blocks; for tasks `T1`, `T2` and `T4`, blocks are directly read from persistent storage and not stored in Jiffy.

challenges introduced by serverless analytics (primarily due to decoupling of compute tasks from their intermediate data). In this section, we describe how Jiffy employs hierarchical addressing (§2.2.1), intermediate data lifetime management (§2.2.2) and flexible data repartitioning (§2.2.3) to resolve these challenges. To assist our discussion, we will use the example in Figure 2.3, which shows the execution plan for a representative analytics job. The plan is organized as a directed acyclic graph (DAG) where nodes correspond to computation tasks (implemented as serverless functions[2]), while edges denote intermediate data exchange between them via Jiffy.

2.2.1 Hierarchical Addressing

Analytics job are usually organized around multiple stages or a directed acyclic graph. In serverless analytics, where compute elasticity is a first-class primitive, each job may execute tens to thousands of individual tasks [39,40,42,50–61]. Thus, performing fine-grained resource allocation requires an efficient mechanism to keep an up-to-date mapping between tasks and storage blocks allocated to individual tasks. Moreover, the number of tasks reading and writing to the shared ephemeral storage can change rapidly. Under such high concurrency and churn, it becomes important to provide isolation at the granularity of individual tasks: arrival and departure of a task should not affect the resources allocated to other tasks, even from the same job (since it can degrade the *overall* job performance). In this subsection, we describe Jiffy's hierarchical addressing — a simple, effective, mechanism that enables Jiffy to maintain a mapping between individual tasks and storage blocks allocated to these tasks, as well as provide isolation at individual task granularity.

2. Functions refer to a basic computation unit in serverless architectures, *e.g.*, Amazon Lambdas [45], Google Functions [47], Azure Functions [46], etc.

Motivated by the Internet hierarchical IP addressing mechanism that captures *network structure*, Jiffy employs a software-based hierarchical addressing mechanism that captures *execution structure* in analytics jobs. Specifically, Jiffy organizes intermediate data for analytics jobs within a "virtual" address hierarchy to capture the dependencies between intermediate data for different tasks. We provide an example below, but conceptually, internal nodes in the hierarchy correspond to tasks in the DAG, while leaf nodes correspond to Jiffy blocks storing intermediate data generated by the tasks. Blocks form the final layer of the hierarchy: block addresses are defined by the path used to reach it in the hierarchy. The immediate address prefix of a block, therefore, identifies the task that generated it. Finally, the edges between internal nodes capture the dependencies between the intermediate data generated by them. To construct the address hierarchy, Jiffy uses the execution plan for a job (*e.g.*, using AWS Step Function and Azure Durable Function, or via explicit workflow specification from the job). Otherwise, Jiffy initializes the hierarchy to a single node, and *deduces* the rest on-the-fly based on the intermediate data dependencies between the job's tasks (during registration of individual tasks using Jiffy API §**??**); this allows Honeycomb to support dynamic query plans, where the DAG is not known a priori.

Example. Figure 2.4 shows the address hierarchy for the job from Figure 2.3. The internal nodes `T1`-`T9` correspond to tasks in the DAG, while leaf nodes `B3_1`, `B3_2`, etc., correspond to the data blocks allocated to them by Jiffy for storing their intermediate data. Edges (`T1`, `T5`) and (`T2`, `T5`) in the address hierarchy indicate that the intermediate data in `T5` depends on the intermediate data from both `T1` and `T2`. The complete address of block `B6_2` under `T6` would be `T4.T6.B6_2`, while the address-prefix `T4.T6` identifies all blocks allocated to `T6`. To construct the address hierarchy, Jiffy either uses the execution plan from Figure 2.3 if the job provides it, or deduces it on the fly. For instance, Jiffy can deduce that since all sub-tasks in `T7` access data produced by `T3`, `T5`, and `T6`, `T7` must have them as parents in the hierarchy.

Organizing intermediate data across an address hierarchy allows Jiffy to manage resource allocations for an address-prefix independent of other prefixes. Specifically, if the storage in a specific address-prefix spills over to persistent storage (using mechanisms from Pocket), it does not affect the performance of other address-prefixes. Moreover, Jiffy ensures that once a block is allocated to an address-prefix, it will not be reclaimed until the application either explicitly reclaims it, or stops

renewing leases for it (§2.2.2), affording *isolation* at address-prefix granularity. Since address-prefixes correspond to tasks in Jiffy address hierarchy, this enables task-level isolation regardless of task concurrency and churn. This is similar to virtual memory, where each process is is assigned its own virtual address space that enables isolation at process granularity; Jiffy does this at individual task granularity using hierarchical addressing that captures the execution structure of the job.

We outline two important design issues. First, Jiffy's fine-grained resource allocation should be decoupled from the policies required to enforce desired system behavior. For instance, algorithms to achieve fairness in resource allocation across various jobs or tenants can be easily integrated on top of Jiffy allocation mechanism. This is orthogonal to Jiffy's goals of enabling fine-grained sharing (where the overall goal is high resource utilization). Second, address translation — the mapping from virtual addresses to physical storage blocks — is performed similar to Pocket [39] at the centralized metadata server. Thus, unlike hardware address translation that imposes a limit on the size (depth and breadth) of the execution DAG, Jiffy can easily perform addressing for arbitrary DAGs. Jiffy's hierarchical addressing and fine-grained resource allocation does introduce additional complexity at the controller; we evaluate the scalability of our controller implementation in the evaluation section and demonstrate that Jiffy can still scale to $\sim$45K requests per second per core, which is large enough for most realistic deployments.

Block sizing. Similar to page-size in traditional virtual memory, block-size in Jiffy exposes well-known tradeoffs between the amount of metadata that needs to be stored at the control plane, and memory utilization. In particular, larger block-sizes reduce the amount of per-block metadata at the control plane, at the cost of potentially reduced memory utilization from data fragmentation within blocks, and vice versa. We note that Jiffy does not suffer the traditional overheads of higher I/O with larger blocks, since it supports fine-grained access within blocks via its data structure interface (§**??**). Moreover, Jiffy also controls under-utilization within a block via data repartitioning, as described in §2.2.3. While Jiffy block sizes can be configured, it employs the block sizes typically used for files in analytics frameworks (*e.g.*, 128MB in HDFS [73]) for compatibility.

Isolation granularity. Since the nodes in the address hierarchy correspond to tasks, Jiffy provides task-level isolation. It is, however, possible to provide finer or coarser-grained isolation by simply adding another layer to the hierarchy (*e.g.*, for isolation at the granularity of tables in data

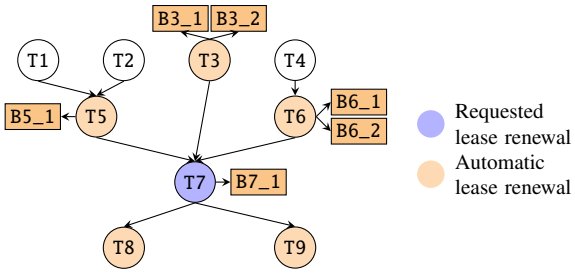


Fig. 2.5: **Lease Renewal via Address Hierarchy.** Hierarchical addressing simplifies lease renewal in Jiffy (§2.2.2), since lease renewal for an address-prefix automatically implies renewals for all parent and descendent address-prefixes in the hierarchy.

lake queries) or removing a layer from the hierarchy (*e.g.*, for stage-level isolation in MapReduce frameworks). We chose task-level isolation as our default since most analytics frameworks stand to benefit from task-level isolation, but do not require finer-grained isolation. Individual applications can, however, chose to create custom hierarchies using Jiffy API, as we outline in §**??**.

2.2.2 Data Lifetime Management

Existing ephemeral storage systems that perform job-level resource allocation also manage data lifetimes at job granularity — reclaiming storage when the job explicitly deregisters. A unique feature of serverless analytics is that a task's intermediate data storage is decoupled from its execution: while its execution occurs at the serverless compute platform, the data generated and consumed by it would reside at the ephemeral storage system. This also results in the decoupling of their fault domains: with standard mechanisms (*e.g.*, reference counting approaches [74–76]), when the task fails, its corresponding intermediate data becomes dangling state at the ephemeral storage system. To avoid the resulting inefficiency, we need additional mechanisms to efficiently perform task-level data lifetime management.

Jiffy achieves this by integrating well-known lease management mechanisms [77–79] with hierarchical addressing to enable lifetime management of intermediate data. In particular, Jiffy associates each address-prefix in a job's hierarchical addressing with a lease, and only keeps its data in memory as long as its lease is renewed. Consequently, a job periodically renews leases for the address-prefixes of tasks that are currently running (serverless environments already monitor function execution for billing purposes; these functions could be used to trigger lease renewals). Jiffy tracks the time a lease was last renewed for each node in the address hierarchy, and updates it for

the relevant nodes when a new renewal request for a particular address-prefix is received. Finally, on lease expiry for a particular address-prefix, Jiffy reclaims all resources allocated to it.

The new aspect of lease management in Jiffy is that it exploits the DAG-based hierarchical addressing to determine dependencies between leases. On receiving a lease renewal request from a task that is currently running, Jiffy renews leases not only for its address-prefix, but also for the prefixes of tasks that they depend on (i.e., parent nodes in the hierarchy), and for all the prefixes of tasks that depend on it (i.e., all descendant nodes in the hierarchy). This ensures that while a task is running, not only is its own intermediate data kept in memory, but also the data for all the tasks that it depends on, and all the tasks that depend on it. Moreover, this also significantly reduces the number of lease renewal messages that a job has to send.

Jiffy's leasing mechanism finds a favorable tradeoff between age-based eviction (*e.g.*, in caching approaches, where jobs have no control on intermediate data lifetime) and explicit acquisition and release (where jobs have full control, but job failures could lead to orphaned state). Jiffy's mechanism not only provides jobs control over the lifetime of their storage resources (via explicit leases), but also ties the fate of the allocated resources to the job — if a lease is not renewed (*e.g.*, due to job or task failure, or if resources are no longer needed), Jiffy reassigns resources to other jobs or tasks upon lease expiry.

Example. In Figure 2.3, during task `T7`'s execution, the job periodically renews leases for the prefix `T4.T6.T7`[3] — Jiffy keeps the intermediate data for the blocks under it in memory as long as the job renews leases for it. Moreover, a lease renewal for task `T7`'s prefix also renews leases for its parent tasks prefixes (i.e., for tasks `T3`, `T5`, `T6`) and for its descendent task prefixes (i.e., for tasks `T8`, `T9`), as shown in Figure 2.5. Thus, renewing task `T7`'s lease ensures that `T7` can still use the intermediate data generated by its parent tasks; moreover, if any of `T7`'s downstream tasks are active, their intermediate data is automatically kept in memory as well.

Lease duration. Lease duration in Jiffy exposes a tradeoff between control plane bandwidth and system utilization *over time*. Specifically, longer lease durations reduce the network traffic to the control plane since jobs renew their leases at coarser granularities, but reduce system utilization since Jiffy does not reclaim (potentially unused) resources from jobs until their leases expire. The

3. Note that task `T7` has four different address-prefixes: the job can renew leases for its data using any of them.

Table 2.1: **Jiffy User-facing API**. See §**??** for details.

	API	Description
	`connect(honeycombAddress)`	Connect to Jiffy.

Address Hierarchy	`createAddrPrefix(addr, parent, optionalArgs)`
	`createHierarchy(dag, optionalArgs)`
	`flushAddrPrefix(addr, externalPath)`
	`loadAddrPrefix(addr, externalPath)`
		Create address-prefix `addr` with given `parent` address `optionalArgs` (*e.g.*, initial capacity), or, create addres from execution plan provided as a DAG `dag`.
		Flush/load data in address-prefix to external persistent

	`leaseDuration = getLeaseDuration(addr)` `renewLease(addr)`	Get the lease duration associated with address-prefix `a` Send lease renewal request for address-prefix `addr`.
Data Structure	`ds = initDataStructure(addr, type)`	Initialize data structure of given `type` in address-prefix get handle `ds` that encapsulates physical locations of al
	Data structure-specific interface implemented using block API (Figure 2.6).	See Table 2.2 in §**??** for examples.
	`listener = ds.subscribe(op)` `notif = listener.get(timeout)`	Subscribe to notifications for operations of type `op` on Get latest notification; waits `timeout` seconds for resp

problem of picking the right lease duration based on these constraints has been well studied in prior work [77, 78], and can be configured in Jiffy to meet deployment-specific goals.

2.2.3 Flexible Data Repartitioning

Decoupling compute tasks from their intermediate data in serverless analytics makes it challenging to efficiently achieve ephemeral storage elasticity at fine granularities. Specifically, as storage is allocated/deallocated to a task, the intermediate data needs to be repartitioned across the remaining blocks. However, decoupling of compute tasks from ephemeral storage, and large number of concurrent tasks means that this repartitioning should not be offloaded to the application. For instance, many existing serverless analytics approaches [39,50] employ key-value stores for intermediate data storage. In such a setting, if the compute task were to repartition the intermediate data on memory scaling, it would have to first read the key-value pairs from the store over the network, compute the data partitions across the new memory allocation, and write back the data to the store. This would incur significant network latency and bandwidth overheads for the task.

As we discuss in §**??**, Jiffy already implements standard data structures used in data analytics frameworks — *e.g.*, files [52,57–59,61], to key-value pairs [39,40,40,42,50,54,56,60] to queues [51, 53]. Analytics jobs using these data structures can offload repartitioning of intermediate data upon resource allocation/deallocation to Jiffy. Each block allocated to a Jiffy data structure tracks the fraction of the block memory capacity that is currently being used to store data. Whenever the usage

grows above a high threshold, Jiffy, in turn, allocates a new block to the corresponding address-prefix[4]. Subsequently, the overloaded block triggers data structure-specific repartitioning to move part of its data to the new block. Similarly, when the block usage drops below a low threshold, Jiffy identifies another block in the address-prefix with low-usage with which the block can merge its data. The block then conducts the required repartitioning, after which Jiffy deallocates it. Note that by having the target block conduct the repartitioning instead of the compute task, Jiffy avoids the network and computational overheads for task itself. Finally, we note that data repartitioning occurs asynchronously in Jiffy: data access operations across data structure blocks can proceed even while repartitioning is in progress. This allows Jiffy to ensure application performance is minimally impacted due to data repartitioning (§**??**).

Data structures included in Jiffy already allow us to implement serverless incarnations of several powerful distributed programming frameworks on top of Jiffy: MapReduce [69, 81], Dryad [70], StreamScope [71] and Piccolo [72]. We note that data structures used in analytics frameworks — files, queues, key-value stores — require very simple repartitioning mechanisms (unlike data structures like B-trees or other ordered trees that are not used in data analytics frameworks). As such, serverless applications employing these programming models can run on Jiffy and leverage its flexible data repartitioning without any modification.

Thresholds for elastic scaling. The high and low thresholds for elastic scaling in Jiffy expose a tradeoff between the data plane network bandwidth and task performance on one hand, and system utilization on the other. Specifically, if the high and low thresholds are set high and low enough, respectively, then elastic scaling is triggered rarely, reducing the amount of network traffic due to data repartitioning. At the same time, extreme threshold values also negatively affect system utilization within the blocks, *e.g.*, lower low-thresholds result in larger number of nearly empty blocks.

We note that appropriate values for these thresholds depend on the workload characteristics, *e.g.*, prior work [82, 83] has explored them for key-value stores. While Jiffy's contribution is to enable flexibility in data repartitioning, selecting threshold values is complementary to its design.

4. Similar to existing systems [24, 25, 39, 80], Jiffy can trivially scale its cluster capacity: if the number of free blocks available increase/decrease beyond a certain threshold, Jiffy adds/removes servers to adjust physical memory resources. Here, we focus only on fine-grained elasticity.

As such, Jiffy exposes them as configurable parameters.

2.3 Jiffy Implementation

Jiffy implementation builds on Pocket (§**??**), and as such, inherits its scalable and fault-tolerant metadata plane, multi-tiered data storage, system-wide storage capacity scaling, analytics execution model, etc. However, Jiffy implements hierarchical addressing, lease management and efficient data repartitioning (§**??**) to resolve unique challenges introduced by serverless environments. We now describe Jiffy interface (§**??**) and implementation (§2.3.2), focusing on these new features.

2.3.1 Jiffy Interface

We describe Jiffy interface in terms of its user-facing API (Table 2.1) and internal API (Figure 2.6).

User-facing API. Jiffy's user-facing interface (Table 2.1) 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 [24, 84], 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()`.

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

Fig. 2.6: **Jiffy Internal API.** The block interface is used internally in Jiffy to implement the data structure APIs (§**??**).

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

	Data Structure	Operators				
		`writeOp`	`readOp`	`deleteOp`	`getBlock`	`repartition`
Built-in	File (§2.4.1)	`write`	`read`	`-`	Route to block based on file offsets.	Not required
	FIFO Queue (§2.4.2)	`enqueue`	`dequeue`		`enqueue` to tail, `dequeue` to head block.	Not required
	KV-Store (§2.4.3)	`put`	`get`	`delete`	Route to block based on key hash.	Hash-based rep
Custom data structures.						

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* (Figure 2.6) which uniquely defines 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 (§**??**) and 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.

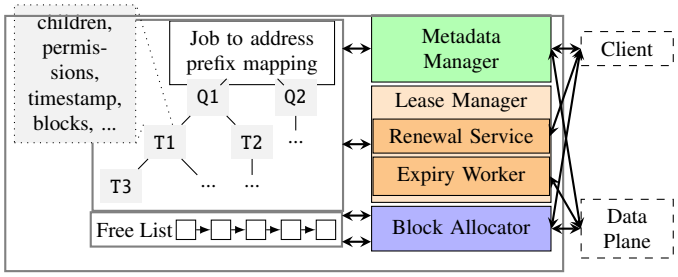


Fig. 2.7: **Jiffy controller.** See §2.3.2 for details.

2.3.2 System Implementation

Since Jiffy design builds on Pocket, its high-level design components are also similar, except for one difference: Jiffy combines the control and metadata planes into a unified control plane. We found this design choice allowed us to significantly simplify interactions between the control and metadata components, without affecting their performance. While this does couple their fault-domains, standard fault-tolerance mechanisms (*e.g.*, the one outlined in [39]) are still applicable to the unified

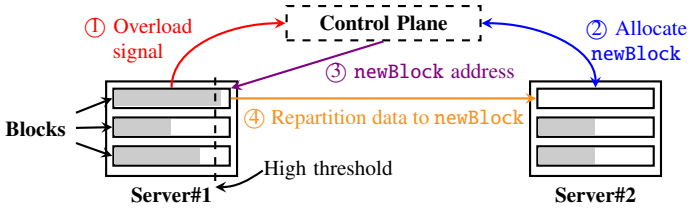


Fig. 2.8: **Data repartitioning on scaling up capacity.** Scaling down capacity employs a similar approach (§2.3.2).

control plane.

Control plane. The Jiffy controller (Figure 2.7) 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 variety of metadata for its address-prefix, including access permissions (for enforcing access control), timestamps (for lease renewal), a block-map (to locate the blocks associated with the address-prefix in the data plane), along with metadata to identify the data structure associated with the address-prefix and how data is partitioned across its blocks. The mapping between jobIDs (which uniquely identify jobs) and their address hierarchies is stored in a hash-table at the controller.

While the block allocator and metadata manager are similar to their counterparts in Pocket, the lease manager implements lifetime management in Jiffy. It comprises a lease renewal service that listens for renewal requests from jobs and updates the lease renewal timestamp of relevant nodes in its address hierarchy, and a lease expiry worker that periodically traverses all address hierarchies, marking nodes with timestamps older than the associated lease period as expired. Finally, Jiffy adopts mechanisms from Pocket to facilitate control plane scaling and fault tolerance; we refer the reader to [39] for details.

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

maps data structure operations to client handles that have subscribed to receive notifications for that operation.

Data repartitioning for a Jiffy data structure is implemented as follows: when a block's usage grows above the high threshold, the block sends a signal to the control plane, which, in turn, allocates a new block to the address-prefix and responds to the overloaded block with its location. The overloaded block then repartitions and moves part of its data to the new block (see Figure 2.8); 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 [85] at block granularity, and synchronously persisting data to external stores (*e.g.*, S3) at address-prefix granularity.

2.4 Programming Models on Jiffy

We now describe how Jiffy's built-in data structures (Table 2.2) enable many distributed programming frameworks atop serverless platforms (§2.4.1-§2.4.3).

2.4.1 Map-Reduce Model

A Map-Reduce (MR) program [69] 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 [69, 81, 86] 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. We describe Jiffy files next.

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 `read`s, and `write`s 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.

2.4.2 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 [70] as a reference dataflow execution engine, where channels can be files, shared memory FIFO queues, etc. 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 [71] 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 vertex scheduling, fault tolerance, and lease renewals for task address-prefixes. 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 availability of items produced by upstream tasks via notifications, as described below.

Jiffy Queues. The 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 FIFO 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 link-list, respectively. While, blocks can be both added and removed from a FIFO queue, queues do not need subsequent data repartitioning. Finally, the FIFO queue leverages Jiffy notifications to asynchronously detect when the there is data in the queue to consume, or space in the queue to add more items via subscriptions to `enqueue` and `dequeue`, respectively.

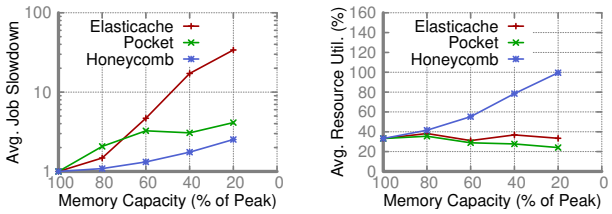
2.4.3 Piccolo

Piccolo [72] is a data-centric programming model that allows distributed compute machines to share distributed, mutable state. Piccolo kernel functions specify sequential application logic and share state with concurrent kernel functions via a KV interface, while centralized control functions create and coordinate both shared KV stores and kernel function instances. 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 run on a centralized master. The shared state is stored across Jiffy's KV-store data structures (described below). KV-stores may be created per kernel function, or shared across multiple functions, depending on the application needs. The master periodically renews leases for Jiffy KV-stores. Like 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 $[0, H\text{-}1]$ (H=1024 by default). The KV-store shards KV pairs across multiple Jiffy blocks, such that each block owns one or more hash-slots in this range. Note that a hash-slot is completely contained in a single block. The controller stores the mapping between the blocks and the hash slots managed by them; this metadata is, again, cached at the client and updated during resource scaling. Each block stores KV pairs that hash to its slots as a hash-table. The KV-store supports typical `get`, `put`, and `delete` operations as implementations of `readOp`, `writeOp` and `deleteOp` operators. The `getBlock` operator routes requests to KV-store blocks based on key-hashes.

Unlike files and queues, data needs to repartitioned for the KV-store when a block is added or removed. When a block is close-to-full, Jiffy reassigns half of its hash-slots to a new block, moves the corresponding key-value pairs to it, 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.



(a) Job performance (b) Resource utilization

Fig. 2.9: **Fine-grained task-level elasticity in Jiffy** enables (a) better job performance, and (b) higher resource utilization under constrained capacity. In (a), the slowdown is computed relative to the job completion time with 100% capacity (for this data point, Elasticache performance was 30% worse than Pocket, and Pocket performance was 5% worse than Jiffy). See §2.5.1 for details.

2.5 Evaluation

Jiffy is implemented in 25K lines of C++, with client libraries in C++, Python and Java (∼1K LOC each); this is in addition to the original Pocket code. In this section, we evaluate Jiffy to demonstrate its benefits (§2.5.1, §2.5.2) and to understand the contribution of individual Jiffy mechanisms to its overall performance (§**??**). Finally, we evaluate Jiffy controller overheads in §A.1.1.

Experimental setup. Unless otherwise specified, each intermediate storage system in our experiments is deployed across 10 m4.16xlarge EC2 [87] instances, while serverless applications are deployed across AWS Lambda [87] instances. Since Jiffy builds on Pocket design, it supports addition of new instances to increase the system-wide capacity. However, our experiments do not evaluate overheads for doing so, since, it is orthogonal to Jiffy's goals; we specifically focus on multiplexing available system storage capacity for higher utilization and reducing the need for adding more capacity. Jiffy employs 128MB blocks, 1s lease duration and 5% (low) and 95% (high) as thresholds for data repartitioning.

2.5.1 Benefits of Jiffy

Jiffy enables fine-grained resource allocation for serverless analytics. We demonstrate the benefits of this approach to job performance and resource utilization for roughly 50,000 jobs across 100 randomly chosen tenants over a randomly selected 5 hour window in the Snowflake workload[5] [61].

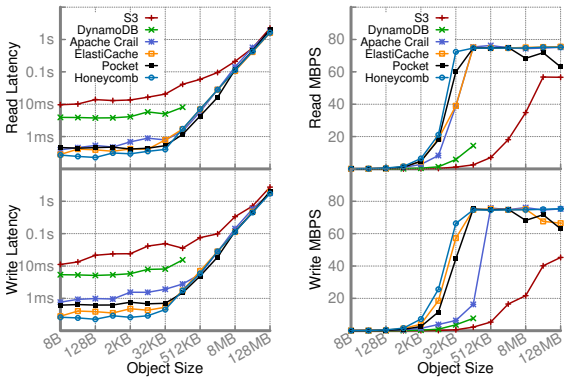
We compare Jiffy (with the MR programming model, §**??**) against Elasticache [80] and Pocket [39]. Elasticache represents systems that provision resources for *all* jobs. Since Elasticache does not sup-

5. We were unable to evaluate the entire 14 day window with $>$ 2000 tenants due to intractable cost overheads.

port multiple storage tiers, if available capacity is insufficient, jobs must write their data to external stores like S3 [88]. Pocket, on the other hand, reserves and reclaims resources at *job* granularity; if available capacity is insufficient, Pocket allocates resources on secondary storage (SSD). Note that Pocket's utilization can sometimes be *lower* than Elasticache, since it provisions for the peak of each job *separately*, sacrificing utilization for job-level isolation. Finally, our evaluation for Pocket places its control and metadata services on the same server to ensure a fair comparison with Jiffy's unified control plane.

Impact of fine-grained elasticity on job performance. We demonstrate this impact by constraining the amount of available capacity at the intermediate store for the Snowflake workload. Figure 2.9(a) shows the average job slowdown as the capacity is reduced to a certain percentage of the peak utilization for the workload within the evaluated time window (i.e., across all jobs). With Elasticache, job performance suffers significantly when the intermediate data grows larger than capacity ($34\times$ slowdown at 20% capacity), since the data must now be accessed from S3. With Pocket, the data spills to SSD when the allocated capacity at the DRAM-tier (during job registration) is insufficient. While the slowdown is less severe due to its efficient tiered-storage, jobs still experience a $> 4.1\times$ slowdown at 20% capacity. Finally, Jiffy observes much lower job performance degradation with constrained capacity ($< 2.5\times$ at 20% capacity). This is because task-level elasticity and lease based reclamation of faster storage capacity allows Jiffy to efficiently multiplex capacity across multiple jobs at a much finer granularity than Pocket. This, in turn, significantly reduces data spilling over to a slower storage tier in Jiffy compared to Pocket. We confirm this intuition further by studying the impact of fine-grained elasticity on resource utilization next.

Impact of fine-grained elasticity on resource utilization. Figure 2.9(b) shows the resource utilization across the compared systems under constrained capacity. While the resource utilization for Elasticache and Pocket either decreases or remains the same as the system capacity is reduced, resource utilization *improves* for Jiffy. This is because Pocket and Elasticache provision capacity (at job or coarser granularity), and the unused capacity is wasted, regardless of the total system capacity. In contrast, Jiffy is able to better multiplex the available capacity across various jobs owing to its fine-grained elasticity and lease-based reclamation of unused capacity. By making better use of available capacity, Jiffy ensures that a much smaller fraction of data spills over to SSD, resulting in



(a) Latency (b) MBPS

Fig. 2.10: **Jiffy performance comparison with existing storage systems (§2.5.2).** Despite providing the additional benefits demonstrated in §2.5.1, Jiffy performs as well as or outperforms state-of-the-art storage systems for serverless analytics.

better performance in Figure 2.9(a).

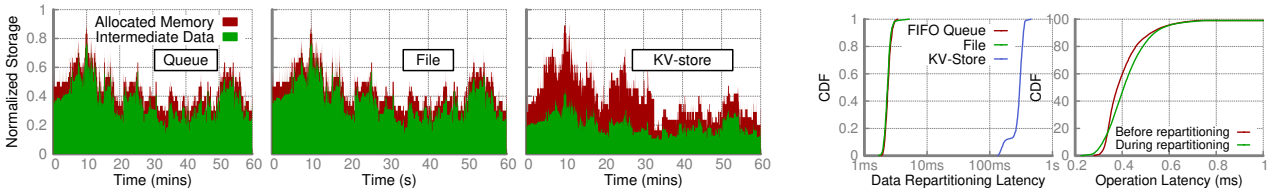
2.5.2 Performance Benchmarks for Six Systems

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

To measure latency and throughput for the above systems, we profiled synchronous operations issued from an AWS Lambda instance using a single-threaded client. Figure 2.10 shows that in-memory data stores like Elasticache, Pocket and Apache Crail achieve low-latency (sub-millisecond) and high-throughput. In contrast, persistent data stores like S3 and DynamoDB observe significantly higher latencies and lower throughput; note that DynamoDB only supports objects up to 128KB. Jiffy matches the performance achieved by state-of-the-art in-memory data stores, while additionally providing the benefits outlined in §2.5.1.

2.5.3 Understanding Jiffy Benefits

Figure 2.9 already shows how fine-grained elasticity in Jiffy allows it to achieve performance and resource utilization gains over the compared state-of-the-art systems. As noted earlier, this fine-grained elasticity is enabled by hierarchical virtual addressing combined with with flexible data



(a) Efficient lifetime management

(b) Efficient data repartitioning

Fig. 2.11: **Jiffy data lifetime-management and data repartitioning.** (a) Jiffy enables fine-grained elasticity via lease-based lifetime management for its built-in data structures, FIFO Queue (left), File (center) and KV-store (right), reclaiming resources from tasks as soon as their leases expire. (b) Jiffy facilitates efficient data repartitioning for its data structures when their allocation is scaled up, with repartitioning for a single block completing in 2-500ms (left). Moreover, Jiffy latency for 100KB `gets` is minimally impacted during KV-store repartitioning. Note: plots in (a), (b) share a common y-axis; x-axis for (c, left) is in log-scale.

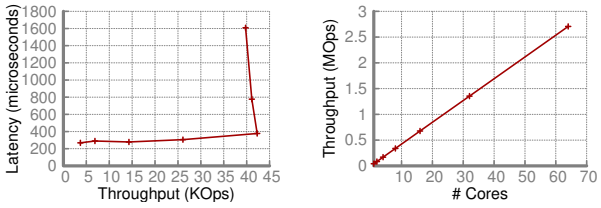
lifetime management and data repartitioning in Jiffy. In this section, we evaluate their impact in isolation.

Fine-grained elasticity via data lifetime management. Unlike traditional storage systems, Jiffy's lease-based data lifetime management allows it to reclaim unused resources from jobs, and potentially assign them to other jobs that might need them. Coupled with fine-grained resource allocations and efficient data repartitioning, this enables fine-grained elasticity for serverless jobs running on Jiffy. To understand how, we evaluate storage allocation across different Jiffy data structures (Figure 2.11(a)) when subjected to Snowflake workload from Figure 2.2.

FIFO queue and file observe seamless elasticity in allocated resources as intermediate data is written to them since they do not require repartitioning. The allocated capacity exceeds the intermediate data size for the data structures by only a small amount; this accounts for the additional metadata stored at each of the blocks (*e.g.*, object metadata for the items enqueued in the FIFO queue, etc.), along with unused space within the head/tail blocks. For the KV-store, the inserted keys were sampled from a Zipf distribution over the keyspace since the Snowflake dataset does not provide access patterns. Due to the skew, a few Jiffy blocks receive most of the key-value pairs, and repeatedly split across newly allocated blocks when their used capacity grows too high. The allocated capacity is therefore higher than the dataset size, since the used capacity is low for most blocks owing to the Zipf key sampling; this corresponds to the worst-case for the KV-Store. However, Jiffy's lease mechanism reclaims resources allocated to the data structures soon after their utility is over, ensuring that the overheads are short-lived.

Efficient elastic scaling via flexible data repartitioning. A key contributor for the fine-grained resource elasticity achieved by Jiffy is its flexible but efficient data repartitioning approach. Figure 2.11(b) shows the CDF of data repartitioning latency per block across the three data structures, when subjected to the Snowflake workload from above. The data repartitioning latency shown here corresponds to the total time taken from the detection of an overloaded/underloaded block to the end of data repartitioning. The storage server takes ∼1-1.5ms to connect to the controller, and two round-trips (100-200μs in EC2) to trigger allocation/reclamation of data blocks and update for partitioning metadata at the controller. Unlike FIFO Queue and File, KV-Store also requires repartitioning data across blocks. However, since repartitioning a single block only requires moving only half the block capacity (∼64MB), Jiffy is able to repartition the data in a few hundred milliseconds over 10Gbps links. As such, Jiffy repartitions data within a block for its built-in data structures with very low latency (2-500ms).

Finally, we also note that Jiffy does not block data structure operations during data repartitioning. In fact, these operations are minimally impacted during this period: Figure 2.11(b) shows that the CDF for 100KB `get` operations on the KV-Store prior to and during scaling are almost identical.



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

(b) Controller throughput scaling with multiple cores.

Fig. 2.12: **Jiffy controller performance.** Details in Appendix A.1.1.

2.5.4 Controller Overheads

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

workloads and those evaluated in [39].

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

Storage overheads. The task-level metadata storage in Honeycomb has an overhead of only 64 bytes of fixed metadata per task and 8 bytes per block. For the default 128MB blocks used in Jiffy, the metadata storage overhead is a tiny fraction of the total storage ($< 0.00005 - 0.0001\%$).

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

Data center network bandwidth is approaching that of intra-server resource interconnects [89, 90], and is soon poised to surpass it [**?**]. This has driven significant academic [2–9, 31, 32, 91] and indus-

try [92–97] interest in memory disaggregation, where compute and memory are physically separated into network-attached *resource blades*, drastically improving resource utilization, hardware heterogeneity, resource elasticity and failure handling compared to traditional data center architectures.

However, memory disaggregation is challenging due to three requirements. First, access to remote memory must have low latency and high throughput — prior work [2,3,31,32] have targeted 10 μs latency and 100 Gbps bandwidth per compute blade to minimize application performance degradation. Second, both memory and compute resources available to applications must scale elastically, in keeping with the promise of disaggregation. Finally, wide adoption and immediate deployment requires support for unmodified applications.

Despite years of research towards enabling memory disaggregation, none of the known approaches support all three requirements simultaneously (§**??**). Most approaches require application modifications due to changes in hardware [7, 93, 94, 98], programming model [72, 99], or memory interface [25, 27, 100]. Recent approaches that enable transparent access to disaggregated memory [2, 31, 32] limit application compute elasticity — processes are limited to compute resources on a single compute blade to avoid cache coherence traffic over the network due to performance concerns.

We present MIND, the first memory management system for rack-scale memory disaggregation that simultaneously meets all three requirements for disaggregated memory. Our key idea is to place the logic and metadata for memory management *in the network fabric*. MIND's design builds on the observation that the network fabric in the disaggregated memory architecture is essentially a CPU-memory interconnect. In MIND, centrally-placed in-network processing devices like programmable network switches [101–103] therefore assume the role of the MMU to enable a high-performance shared memory abstraction. Since MIND realizes the logic and metadata for memory management in programmable hardware at line rate [101], latency and bandwidth overheads are minimal.

Realizing in-network memory management, however, requires working with the unique constraints imposed by programmable switch ASICs. First, today's switch ASICs only have a few megabytes of on-chip memory, making it challenging to store traditional page tables for potentially terabytes of disaggregated memory. Second, switch ASICs only permit a few cycles of limited computations per packet to ensure line-rate processing, while cache coherence may require complex state transition logic for each cached block. Finally, these ASICs [104] have staged packet

processing pipelines where compute and memory resources are spread across multiple physically decoupled match-action stages, introducing interesting challenges in partitioning and placing the logic and metadata for memory management across them.

To meet the three requirements of memory disaggregation, MIND effectively navigates the above constraints and explores the capabilities of today's programmable switches to enable in-network memory management for disaggregated architectures. It does so through a principled redesign of traditional memory management:

- MIND employs a *global virtual address space* shared by all processes, range partitioned across memory blades to minimize the number of address translation entries that need to be stored in the on-chip memory of switch ASIC. At the same time, it employs a physical memory allocation mechanism that load balances allocations across memory blades for high memory throughput (§3.4.1).
- MIND features domain-based memory protection inspired by capability-based schemes [**?**, **?**, 105] that enables fine-grained and flexible protection by decoupling the storage of memory permissions from address translation entries. Interestingly, such a decoupling actually *reduces* the on-chip memory overheads at the switch ASIC (§3.4.2).
- MIND adapts directory-based MSI coherence [106] to the in-network setting. To mitigate the network overheads of cache coherence, MIND exploits network-centric hardware primitives such as multicast in the switch ASIC to efficiently realize its coherence protocol (§3.4.3).
- We find that the limited on-chip memory at the switch ASIC forces the cache directory to track memory regions at coarse granularities, which in turn results in performance degradation due to *false invalidations* of pages in those regions (§3.4.3). We address this through a novel Bounded Splitting algorithm (§3.5) that dynamically sizes memory regions to bound both the switch storage requirements as well as performance overheads due to false invalidations.

We realize MIND design on a disaggregated cluster emulated using traditional servers connected by a programmable switch. Our results show that MIND enables transparent resource elasticity for real-world workloads while matching the performance for prior memory disaggregation proposals (§3.14).

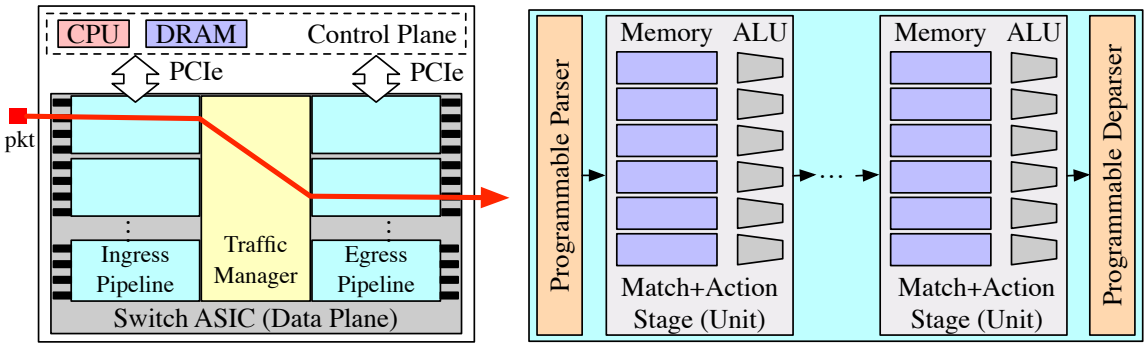


Fig. 3.1: **Enabling technologies for MIND.** (left) Programmable switch architecture and (right) Switch ingress/egress pipeline.

Table 3.1: **In-network technology tradeoffs.** See §**??** for details.

	RMT	FPGA	Custom ASIC	CPU
Line-rate	✓	✓	✓	✗
Available	✓	✓	✗	✓
Low Power	✓	✗	✓	✗
Low Cost	✓	✗	✓	✗

We also find that while MIND is competitive with compared systems, workloads with high read-write contention experience sub-linear scaling with more threads due to limitations of current hardware. Current x86 architectures preclude realization of relaxed consistency models commonly employed in shared memory systems [107], and the switch TCAM capacity is close to saturated with cache directory entries for such workloads. We discuss approaches that could enable better scaling with future improvements in switch ASIC and compute blade architectures in §3.8.

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. Similar to prior work [2–9], 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.

We now briefly describe MIND's enabling technologies.

Programmable switches. In recent years, programmable switches have evolved along two well-coordinated directions: development of P4 [104, 108, 109], a flexible programming language for network switches, and design of switch hardware that can be programmed with it [102, 103, 110, 111]. These switches host an application-specific integrated circuit (ASIC), along with a general purpose CPU with DRAM, as shown in Figure 3.1 (left). The switch ASIC comprises ingress pipelines, a

traffic manager and egress pipelines, which process packets in that order. Programmability via P4 is facilitated through a programmable parser and match-action units in the ingress/egress pipelines, as shown in Figure 3.1 (right). Specifically, the program defines how the parser parses packet headers to extract a set of fields, and multiple stages of match-action units (each with limited TCAM/SRAM and ALUs) 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 the above focuses on switch ASICs with Reconfigurable Match Action Tables (RMTs) [110], it is possible to realize MIND using FPGAs, custom ASICs, or even general purpose CPUs. While each of them exposes different tradeoffs (Table 3.1), 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 [**?**, 112–115] and distributed shared memory (DSM) architectures [107, 116–119]. 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 additionally has a cache to facilitate performance for frequently accessed memory blocks. We distinguish memory blocks from pages since caching granularities, i.e., block, can be different from memory access granularities, i.e., page.

With the copies of blocks potentially residing across multiple node caches, coherence protocols [106, 120–123] are required to ensure each node operates on the latest version of a block. In popular directory-based invalidation protocols like MSI [106] (used in MIND), each memory block can be in one of three states: **M**odified (**M**), where a single node has exclusive read and write access to (or, "owns") the block, **S**hared (**S**), where one or more caches have shared read-only access to the block, and **I**nvalid (**I**), where the block is not present in any cache. A directory tracks the state of each block, along with the list of nodes ("sharer list") 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; it triggers a state transition and potential invalidation of the block across other nodes, followed by retrieving the block from the node that owns the block.

While it is possible to realize more sophisticated coherence protocols, we restrict our focus to MSI in this work due to its simplicity — we defer a discussion of other protocols to §3.8.

As outlined in §4.2, 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, (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 the 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, and defer the discussion of other related work to §3.9.

Transparent designs. While transparent 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 in §**??** 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, *e.g.*, 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 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 [2–4, 32, 92, 93], i.e., while compute blades cache data locally, different compute blades do not share memory to avoid sending

Table 3.2: **Parallels between memory & networking primitives.**

Virtual Memory	⟺	Networking
Memory allocation		IP assignment
Address translation		IP forwarding
Memory protection		Access control
Cache invalidations		Multicast

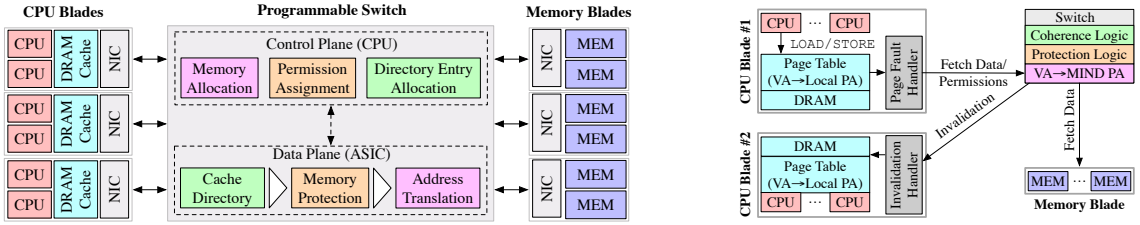


Fig. 3.2: **(left) High-level MIND architecture, and, (right) data flow for memory accesses in MIND.** See §3.3.2 for details.

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.

3.3 MIND Overview

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 [101–103] permit the implementation of such logic in integrated programmable ASICs. We show that these ASICs are capable of executing it at line rate even for multi-terabit traffic. In fact, many memory management functionalities have similar counterparts in networking (Table 3.2), 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, as we show in §**??**.

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; (*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.

3.3.1 Design Principles

MIND follows three principles to meet the goals for memory disaggregation outlined in §4.2:

P1. *Decouple memory management* functionalities to ensure each can be optimized for their specific goals.

P2. Leverage *global view* of the disaggregated memory subsystem at a centralized control plane to compute optimal policies for each memory management functionality.

P3. Exploit *network-centric hardware primitives* at the programmable switch ASIC to efficiently realize policies computed using **P2**.

MIND follows principle **P1** to decouple memory allocation from addressing (§3.4.1), address translation from memory protection (§3.4.2), cache accesses and eviction from coherence protocol execution (§3.4.3), and employs principles **P2** and **P3** to efficiently realize their goals. Note that traditional server-based OSes are unable to leverage these principles due to their reliance on *fixed-function* hardware modules such as the MMU and memory controller — most common implementations of such modules couple many memory management functionalities (*e.g.*, address translation and memory protection in page-table walkers) for a host of complexity, performance, and power reasons [124–126].

3.3.2 Design Overview

MIND exposes a *transparent virtual memory* abstraction to applications, similar to server-based OSes. Unlike prior disaggregated memory designs, MIND places all logic and metadata for memory management in the network, instead of CPU or memory blades [31, 32], or a separate global controller [2].

Figure 3.2 (left) provides an overview of MIND design, while Figure 3.2 (right) depicts the data flow for memory accesses in MIND. The *CPU blades* run user processes and threads, and possess

a small amount of local DRAM that is used as a cache. All memory allocations (*e.g.*, via `mmap` or `sbrk`) and deallocations (*e.g.*, via `munmap`) from the user processes are intercepted at the CPU blade, and forwarded to the *switch control plane*. The control plane possesses a global view of the system, which it leverages to perform memory allocations, permission assignments, etc., using principle **P2**, and respond to the user process. All memory LOAD/STORE operations from the user processes are handled by the CPU blade cache (§3.4.3). The cache is virtually addressed[1], and stores permissions for cached pages to enforce memory protection. If a page is not locally cached, the CPU blade triggers a page-fault and fetches the page from *memory blades* using RDMA requests, evicting other cached pages if necessary. Similarly, if the memory access requires an update to a cached block's coherence state (*e.g.*, STORE on a Shared or **S** block), a page-fault is triggered to initiate cache coherence logic at the switch. Note that the page-fault based design requires MIND to perform page-level remote accesses, although future CPU architectures may enable more flexible access granularities (§3.8).

Since the CPU blade does not store memory management metadata, the RDMA requests are for virtual addresses and do not contain endpoint information (*e.g.*, IP address) for the memory blade that holds the page. Consequently, the *switch data plane* intercepts these requests. It then performs necessary cache coherence logic, including lookups/updates to the cache directory and cache invalidations on other CPU blades (§3.4.3, §3.5). In parallel, the data plane also ensures the requesting process has permissions to access the requested page (§3.4.2). If no CPU blade cache holds the page, the data plane translates the virtual addresses to physical addresses (§3.4.1), forwarding the request to the appropriate memory blade. These memory management functionalities are decoupled as separate modules following **P1**, and efficiently realized in the switch ASIC following **P3**.

In MIND's design, the memory blades simply store the actual memory pages, and serve RDMA requests for physical pages. Unlike prior works that employ RPC handlers and polling threads [2], MIND leverages one-sided RDMA operations [27] to obviate the need for any CPU cycles on the disaggregated memory blades. This is a step towards true hardware resource disaggregation, where memory blades need no longer be equipped with any general-purpose CPUs.

1. Note that while it is hidden from applications, CPU blades maintain a local page-based virtual memory abstraction to translate MIND virtual addresses to physical addresses for cached pages in local DRAM (Figure 3.2 (right)).

3.4 MIND Design

Placing memory management logic and metadata in the network provides the opportunity for simultaneously achieving memory performance and resource elasticity. We now describe how MIND optimizes for the individual goals of memory allocation and addressing (§3.4.1), memory protection (§3.4.2), and cache coherence (§3.4.3), while operating under the constraints of programmable switches. Finally, we detail how MIND handles failures (§3.4.4).

3.4.1 Memory Allocation & Addressing

Traditional virtual memory uses fixed sized pages as the basic units for both translation and protection; as a result, it cannot achieve the goal of storage efficiency without increasing memory fragmentation: small pages reduce memory fragmentation but require more translation entries, and vice versa. Following **P1**, MIND overcomes this by *decoupling* address translation and protection. That is, MIND's translation is blade-based while protection is `vma` based (§3.4.2).

Storage-efficient address translation. MIND eschews page-based protection but uses a *single global virtual address-space* across all processes, allowing translation entries to be shared across them. Our approach builds on decades of research on virtual memory designs that also exploit a single address space [**?**, **?**, 99, 107, 127], but adds techniques to minimize storage overheads for in-network address translation. In particular, MIND *range partitions* the virtual address space across different memory blades, such that the entire virtual-address space maps to a contiguous range of physical address space. This allows us to use a single translation entry for each memory blade: any virtual address that falls within its range can be directly routed to that memory blade, minimizing the storage required on switch data plane. In MIND, this mapping only changes when new memory blades join, old ones retire or if memory is moved between blades.

Balanced memory allocation & reduced fragmentation. MIND's control plane, leveraging its global view of allocations (**P2**), tracks the total amount of memory allocated on each memory blade and places a new allocation on the blade with the least allocation, to achieve near-optimal load-balancing. We validate this empirically in §3.14.

Moreover, since there is a one-to-one mapping between virtual and physical addresses within a particular memory blade, MIND minimizes external fragmentation at each memory blade by using

traditional virtual memory allocation schemes that have evolved to facilitate the same, *e.g.*, first-fit allocator in our implementation [128]. The result of memory allocation is a virtual memory area (`vma`), identified by the base virtual address and length of the area, *e.g.*, `<0x00007f84b862d33b, 0x400>` for a 1KB area. As will be elaborated in §3.4.2, `vma` is the basic unit of protection in MIND. This allows multiple processes to have non-overlapping `vma`s on the same blade, minimizing memory fragmentation.

Isolation. We note that MIND's global virtual address-space does not compromise on *isolation* between processes. First, since the switch intercepts allocation requests across all compute blades, and possesses a global view of valid allocations at any time, it can easily ensure allocations are non-overlapping across different processes. Second, we show in §3.4.2 that MIND's `vma`-based protection allows flexible access control between processes in a single global virtual address-space.

Transparency via outlier entries. MIND's one-to-one mapping between virtual and physical addresses does not preclude supporting unmodified applications with static virtual addresses embedded within their binaries, or OS optimizations such as page migration [129], i.e., moving pages from one memory blade to another. MIND maintains separate *range-based* address translations [130] for physical memory regions that correspond to static virtual addresses or migrated memory. These *outlier* entries are stored succinctly in the switch TCAM, where the TCAM's longest-prefix matching (LPM) property ensures that only the most specific entry (i.e., one with the longest prefix) is considered when translating a virtual address, ensuring correctness.

3.4.2 Memory Protection

As MIND decouples translation and protection, it uses a separate table to store memory protection entries in the data plane. Consequently, an application can assign access permissions to a `vma` of any size. The size of this protection table is proportional to the number of `vma`s. We find this number is reasonably small in our experiments and the protection table can easily fit in the switch ASIC even for a wide range of memory-intensive applications (§3.14). This is because the first-fit allocator and Linux's `glibc` allocation requests [131] do a good job of ensuring `vma`s are large and contiguous.

Fine-grained, flexible memory protection. Similar to prior work on capability-based systems [105, 127], MIND supports two key abstractions: *protection domains* and *permission classes*. Protection

domains identify the entity that may (or may not) have permissions to access a particular memory region of arbitrary size, while the permission class identifies what the entity can do to the memory region. MIND's control plane exposes a set of APIs for memory allocation and permission changes that allows an application to specify a protection domain identifier (PDID) for an arbitrary virtual memory area (vma) and assign a permission class (PC) to the pair <PDID, vma>. The mapping <PDID, vma> $\rightarrow$ PC is stored as an entry in the protection table in the data plane. For existing applications, MIND simply takes the process identifier (PID) as the PDID, and uses Linux memory permissions (*e.g.*, read-only, read-write, etc.) as permission classes. Note that MIND *can* support richer memory protection semantics than traditional OSes, *e.g.*, user programs that serve multiple client sessions, such as ssh servers or database services, can assign a separate protection domain per session to prevent one session from accessing data from other sessions [127].

Following principle **P3**, we leverage TCAM-based parallel range matches in the programmable switch ASIC — typically used for IP subnet matches — to efficiently support fine-grained matching for <PDID, vma> entries embedded in memory access requests and obtain corresponding the permission class (PC). If there is a mismatch between PC and the memory access type, or the <PDID, vma> entry does not exist, the request is rejected.

Optimizing for TCAM storage. One limitation of TCAM is that each of its entries can only match power-of-two ranges. MIND overcomes this by splitting an arbitrary-sized virtual address range into multiple power-of-two-sized entries. Note that the number of entries required for a range of size s is upper-bounded by $\lceil \log_2(s) \rceil$. In order to meet our goal of storage efficiency in the switch data plane, the control plane (1) only performs virtual address allocations that are aligned with the power-of-two sizes to ensure each region can be represented using a single TCAM entry, and (2) coalesces adjacent entries with if they belong to the same protection domain and have the same permission class. Interestingly, memory allocations requested by underlying libraries (*e.g.*, `glibc`) are mostly in power-of-two sizes anyway, enabling storage-efficiency for TCAM entries.

3.4.3 Caching & Cache Coherence

In MIND design, while the caches[2] reside on compute blades, the coherence directory and logic reside in the switch. This already permits access to the cache directory in half a round-trip, significantly reducing the latency overheads for the coherence protocol execution. For MSI protocol, even the most expensive and relatively uncommon transitions (i.e., **M→S/M**) incur two round-trips, while common transitions incur only a single round-trip, as we show in §3.7.2. While performance is a primary objective in MIND's cache coherence, the coherence protocol must also be realizable under the compute and memory constraints of switch ASICs. We now outline challenges in adapting traditional cache management to our in-network setting, along with how MIND resolves them.

Storage vs. performance tradeoff

Traditional caching and cache coherence mechanisms applied to MIND expose a tradeoff between cache performance and the storage efficiency at the switch data plane. Specifically, reducing the number of directory entries requires larger cache granularities (i.e., larger memory blocks), which results in worse performance. For instance, when large (*e.g.*, 2 MB) memory blocks are used, updating a small (*e.g.*, 4 KB) region within the block will invalidate the entire block. We refer to these invalidations as *false invalidations* — dirty pages invalidated along with the requested page because there are in the same memory block tracked by a directory entry. This leads to wastage in both memory bandwidth and cache capacity, i.e., fewer frequently accessed data items in the cache. We empirically highlight this tradeoff in §C.3.

MIND addresses this challenge using two approaches: it decouples the cache and directory granularities (following principle **P1**), and appropriately sizes the memory region tracked by each cache directory entry leveraging the global view of memory traffic at the control plane (following principle **P2**), as we describe next.

Decoupling cache access & directory entry granularities. Our first approach employs principle **P1** — decoupling the granularity of cache (and memory) accesses from the granularity at which

2. Note that we use the term 'cache' to refer to the DRAM at the CPU blade under the partial disaggregation model, and not hardware (L1/L2/L3) caches.

cache coherence is performed. This allows memory accesses (*e.g.*, evictions or remote memory reads) to be performed at finer granularities, while directory entries are tracked at coarser granularities. Specifically, accesses to the local DRAM cache at CPU blades, and even the movement of data between the CPU caches and memory blades, occur at the fine page granularity (4 KB in MIND, similar to prior work [2, 31, 32]). However, the coherence protocol tracks directory entries (stored at the switch data plane) at larger, variable-sized *region* granularities — when a 4 KB page is cached at a CPU blade, MIND creates a directory entry for the region that contains the page. An invalidation of the region triggers an invalidation of all dirty pages in the region, as tracked by the individual CPU blades that cache them.

Storage & performance-efficient sizing of *regions*. Even with the decoupling described above, the region *sizes* still expose a tension between coherence performance (*e.g.*, larger false invalidation counts due to larger region sizes) and directory storage efficiency (*e.g.*, more directory entries due to smaller region sizes). To appropriately size regions, MIND leverages global view of memory traffic at the switch control plane (**P2**). Briefly, MIND starts each directory entry with a very large memory region; when the overhead due to false invalidation is high, it splits the region and creates a new directory entry. It does so repeatedly, until either the overhead is below a predetermined threshold, or the region size reaches 4 KB, i.e., the page size. In doing so, MIND dynamically customizes the region sizes to resolve the tension between performance and directory storage efficiency using a novel Bounded Splitting algorithm — we defer its details to §3.5.

In-Network Coherence Protocol

Due to the limited computational capability at the switch ASIC, MIND employs the simple directory-based MSI coherence protocol [106]. While we defer the implementation details of the coherence protocol to §3.6.3, we highlight here how MIND employs network-centric hardware primitives to efficiently realize the coherence protocol in the switch, leveraging principle **P3**. Specifically, several state transitions in the MSI protocol require generating invalidation requests to CPU blades that have shared access to a region, to ensure correctness. To facilitate this in a network-efficient manner, we leverage *multicast* functionality supported natively in most switches — we create a multicast group

for all CPU blades in the rack, and send an invalidation request containing the list of sharers to the group. However, broadcasting invalidations to blades not in the sharer list would consume unnecessary network bandwidth. As such, we embed the sharer list within the invalidation request, and drop requests in the egress path of the switch data plane if the output port does not lead to a blade in the sharer list.

3.4.4 Handling Failures

We now discuss how MIND handles failures at different components in our disaggregated architecture.

CPU, memory blade and switch failures. MIND does not innovate on fault-tolerance for CPU and memory blade failures: mechanisms developed in prior work [2, 32, 91] for fault tolerance can be readily adapted to our design. To handle switch failures, we consistently replicate the control plane at a backup switch — on a failure, the data plane state is reconstructed at the backup switch using the control plane state. Since the control plane is only updated infrequently due to metadata operations (*e.g.*, system calls), the overhead added due to such replication in minimal.

Communication failures. MIND uses ACKs and timeouts to detect packet losses. When a memory access triggers invalidations, the requesting compute blade waits for ACKs indicating successful invalidation from all sharers, and resends the request if timeout occurs. If the compute blade does not receive an ACK even after a predefined number of retransmissions, it sends a `reset` message for the corresponding virtual address to the switch control plane. This, in turn, forces all compute blades to flush their data for that address and removes the corresponding cache directory entry in the data plane. This `reset` mechanism prevents deadlocks when compute blades fail in the middle of a cache coherence state transition.

3.5 Bounded Splitting: Algorithm & Analysis

We next provide details and a formal analysis of the bounded splitting algorithm used by MIND to dynamically determine the memory region size tracked by each cache directory entry. This algorithm is a key component of MIND's cache coherence design outlined in §3.4.3.

3.5.1 Algorithm

The bounded splitting algorithm starts by partitioning the entire virtual address space into N contiguous regions of size M pages each. It then works in disjoint *epochs* of equal length. In each epoch, it tracks the total number of times any page is falsely invalidated within the epoch — we refer to this as the *false invalidation count* — for every region.

Bounded Splitting uses false invalidation count as a measure of the performance overhead — we characterize the impact of false invalidations on performance in §3.14. It therefore seeks to keep the count below a threshold, denoted by t. If any region has a false invalidation count $> t$ in an epoch, it splits that region into two equal halves and creates a new directory entry accordingly. It bounds the smallest size of any memory region to 4KB (the page size), ensuring that any M sized block is split at most $\log_2 M$ times over as many epochs. Note that for 4KB regions, the number of falsely invalidated pages is trivially zero; however, maintaining all regions at that size would require storing $N \cdot M$ directory entries at the switch, which is impractically large.

Stability assumptions. The bounded splitting is based on two implicit assumptions related to the epoch length:

- the access pattern across the various memory regions remains stable for at least $O(\log_2 M)$ epochs, and
- the set of allocated pages remains unchanged across $O(\log_2 M)$ epochs.

We show in §3.14 that appropriate epoch sizing allows us to ensure both assumptions for our evaluated workloads.

3.5.2 Performance Bounds

The key challenge in bounded splitting is bounding the number of directory entries that must be stored, one per memory region. The total *worst-case* number of regions depends on two factors: (i) worst-case number of *sub*-regions generated by each M-sized region, and (ii) the value of t. We first analyze the worst-case bound on the number of sub-regions per M-sized region, and then bound the worst-case for total number of regions overall (and therefore, the total number of directory entries) by appropriately setting the value of t.

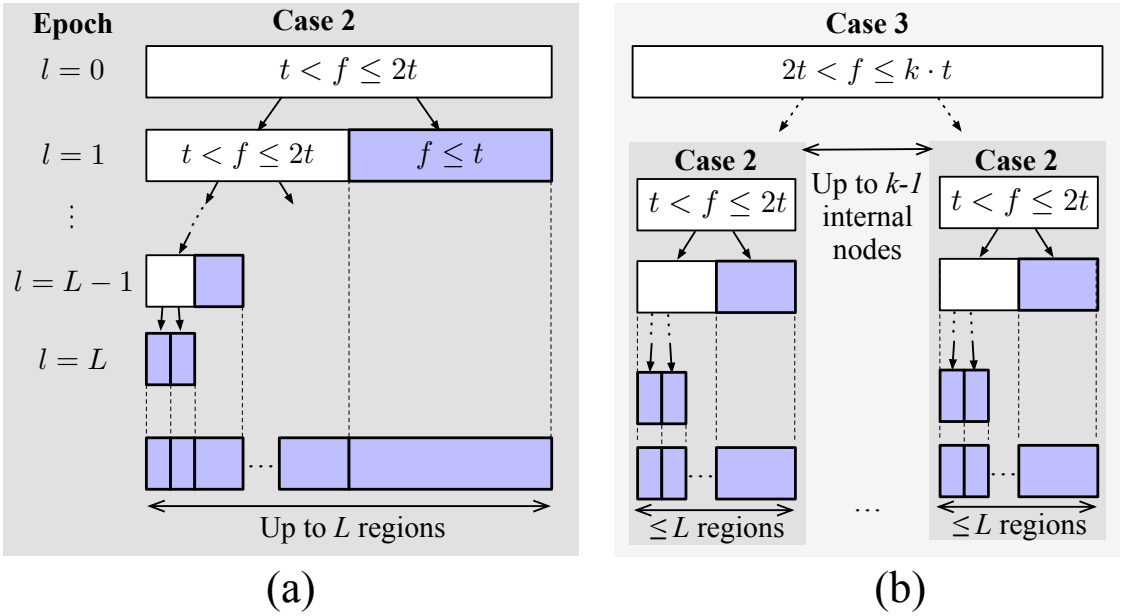


Fig. 3.3: **Splitting process for cache blocks depicted as a binary tree.** Note that $L = \log_2 M$; see §3.5.2 for details.

Bounding the number of sub-regions per M-sized region. We establish the worst-case bound in the following theorem:

Theorem 3.5.1. *The number of sub-regions for an M-sized region with false invalidation count f is upper-bounded by $S = (\lceil \frac{f}{t} \rceil - 1) \cdot (1 + \log_2 M)$.*

Proof. In the bounded splitting algorithm, we dynamically manage sizes of each memory region, splitting it into two smaller regions until the false invalidation count for the region is less than t. As noted above, since we limit the smallest region size to 4KB, an M-sized region may be split at most $\log_2 M$ times, across as many epochs. Figure 3.3 depicts the splitting process as a binary tree across the various epochs, where the level of the tree l denotes the epoch index ($0 \leq l < \log_2 M$). For the sake of exposition, we set $M = 2$MB. We leverage two observations to prove the above bound:

- **O1:** Splitting a region can only *decrease* the false invalidation count across the two splits, i.e., if a region with f false invalidation count is split into two regions with f' and f'' false invalidation count, then $f' + f'' \leq f$.
- **O2:** For a 4KB region, the false invalidation count is zero.

The maximum number of regions S generated by splitting an M-sized region can be categorized into three cases:

Case 1: $f \leq t$**.** Since false invalidation count is already below the threshold, the region does not need to be split, i.e., $S = 1$.

Case 2: $t < f \leq 2t$**.** To bring the false invalidation count below t, the region will be split into two. Due to observation **O1**, there are two possibilities: (i) both resulting regions have false invalidation count $< t$, or (ii) one region r_1 still has false invalidation count $> t$ while the other region r_2 has false invalidation count $< t$. For (i), the resulting regions do not need to be split any further, while for (ii), r_1 must be split further in the next epoch. In the worst case, the splits will continue for at most $\log_2 M$ epochs — when the region size reaches 4KB, no further splits will be required (due to observation **O2**). Thus, $S = 1 + \log_2 M$, as shown in Figure 3.3 (a).

Case 3: $2t < f \leq k \cdot t$**, where** $k = \lceil \frac{f}{t} \rceil$**.** The worst-case scenario that maximizes the number of generated regions must maximize the number of "internal nodes" that can generate such regions in the binary tree depicting the splitting process. In particular, the scenario should create as many internal node regions with false invalidation count between t and $2t$ as possible, and then employ **Case 2** to maximize the number of final regions generated by each "internal node" region. Note that for $2t < f < k \cdot t$, the region may be split into at most $k - 1$ regions where each region has false invalidation count between t and $2t$ (regardless of how many epochs it takes). With the worst-case number of regions generated by each such internal node region given in **Case 2**, the upper bound on the number of generated regions is given by (Figure 3.3 (b)):

$$S = (k-1) \cdot (1 + \log_2 M) = (\lceil \frac{f_i}{t} \rceil - 1) \cdot (1 + \log_2 M)$$

As such, across the three cases, the total number of regions is at most $(\lceil \frac{f}{t} \rceil - 1) \cdot (1 + \log_2 M)$. □

Bounding the total number of regions. We now consider the worst-case number of regions S_{max} contributed by *all* M-sized regions. Let f_i be the number of false invalidation count for an M-sized region i ($1 \leq i \leq N$), and S_i be the worst-case number of regions generated by it, then:

$$S_{max} = \sum_{i=1}^{N} S_i = \sum_{i=1}^{N} (\lceil \frac{f_i}{t} \rceil - 1) \cdot (1 + \log_2 M)$$

To bound S_{max}, we must set t appropriately. In order to ensure fairness across all M-sized regions,

we set the threshold t as a fraction of the average false invalidation across them, i.e.,

$$t = \frac{1}{c \cdot N} \cdot \sum_{i=1}^{N} f_i \tag{3.1}$$

where c is a constant parameter.

This allows us to bound S_{max} as follows:

$$\begin{aligned} S_{max} &= \sum_{i=1}^{N} (\lceil \frac{f_i}{t} \rceil - 1) \cdot (1 + \log_2 M) \leq \sum_{i=1}^{N} \frac{f_i}{t} \cdot (1 + \log_2 M) \\ &= c \cdot N \cdot (1 + \log_2 M) \qquad \text{(From Eq. 3.1)} \end{aligned}$$

If use up all the available switch data plane capacity to store S_{max} entries, we can set c as $\frac{S_{max}}{N \cdot (1+\log_2 M)}$, which will always ensure the total number of regions is $\leq S_{max}$.

Split vs. merge-based approach. The approach we have described so far starts with M-sized regions, and splits into regions until the false invalidation count for each region reduces below the threshold t. An alternate but equivalent strategy would begin with 4KB regions and merge them into larger regions as long as the false invalidation count per region remains below t. In fact, it is possible to begin with any intermediate region size, and split or merge as necessary. In MIND, we use a default of 16KB since it provides a favorable tradeoff between storage and performance overheads for our evaluated workloads — we defer a detailed analysis to §3.14.

From theory to practice. At $c = 1$, the dynamic resizing approach outlined above reduces the amount of storage required for directory entries from $M \cdot N$ to a worst-case of $(1 + \log_2 M) \cdot N$ — an *exponential* decrease. At the same time, it ensures that the number of false invalidation count remains under $\frac{\sum f_i}{N}$. However, we note that our theoretical analysis only reveals the *worst-case* — in practice, we find both storage and performance overheads are much lower, as we show in §3.14. As such, we can set the value $c > 1$ to increase switch data plane storage utilization without hitting its capacity in practice. In fact, we dynamically adjust the value of c such that the utilization of the switch data plane storage in any epoch remains below 95%.

3.6 MIND Implementation

We now describe MIND implementation. MIND exposes Linux memory and process management system call APIs, and splits its kernel components across CPU blade and the programmable switch. We now describe these kernel components, along with the RDMA logic required at the memory blade.

3.6.1 CPU Blade

MIND assumes a partial disaggregation model, where the CPU blades possess a small amount of local DRAM as cache (§**??**). The CPU blades in our prototype use traditional servers with no hardware modifications. We implemented CPU blade kernel components as a modified Linux kernel 4.15. MIND provides transparent access to the disaggregated memory, by modifying how `vma`s and processes are managed and how page faults are handled at the CPU blade, as we detail next.

Managing `vma`s. To handle the creation and removal of `vma`s due to process heap allocation/deallocation requests, such as `brk`, `mmap`, and `munmap`, the kernel module intercepts such requests from the process and forwards them to the control plane at the switch over a reliable TCP connection. The switch subsequently creates new `vma` entries, and responds with the same return value (*e.g.*, virtual address of the allocated `vma`) as the local version of the system calls — ensuring transparency for user applications. The switch returns Linux-compatible error codes (*e.g.*, `ENOMEM`) if there are any errors.

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 the internal representation of processes (i.e., Linux's `task_struct`) and a mapping between the compute blades and processes they host. MIND assigns threads running on different CPU blades the same PID if they belong to the same process, permitting them to transparently share the same address-space via memory protection and address translation rules installed at the switch. Finally, we do not focus on scheduling in this work and simply place threads and processes across compute blades in a round-robin manner.

Page fault-driven access to remote memory. When a user application tries to access a memory address not present in the CPU blade cache, a page fault handler is triggered and the CPU blade

kernel sends a one-sided RDMA read request to the switch with the virtual address and the requested permission class, i.e., read or write for Linux. At the same time, the page to be used by the user application is registered to the NIC as the receiving buffer, obviating the need for additional data copies. Once the page is received, the local memory structures such as PTEs are populated and the control is returned to the user. Our implementation of the CPU blade DRAM cache is similar to LegoOS [2], but additionally handles cache invalidations for coherence. Specifically, the cache tracks the set of writable pages locally, and on receiving an invalidation request for a region, it flushes all writable pages in the region and removes all local PTEs.

While the above approach enables transparency for access to disaggregated memory, it presents a significant limitation in our implementation — it restricts the memory consistency model in MIND to stronger Total Store Order (TSO), and precludes weaker consistency models, *e.g.*, Process Store Order (PSO) used in DSM approaches [107]. This is because unlike TSO, PSO enables multiple writes to a cached memory region to be propagated asynchronously, but blocks if there is a subsequent read to the same region. Realizing this relaxation using page faults requires such writes to be buffered at the compute blade's local DRAM cache without triggering a page fault, but triggering one on a subsequent read to the same page. Unfortunately, this is impossible in traditional x86 or ARM architectures, since they do not support triggering a trap on read without also triggering one for a write. Consequently, MIND's stricter TSO model results in limited scalability for workloads with high read/write contention to shared memory regions, as we show in §3.7.1. We discuss possible architectural changes to address this in §3.8.

3.6.2 Memory blade

Unlike prior disaggregated memory systems [2, 32] or distributed shared memory systems [107], MIND does not require any compute logic or data plane processing logic to run on the memory blades, obviating the need for general purpose CPUs on them. However, since the memory blade in our prototype is realized on traditional Linux servers, we rely on the kernel module at the memory blade to perform RDMA-specific initializations. When a memory blade comes online, its kernel module registers physical memory addresses to the RDMA NIC and reports the mapped address to the global controller. However, subsequent one-sided RDMA requests from CPU blades are handled completely by the memory blade NIC without involving the CPUs. Ideally, memory blades would

be realized with all logic, including initialization, completely in hardware, without a CPU. While this would facilitate a memory blade design that is both simple and cheap, it would require new hardware design.

3.6.3 Programmable Switch

The MIND programmable switch module is implemented on a 32-port EdgeCore Wedge switch with a 6.4 Tbps Tofino ASIC and an Intel Broadwell processor, 8 GB RAM and 128 GB SSD. The general purpose CPU hosts the MIND control program, which performs process, memory and cache directory management. Meanwhile, the ASIC performs address translation (§3.4.1) and memory protection (§3.4.2), handles directory state transitions and virtualizes RDMA connections between compute and memory blades. We here provide implementation details of the mechanisms not already described in §**??**.

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/thread management (`task_struct`) as well as memory management (`mm_struct`, `vm_area_struct`). On receiving a system call, the control plane modifies the data structures accordingly, and responds with return values consistent with the system calls to ensure transparency.

Cache directory management. MIND reserves a fixed amount of SRAM at the data plane for storing directory entries, and partitions it into fixed sized slots, one for each *region* entry in MIND. The control plane maintains a *free list* for available slots, and a hash table *used map* which maps the base virtual address for the dynamically sized cache region to the SRAM slot storing its directory entry. All slots are initially added to the free list. MIND creates a directory entry for a region during its allocation by removing an SRAM slot from the free list, populating it with the directory entry with invalid (**I**) state, creating a match-action rule that maps the block virtual address to the SRAM slot at the data plane, and updating its *used map*. A similar process occurs when a region is split, while removing a directory entry follows the reverse procedure.

Directory state transitions. We found that a single match-action unit (MAU) in today's switch ASICs is unable to (i) perform a directory entry lookup, (ii) determine the correct translation based on the current block state and memory access request, and (iii) update the directory entry accord-

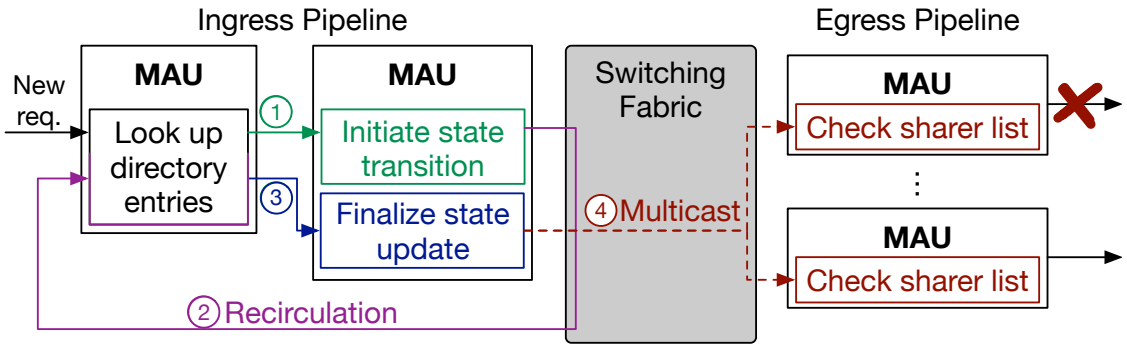


Fig. 3.4: **Performing directory state transitions on switch ASIC.**

ingly all at once, due to their limited compute capability. As such, we split the logic for (i-ii) across two MAUs (① in Figure 3.4): the first MAU stores the directory entries and performs (i), while the second MAU stores a *materialized* state-transition table containing all possible transitions and corresponding actions to be performed for (ii). Note that explicitly storing the state-transition table trades-off data plane memory capacity to overcome the limited compute cycles in an MAU. To perform (iii), the second MAU *recirculates* the memory access request packet within the switch data plane to send it back to the first MAU (②), so that it can update the directory entry according to actions determined by the second MAU (③). If the state transition requires cache invalidations, the data plane creates invalidation requests leveraging *multicast*, as described in §3.4.3. Specifically, these requests are only forwarded to the current sharers for the relevant page (④).

Virtualizing RDMA connections. When a compute blade in MIND issues an RDMA request, it does not know the location of the blade where the page resides. Consequently, the switch data plane in MIND *virtualizes* RDMA connections between all possible CPU-memory blade pairs by transparently transforming and redirecting corresponding RDMA requests and responses between them. Specifically, once an RDMA request's destination blade is identified via address translation or cache coherence, the data plane updates the request's packet header fields such as IP/MAC addresses and RDMA-specific parameters, before forwarding it to the blade.

3.7 Evaluation

We evaluate MIND to answer the following questions:

- Can MIND enable transparent elasticity over performant disaggregated memory for real-world workloads (§3.7.1)?

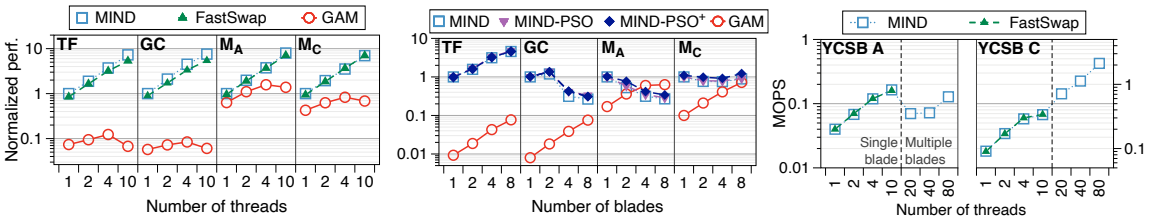


Fig. 3.5: **Performance scaling** (left) on a single compute blade, (center) across compute blades, and (right) for Native-KVS. For (center), each blade runs 10 threads. Performance is normalized by MIND's performance at 1 thread for (left) and 1 blade for (center); the runtimes in seconds for TF, GC, M_A and M_C workloads are 62.8, 59.3, 301.4 and 268.2 for 1 thread, and 69.2, 62.8, 302.3 and 306.9 for 1 blade, respectively.

- What are MIND's performance/resource bottlenecks (§3.7.2)?
- How does bounded splitting perform in isolation (§C.3)?

Compared systems. We compare MIND against two extreme designs in disaggregated memory (§**??**): a *transparent* DSM-based approach with cache-coherence that supports compute elasticity, and a *non-transparent* approach that limits compute elasticity to a single compute blade. For the former, we adapt GAM [107], a software-based DSM, to our disaggregated setting where cache directory is implemented at the compute blades. For the latter, we employ FastSwap [31], a state-of-the-art swap-based disaggregated memory system. All systems employ RDMA for efficient remote memory accesses. Finally, MIND uses an initial region size of 16 kB and epoch size of 100 ms for its bounded splitting algorithm.

Cluster setup. We used a cluster comprising five servers connected via the programmable switch described in §3.6.3. We used a single server equipped with two 18-core Intel Xeon processors, 384 GB of memory and four Nvidia/Mellanox CX-5 100 Gbps NICs, to host multiple memory blade VMs. To highlight the overhead and scalability of in-network cache coherence protocol, we used the remaining four machines, each equipped two 12-core Intel Xeon processors and two Nvidia/Mellanox CX-5 100 Gbps NICs, to host two CPU blade VMs on each server. Similar to prior work [2], we emulate the partial disaggregation model by limiting the local DRAM usage at each compute blade to 512 MB, which is about 25% of the memory footprint for our evaluated workloads. Note that each CPU and memory blade VM in our setup had dedicated access to a separate 100 Gbps NIC to ensure they represent separate network attached entities.

Applications and workloads. We use several real-world workloads in our evaluation: Tensor-

Flow [132] with ResNet-50 [133] on CIFAR-10 [134] (denoted as TF), GraphChi [135] with PageRank [136] on Twitter graph [137] (denoted as GC), and Memcached [138] with YCSB [139] workload A (50% read, 50% write split, denoted as M_A) and workload C (100% reads, denoted as M_C). Since GAM is a *software* DSM system, it requires applications to use a specialized memory API, while MIND and FastSwap are transparent to applications. To ensure consistent comparison under different interfaces, we captured the memory accesses from our workloads using Intel's PIN [140], and used a memory access emulator to generate the exactly same memory accesses across all three systems. In addition, we also present results for native execution of a simple key-value store (denoted as Native-KVS) on MIND and FastSwap, since they support a transparent memory interface.

3.7.1 Performance Scaling for Real-World Workloads

We start by evaluating MIND's performance scalability.

Intra-blade scaling. Figure 3.5 (left) shows performance scaling across all systems as the number of threads is increased on a single compute blade. We report performance as the inverse of runtime, normalized by the performance of MIND for 1 thread. MIND and FastSwap scale almost linearly with more compute blades because of their efficient page-fault driven remote memory accesses. In contrast, GAM scales linearly only up to 4 threads, and sub-linearly after that due to software overheads from its user-level library. For instance, GAM must check access permissions for every memory access by acquiring a lock, while MIND and FastSwap can leverage the hardware MMU to facilitate the same. Such overheads become significant as the compute resources on a single compute blade come under contention at 10 threads running on a 12-core node.

Inter-blade scaling. Next, we evaluate inter-blade scalability by running 10 execution threads per-blade, for up to 8 compute blades. Figure 3.5 (center) shows our results; here, while MIND's default memory consistency model is strict (TSO, §**??**), MIND-PSO denotes the *simulated* performance of MIND with the weaker PSO model (same as GAM). MIND-PSO+ additionally simulates the effect of infinite switch capacity for directory storage. Since we are forced to simulate MIND-PSO and MIND-PSO+ using traces collected on a real TSO-based system, the traces retain additional TSO-associated queuing delays that we cannot elide, i.e., while our simulations can reorder writes and non-conflicting reads, queuing delays remain; in other words, our MIND-PSO and MIND-PSO+ re-

sults are underestimates to potential performance of a hardware-based solution. Finally, we omit FastSwap as it does not transparently scale beyond a single compute blade, similar to other disaggregation proposals [2, 32].

For a machine learning workload (TF), MIND's performance scales well despite its stricter memory consistency model compared to GAM — doubling the number of compute blades improves MIND's performance by $\sim 1.67\times$, with a $59\times$ speeded compared to GAM at 8 compute blades. For GC, MIND's performance increases from 1 to 2 compute blades, but starts to decrease beyond that. This is because GC's graph traversals incur random and often contentious access to shared data compared to machine learning workloads in TF. GC writes $\sim 2.5\times$ more data in shared pages than TF, generating significantly more state transitions to modified (**M**) state, and incurring frequent invalidations (§3.7.2). PSO partly alleviates this overhead by permitting writes to be performed asynchronously, but still does not permit linear scaling beyond 2 compute blades. Instead, GAM scales better because the performance differential between its local and remote accesses is small — local accesses are $10\times$ slower than that of MIND (due to software implementation of local accesses), while remote access latencies are similar for both. Consequently, performing more remote accesses (during invalidations) does not impact GAM performance as much as it does for MIND.

Finally, M_A and M_C have more sharers with much larger number of shared writes compared to TF and GC. As a result, MIND does not scale well beyond 1 compute blade because: (1) more blades contend for acquiring write permission to the same region incurring multiple invalidations and significantly smaller number of local memory accesses, and (2) the directory storage at the switch becomes a bottleneck (as we show in §3.7.2), frequently resulting in false invalidations for heavily shared memory regions. We confirm these insights through MIND-PSO and MIND-PSO+ simulated results, which show that employing weaker memory consistency models and infinite directory capacity improves MIND's performance to some extent. Note that for M_C, MIND's performance increases from 4 to 8 blades since the number of invalidations do not increase by much. GAM scales better due to its weaker consistency model, and by leveraging its software implementation to facilitate several memory access reorderings which are not possible in MIND. Consequently, at 8 compute blades, GAM and MIND-PSO+ achieve roughly similar performance.

Native KVS. Figure 3.5 (right) shows the intra- and inter-blade scaling of Native-KVS on MIND

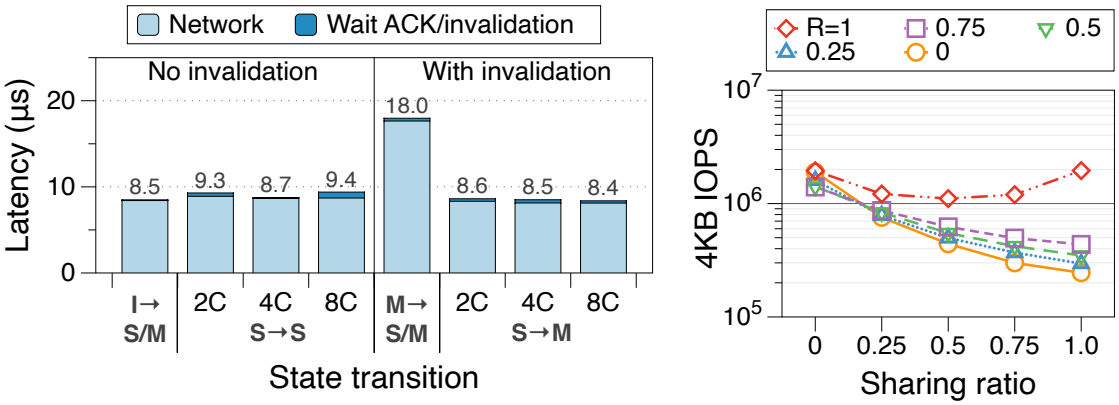


Fig. 3.6: **Performance bottlenecks.** (left) Network latency for state transitions, (right) memory throughput vs. read-write/sharing ratios.

and FastSwap for YCSB-A and C workloads. On a single blade, both MIND and FastSwap observe near linear performance scaling for up to 10 threads. Since FastSwap does not support sharing state across multiple compute blades, we do not report its performance beyond 10 threads. Similar to our results for M_A, MIND does not scale well beyond a single compute blade for the YCSB-A workload (50% reads, 50% writes) due to high read-write contentions. For the YCSB-C workload, Native-KVS scales linearly even beyond a single blade since it is a read-only workload, incurring no invalidations. Interestingly, YCSB-A workload on Native-KVS scales better than M_A — we attribute this to better partitioning of KVS state across compute blades in Native-KVS compared to Memcached.

3.7.2 MIND's Performance and Resource Bottlenecks

We study MIND bottlenecks in terms of (i) memory access performance, and (ii) memory resources at the switch.

Fig. 3.7: **MIND switch resource bottlenecks.** (left) Directory entries, (center) match-action entries for heap, (right) load balancing for heap.

Latency for cache state transitions. Figure 3.6 shows the end-to-end latency due to every possible state transition under the MSI protocol in MIND, including the time required to fetch the data. Note

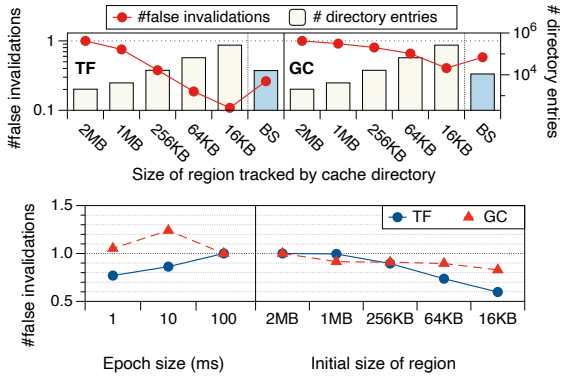


Fig. 3.8: **Evaluating MIND's bounded splitting algorithm.** (left) Navigating switch storage vs. performance tradeoff. (right) Impact of epoch & initial region sizing. The number of false invalidations is normalized by value at 2 MB for region size and 100 ms for epoch size.

that this figure only shows latency for remote accesses — local accesses only incur DRAM latency ($<$ 100 ns). On the x-axis, 2 – 8C indicate the number of CPU blades requesting the same page, and $x \rightarrow y$ denotes the state transition, x and y being the initial and final states.

When a blade requests read-only (shared, **S**) mode for a region, and its initial state was either invalid (**I**) or shared (**S**), it does not require any invalidations. Consequently, the data fetch can be performed in a single RDMA request ($\sim$9 μs), as seen in the first four bars. If the transition for a region is either from or to the modified (**M**) state, the requesting blade must wait until the regions is invalidated at all its previous owners. When transitioning from **S** to **M**, the data can be fetched directly from the memory blade via one-sided RDMA operation, while the invalidation at other blades occur in parallel, resulting in a total latency of $\sim$9 μs. When the region is initially in **M** state, the (dirty) data must be fetched from and the region invalidated at the same blade — its current owner. Therefore, the invalidation and data fetch occur sequentially, resulting in $\sim$18 μs latency. Note that since the latency for requests with invalidations is 2$\times$ higher than requests without them, a workload's performance depends on the relative proportion of the different types of requests, as we show next.

Impact of invalidations on memory throughput. Figure 3.6 (right) shows MIND memory throughput across 8 compute blades, running 1 compute thread each, under various read-write and sharing characteristics. We use read ratio to denote the fraction of reads in the workload (remaining accesses are writes), and sharing ratio to denote the portion of memory accesses that occur to a shared region (shared by all threads). We used a total working set size of 400 k pages, with the access pattern

across them being uniform random. If most accesses are reads, then compute blades can share the same region without triggering invalidations (**S**→**S** in Figure 3.6). As such, at read-ratio 1, most of the pages are accessed locally from the cache, resulting in very high memory throughput (1-2 $\times$ 10^6 IOPS) for all sharing ratios. Again, at sharing ratio 0, memory throughput remains high, since accessed pages can remain cached at the compute blade without being invalidated, i.e., most accesses are local. If both the write proportion and sharing ratio are increased, memory throughput drops (by $\sim$10$\times$ at sharing-ratio 1), since they trigger a large number number of **M**→**S**, **S**→**M** transitions with invalidations and permit few pages to be accessed locally.

Cache directory storage. Figure 3.7 (left) shows the number of cache directory entries stored in the switch data plane in MIND over time for the workloads evaluated in §3.7.1 across 8 compute blades, running 10 threads each. In MIND, we fix the total amount of storage allocated to directory storage to 30 k entries. For the TF and GC workloads, MIND's bounded splitting algorithm ensures that the number of directory entries remains well below the limit over time. However, the MC$_A$ and MC$_B$ workloads have a significantly larger number of shared memory regions, with frequent read and write accesses to them; as a consequence, the number of directory entries for the workloads always remains close to the 30 k limit. Recall from §3.7.1 that one of the key reasons for poor scalability of these workloads is the number of false invalidations triggered due to the relatively coarse granularity of tracking directory entries — we believe with future switch ASICs likely to be equipped with more TCAM/SRAM, this bottleneck would no longer exist, permitting more efficient scaling under MIND.

Address translation & memory protection storage. We study the switch storage overheads due to address translation and memory protection on a setup with 8 memory blades, running the TF, GC, and M$_{A/C}$ workloads; we group M_A and M_C since they have the same memory allocations. Figure 3.7 (center) shows that the number of match action rules due to address translation and memory protection in MIND is almost constant, even as the workload size increases. This is due to MIND's per-memory blade partitioning of the address space, and `vma` granularity tracking of memory protection entries. While we have only shown results for three different applications, we find that the number of `vma` entries for typical datacenter applications falls in similar ranges, and well under 1–2 k [**?**, **?**]. In contrast, the number of match-action rules increases linearly with the dataset size

for page-based approaches, despite smaller absolute overheads with 1 GB huge pages. Note that the upper-limit for match-action rules that the switch can store is about 45 k — higher than the 30 k limit for directory entries due to a more compact representation.

MIND's memory allocation also ensures balanced placement of load across memory blades (§3.4.1), as shown via Jain's fairness index metric [141] in Figure 3.7 (right). While 2 MB pages can achieve similar load-balancing, they do so at the cost of much larger number of address translation entries. 1 GB pages, on the other hand, observes poor load balancing for allocation-intensive workloads ($M_{A/C}$).

3.7.3 Evaluating MIND's Bounded Splitting Algorithm

We now evaluate MIND's bounded splitting algorithm.

Storage vs. performance tradeoff. Recall from §3.4.3 that the granularity at which the directory tracks memory regions exposes a tradeoff between the false invalidation count and the size of the directory itself — Figure 3.8 (left) highlights this tradeoff for the TF and GC workloads. Specifically, tracking smaller regions (*e.g.*, 16 kB) permits fewer false invalidations, but at the cost of larger number of directory entries at the switch, while tracking larger regions (*e.g.*, 2 MB) exposes the opposite tradeoff. MIND's bounded splitting algorithm employs adaptive region sizing to balance both the number of directory entries as well false invalidations.

Impact of epoch and initial region sizing. Figure 3.8 (right) shows the impact of epoch size on the total number of false invalidations for TF and GC workloads. Increasing the epoch size from 1 to 100 ms does not have a significant impact on the number of false invalidations, but reduces the control plane overheads. Epoch size smaller than 1ms (not shown) are unable to capture enough invalidations to enable accurate estimation of the distribution, resulting spurious merges/splits and unpredictable false invalidations. We use 100 ms as our default epoch size since it offers a sweet spot for minimizing both false invalidations and control plane overheads.

Figure 3.8 (right) shows that picking smaller initial region sizes results in fewer false invalidations — intuitively, this is because larger initial region sizes require several splits before stabilizing to the appropriate region size, incurring several false invalidations in the interim. We select 16KB as our default initial region size, since smaller region sizes result in too many directory entries during

initialization.

Finally, we note that neither parameter has any noticeable impact on the number of directory entries at stable state.

3.8 Limitations and Future Research

We now discuss the limitations of current MIND implementation, and future research directions to resolve them.

Thread management. Even with our optimizations, remote memory access latency is still at least two orders of magnitude higher than local latency. While our work explores in-network approaches to minimize overheads of coherence, an orthogonal approach of co-locating threads with higher proportion of shared memory accesses could yield significant improvements in end-to-end application performance by reducing the number of invalidations over the network.

Other coherence protocols. While MIND implements the simple MSI coherence protocol, more complex protocols like MOESI may offer better scalability by reducing broadcasts and write-backs to disaggregated memory. Realizing such protocols would require storing larger state transition tables (STT) at the switch and handling more transient states, adding implementation complexity for ensuring correctness. Still, the number of TCAM entries required for STT entries would be quite small (*e.g.*, tens of states for MOESI) relative to switch ASIC capacities, making them realizable today.

Weaker consistency models. As we noted in §**??** and §3.14, our page-fault based implementation on x86 architectures cannot realize weaker consistency models like PSO. To this end, a redesign of the compute blade architecture — *e.g.*, by enabling page-faults on reads (but not writes) to a page — could enable realization of weaker consistency models in MIND, facilitating higher throughput to disaggregated memory.

Scaling beyond a rack. While MIND targets a rack-scale design with a single switch, some workloads may want to scale transparently beyond a single rack. This requires a shift similar to the shift from single node CPUs (akin to the rack in our setting) to multi-node NUMA architectures (akin to datacenter-scale memory disaggregation). Such a design would require extension of MIND design

from a single switch to a datacenter-wide network topology.

Virtualization. While MIND enables protection at a virtual memory level, extensions to virtualization are needed to facilitate true isolation across users for security, resource management, legacy OS support, etc. Providing performance isolation, in particular, would require isolating several different shared resources along the compute-memory interconnect, including network bandwidth, switch and NIC resources.

3.9 Related Work

While we discussed prior disaggregated memory approaches in §**??**, we now discuss other work related to MIND.

In-network computing. There have been several recent efforts that leverage in-network computing for performance gains [142–159]. Most focus was on offloading *application* logic and state to the network, *e.g.*, key-value caches [160, 161] and metadata [162]. Perhaps the most relevant to MIND are NOPaxos [150] and Concordia [163]. NOPaxos leverages the network to order requests for Paxos-based consensus, enabling consistent replication without expensive coordination overheads. While MIND targets a complementary goal of in-network memory management, it could leverage NOPaxos to enable consistent replication of disaggregated memory. Concordia, on the other hand, uses the programmable switch as a cache for directory entries in a DSM; in contrast, MIND realizes memory management completely in the network.

Application-driven memory disaggregation. Recent work argues for OSes to expose resource management abstractions like memory placement and failures to the applications for high performance and better fault-tolerance [**?**, 91]. While MIND argues for a transparent disaggregated shared memory, it is not incompatible with the above approaches — OS-level libraries layered atop MIND could still expose memory placement and failure notifications to applications.

Clover [164] explores the design of a key-value store closely integrated with disaggregated persistent memory. In contrast to MIND's in-network transparent memory management, Clover focuses on lock-free consistent access to KV pairs stored in network-attached memory using atomic RDMA verbs. While a possible design considered in [164] places key-value coordination and access

logic at a centralized coordinator, it does not place the logic in the network fabric and does not consider memory protection, caching or coherence.

Emerging industry standards. While most industry standards for high performance compute-memory interconnects like CCIX [**?**], CXL [10] and OpenCAPI [**?**] target *intra*-server settings, Gen-Z [165] is perhaps the closest to MIND since it targets *inter*-server fabrics. The Gen-Z standard defines operations like `ExclusiveRead` and `Writeback` that may be used as building blocks for software-based coherence [**?**, 165], although we are unaware of any publicly available realization. Moreover, the MMU functionalities in all the above industry standards are realized at the *endpoints*, *e.g.*, at specialized ZMMUs at CPU and memory nodes in Gen-Z; the *fabric* (*e.g.*, the switch) only forwards memory requests and responses. This is in contrast to MIND's approach of in-network memory management, i.e., MIND's design is complementary to the industry efforts towards high-performance interconnects.

3.10 Conclusion

We have presented MIND, an in-network memory management unit for rack-scale memory disaggregation. MIND achieves resource elasticity, performance and transparency through a principled redesign of traditional memory management mechanisms to achieve their individual goals in the disaggregated setting while operating under programmable switch ASIC resource constraints. Our MIND prototype facilitates transparent resource elasticity, while matching the performance of prior memory disaggregation proposals for real-world workloads.

3.11 Near Memory Processing

3.11.1 Introduction

Driven by increasing demands for memory capacity and bandwidth [81,166–171], poor scaling [172–174] and resource inefficiency [32, 175] of DRAM, and improvements in Ethernet-based network speeds [89, 90], recent years have seen significant efforts towards memory disaggregation [1, 2, 4, 31, 32]. Rather than scaling up a server's DRAM capacity and bandwidth, such proposals advocate disaggregating much of the memory over the network. The result is a set of CPU nodes equipped

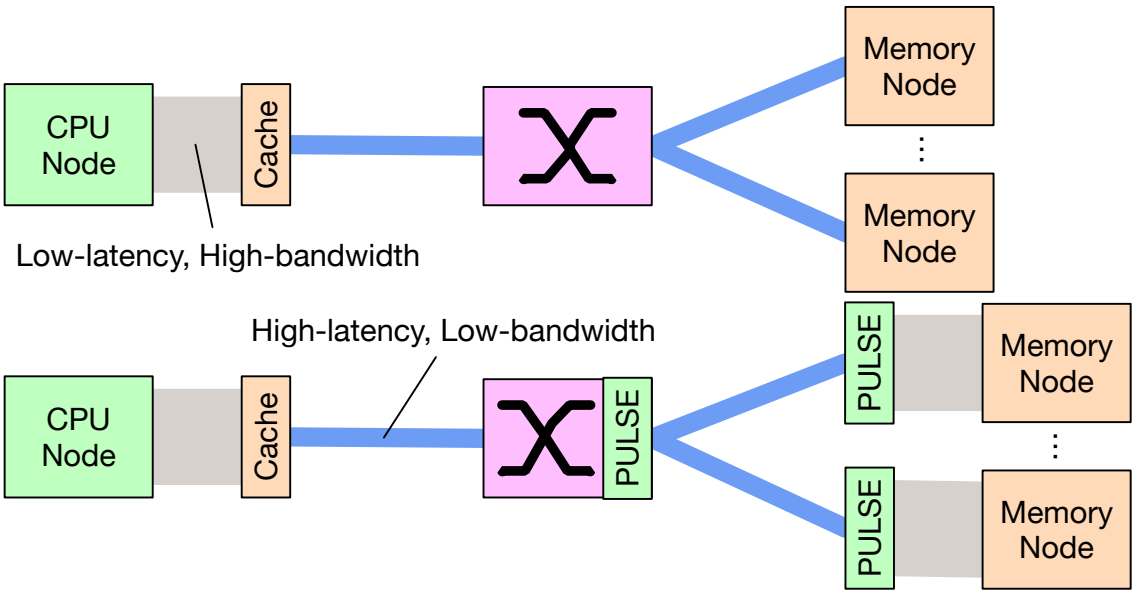
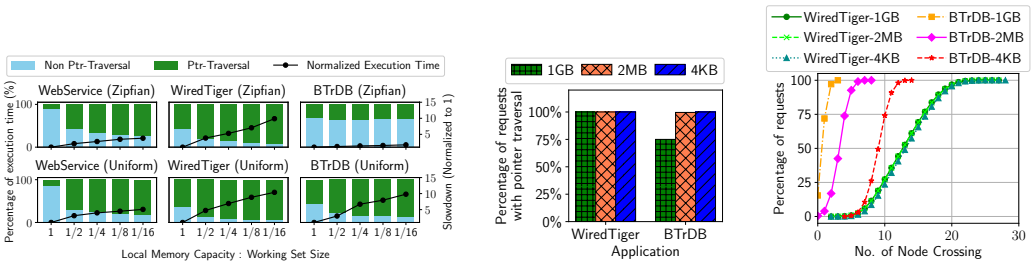


Fig. 3.9: **Need for accelerating pointer traversals.** *(top)* The performance of pointer traversals in disaggregated architectures is bottlenecked by slow memory interconnect. *(bottom)* Just as caches offer limited but fast caches near CPUs, we argue that memory needs a counterpart for traversal-heavy workloads: a lightweight but fast accelerator for cache-unfriendly pointer traversals.

with a small amount of DRAM used as cache[3], accessing memory across a set of network-attached memory nodes with large DRAM pools (Fig. 3.9 (top)). With allocation flexibility across CPU and memory nodes, disaggregation enables high utilization and elasticity.

Despite drastic improvements in recent years, the limited bandwidth and latency to network-attached memory remain a hurdle in adopting disaggregated memory, with speed-of-light constraints making it impossible to improve network latency beyond a point. Even with near-terabit links and hardware-assisted protocols like RDMA [179], remote memory accesses are an order of magnitude slower than local memory accesses [3]. Emerging CXL interconnects [10] share a similar trend — around 300 ns of CXL memory latency compared to 10–20 ns of L3 cache latency [37]. Although efficient caching strategies at the CPU node can reduce average memory access latency and volume of network traffic to remote memory, the benefit of such strategies is limited by data locality and the size of the cache on the CPU node. In many cases, remote memory accesses are unavoidable, especially for applications that rely on efficient in-memory pointer traversals on linked data structures, such as lookups on index structures [180–190] in databases and key-value stores, and traversals in graph analytics [135, 136, 191, 192] (Fig. 3.10, §3.11.2).

3. Not to be confused with die-stacked hardware DRAM caches [176–178].



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

Fig. 3.10: **Time cloud applications spend in pointer traversals.** See §3.11.2 for details.

Similar to how CPUs have small but fast memory (i.e., caches) for quick access to popular data, we argue that memory nodes should also include lightweight but fast processing units with high-bandwidth, low-latency access to memory to speed up pointer-traversals (Fig. 3.9 (bottom)). Moreover, the interconnect should facilitate efficient and scalable distributed traversals for deployments with multiple memory nodes that cater to large-scale linked data structures. Prior works have explored systems and API designs for such processing units under multiple settings, ranging from near-memory processing and processing-in-memory approaches [193–221] for single-server architectures, to the use of CPUs [28,222–226] or FPGAs [227,228] near remote/disaggregated memory, but have several key shortcomings.

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

This limits memory utilization and elasticity since applications must confine their data to a single memory node to accelerate pointer traversals. Their inability to support distributed pointer traversals stems from complex management of address translation state that is required to identify if a traversal can occur locally or must be re-routed to a different memory node (§3.11.2). Second, existing single-node approaches use full-fledged CPUs for expressive and performant execution of pointer-traversals [28,222–224]. However, coupling large amounts of processing capacity with memory — which has utility in reducing data movement in PIM architectures [193–205] — goes against the very spirit of memory disaggregation since it leads to poor utilization of compute resources and, consequently, poor energy efficiency.

Approaches that use wimpy processors at SmartNICs [229, 230] instead of CPUs retain expressiveness, but the limited processing speeds of wimpy nodes curtail their performance and, ultimately lead to lower energy efficiency due to their lengthened executions (§3.14.1, [227]). Lastly, FPGA-based [227, 228, 231] and ASIC-based [220, 221] approaches achieve performance and energy efficiency by hard-wiring pointer traversal logic for specific data structures, limiting their expressiveness.

We design PULSE[4], a distributed pointer-traversal framework for rack-scale disaggregated memory, to meet all of the above needs — namely, expressiveness, energy efficiency, performance — via a principled redesign of near-memory processing for disaggregated memory. Central to PULSE's design is an expressive iterator interface that readily lends itself to a unifying abstraction across most pointer traversals in linked data structures used in key-value stores [24, 138], databases [185–187, 189, 232], and big-data analytics [135, 136, 191, 192] (§3.12). PULSE's use of this abstraction not only makes it immediately useful in this large family of real-world traversal-heavy use cases, but also enables (i) the use of familiar compiler toolchains to support these use cases with little to no application modifications and (ii) the design of tractable hardware accelerators and efficient distributed traversal mechanisms that exploit properties unique to iterator abstractions.

In particular, PULSE enables transparent and efficient execution of pointer traversals for our iterator abstraction via a novel accelerator that employs a *disaggregated* architecture to decouple logic and memory pipelines, exploiting the inherently sequential nature of compute and memory accesses in iterator execution (§3.12.1). This permits high utilization by provisioning more memory and fewer logic pipelines to cater to memory-centric pointer traversal workloads. A scheduler breaks pointer traversal logic from multiple concurrent workloads across the two sets of pipelines and employs a novel multiplexing strategy to maximize their utilization. While our implementation leverages an FPGA-based SmartNIC due to the high cost and complexity of ASIC fabrication, our ultimate vision is an ASIC-based realization for improved performance and energy efficiency.

We enable distributed traversals by leveraging the insight that pointer traversal across network-attached memory nodes is equivalent to packet routing at the network switch (§3.13). As such, PULSE leverages a programmable network switch to inspect the next pointer to be traversed within

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

iterator requests and determine the next memory node to which the request should be forwarded — both at line rate. We implement a real-system prototype of PULSE on a disaggregated rack of commodity servers, SmartNICs, and a programmable switch with full-system effects. None of PULSE's hardware or software changes are invasive or overly complex, ensuring deployability. Our evaluation of end-to-end real-world workloads shows that PULSE outperforms disaggregated caching systems with 9–34× lower latency and 28–171× higher throughput. Moreover, our Xilinx XRT [233] and Intel RAPL [234]-based power analysis shows that PULSE consumes 4.5–5× less energy than RPC-based schemes (§3.14).

3.11.2 Motivation and PULSE Overview

Need for Accelerating Pointer Traversals

Memory-intensive applications [81, 166–171] 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 is still slow [3]. Recent proposals [1–3, 31, 32] alleviate this slowdown by using the DRAM at the CPU nodes to cache "hot" data, but such caches often fare poorly for pointer traversals, as we show next.

Pointer traversals in real-world workloads. Prior studies [81, 135, 138, 235–238] 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 [28], indexing on WiredTiger [232], and time-series analysis on BTrDB [239] — with swap-based disaggregated memory [32][5]. We vary the cache size at the CPU node from 6.25%-100% of each application's working set size. Fig. 3.10(a) shows that (i) all three applications spend a significant fraction of their execution time (13.6%, 63.7%, and 55.8%, respectively) traversing pointers even when their entire working set is cached, and (ii) the time spent traversing pointers (and thus, the end-to-end execution time) increases with smaller CPU node caches. While the impact of access skew is application-dependent, pointer traversals dominate application execution times when more of the application's working set size is remote.

5. We defer the details of the data structures and workloads employed by these applications, as well as the disaggregated memory setup to §3.14.

Distributed traversals. As the number of applications and the working-set size per application grows larger, disaggregated architectures must allocate memory across multiple memory nodes to keep up. Such approaches [1, 2, 31, 32] tend to strive for the smallest viable allocation granularity with reasonable metadata overheads (e.g., 1 GB in [2], 2 MB in [1]) since smaller allocations permit better load balancing and high memory utilization. Unfortunately, finer-grained allocations may cause an application's linked structures to get fragmented across multiple network-attached memory nodes, necessitating many *distributed* traversals.

Fig. 3.10(b) illustrates this impact on a setup with 1 compute and 4 memory nodes: even with large 1 GB allocations, WiredTiger and BTrDB require over 97% and 75% of their requests, respectively, to cross memory node boundaries at least once, with the volume of cross-node traffic increasing at smaller granularities. Fig. 3.10(c) shows the CDF of requests that require a certain number of memory node crossings. While the randomly ordered data in WiredTiger necessitate many cross-node traversals even for large allocations, the time-ordered data in BTrDB reduce cross-node traversals for larger allocation granularities by confining large time windows to the same memory node. However, smaller to moderate allocation granularities — required for high memory utilization — still require many cross-node traversals.

Shortcomings of Prior Approaches

No prior work achieves all four properties required for pointer traversals on disaggregated memory: distributed execution, expressiveness, energy efficiency, and performance. We focus on network-attached memory, although a similar analysis extends to in-memory processing [193–221].

No support for distributed execution. Distributed pointer traversals are required to ensure applications can efficiently access large pools of network-attached memory nodes. Unfortunately, to our knowledge, none of the prior works support efficient multi-node pointer traversals. Therefore, applications must confine their data to a single node for efficient traversals, exposing a tradeoff between application performance and scalability. Recent proposals [29, 164, 240–244] explore specialized data structures that co-design partitioning and allocation policies to reduce distributed pointer traversals atop disaggregated memory. Such approaches complement our work since they still require efficient distributed traversals when their optimizations are not applicable, *e.g.*, not many data

structures benefit from such specialized co-designs.

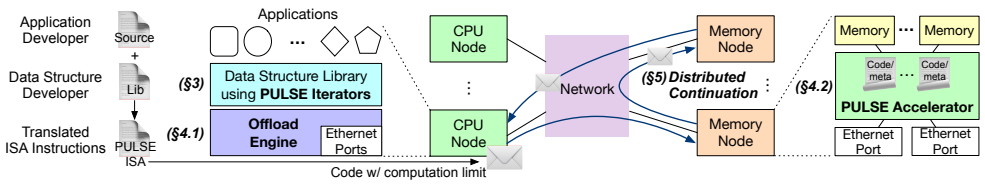


Fig. 3.11: **PULSE Overview.** Developers use PULSE's iterator interface (§3.12) to express pointer traversals, translated to PULSE ISA by its dispatch engine (§3.12.1). During execution, PULSE accelerator ensures energy efficiency (§3.12.1) and in-network design enable distributed traversals (§3.13).

Poor utilization/power-efficiency in CPUs. Many prior works have explored remote procedure call (RPC) interfaces to enable offloading computation to CPUs on memory nodes [28, 222–225]. While CPUs are performant and versatile enough to support most general-purpose computations, the same versatility makes them overkill for pointer traversal workloads in disaggregated architectures — the CPUs on memory nodes are likely to be underutilized and, consequently, waste energy (§3.14), since such workloads are memory-intensive and bounded by memory bandwidth rather than CPU cycles. Since inefficient power usage resulting from coupled compute and memory resources is the main problem disaggregation aims to resolve, leveraging CPUs at memory nodes essentially nullifies these benefits.

Limited expressiveness in FPGA/ASIC accelerators. Another approach explored in recent years uses FPGAs [227, 228] or ASICs [220, 221] at memory nodes for performance and energy efficiency. FPGA approaches exploit circuit programmability to realize performant on-path data processing, albeit only for specific data structures, limiting their expressiveness. Although some FPGA approaches aim for greater expressiveness by serving RPCs [245], RPC logic must be pre-compiled before it is deployed and physically consumes FPGA resources. This limits how many RPCs can be deployed on the FPGA concurrently and also elides runtime resource elasticity for different pointer traversal workloads. ASIC approaches either support a single data structure or provide limited ISA specialized for a single data structure (*e.g.*, linked-lists [220]), limiting their general applicability.

Poor performance/power efficiency in wimpy SmartNICs. The emergence of programmable SmartNICs has driven work on offloading computations to the onboard network processors. Some approaches utilize wimpy processors (*e.g.*, ARM or RISC-V processors) [229] or RDMA processing units (PUs) [230] to support general-purpose computations near memory. While these wimpy processors can eliminate multiple network round trips in pointer traversal workloads, their processing

speeds are far slower than CPU-based or FPGA-based accelerators. Often, such PUs can become a performance bottleneck, especially at high memory bandwidth (∼500 Gbps) [3, 230]. Moreover, wimpy processors tend not to be energy-efficient since their slower execution tends to waste more static power, resulting in higher energy per pointer traversal offload — an observation noted in prior work [227] and confirmed in our evaluation (§3.14).

PULSE Design Overview

PULSE innovates on three key design elements (Fig. 3.11). Central to PULSE's design is its iterator-based programming model (§3.12) that requires minimal effort to port real-world data structure traversals. PULSE supports *stateful* traversals using a *scratchpad* of pre-configured size, where developers can store and update arbitrary intermediate states (*e.g.*, aggregators, arrays, lists, etc.) during the iterator's execution. Properties specific to iterator patterns enable tractable accelerator design and efficient distributed traversals in PULSE.

The iterator code provided by the data structure developer is translated into PULSE's instruction set architecture (ISA) to be executed by PULSE accelerators (§3.12.1). PULSE achieves energy efficiency and performance through a novel accelerator that employs disaggregated logic and memory pipelines and an ISA specifically designed for the iterator pattern. Our accelerator employs a scheduler specialized for its disaggregated architecture to ensure high utilization *and* performance.

PULSE supports scalable distributed pointer traversals by leveraging programmable network switches to reroute any requests that must cross memory node boundaries (§3.13). PULSE employs hierarchical address translation *in the network*, where memory node-level address translation is performed at the switch (i.e., a request is routed to the memory node based on its target address), and the memory node accelerator performs translation and protection for local accesses. During traversal, a memory node accelerator can return a request to the switch if it determines the address is not local; the switch re-routes the request to the correct memory node.

Assumptions. PULSE does not offload synchronization to its accelerators but instead requires the application logic at the CPU node to explicitly acquire/release appropriate locks for the offloaded operation. Recent efforts enable locking primitives on NICs [29, 164] and programmable switches [154]; these are orthogonal to our work and can be incorporated into PULSE.d Finally,

PULSE does not innovate on caching and adapts the caching scheme from prior work [28], which maintains a transparent cache within the data structure library.

3.12 PULSE Programming Model

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

We analyzed the implementations of a wide range of popular data structures [246–249]

to determine the structures common to them in pointer traversals. We found that most traversals (1) initialize a start pointer using data structure-specific logic, (2) iteratively use data structure-specific logic to determine the next pointer to look up, and (3) check a data structure-specific termination condition at the end of each iteration to determine if the traversal should end. This structure resembles that of the *iterator* design pattern, establishing its universality as a design motif common to almost all languages [248]. This is precisely what makes it an ideal candidate for the interface between the hardware and software layers for pointer traversals. As such, PULSE allows developers to program their data structure traversals using the iterator interface shown in Listing 3.1.

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

Stateful traversals. Pointer traversals in many data structures are stateful, and the nature of the state can vary widely. For instance, in hash table lookups, the state is the search key that must be compared against a linked list of keys in a hash bucket. In contrast, summing up values across a

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

Listing 3.1: PULSE interface.

range of keys in a B-Tree requires maintaining a running variable for storing the sum and updating it for each value encountered in the range. To facilitate this, PULSE iterators maintain a `scratch_pad` that the developer can use to store an arbitrary state. The state is initialized in `init()`, updated in `next()`, and finalized in `end()`. Since `execute()` in PULSE's iterator interface returns the contents of `scratch_pad` (Line 10), developers can place the data that they want to receive in it.

Bounded computations. PULSE accelerators support only lightweight processing in memory-intensive operations for high memory bandwidth utilization. While `init()` is executed on the CPU node, `next()` and `end()` are offloaded to PULSE accelerators; hence, PULSE limits what memory accesses and computations can be performed in them in two ways. Within each iteration, PULSE disallows nondeterministic executions, such as unbounded loops, i.e., loops that cannot be unrolled to a fixed number of instructions.

Across iterations, `execute()` in Listing 3.1 limits the maximum number of iterations that a single request is allowed to perform. This ensures that a particularly long traversal does not block other requests for a long time. If a request exceeds the maximum iteration count, PULSE terminates the traversal and returns the `scratch_pad` value to the CPU node, which can issue a new request to continue the traversal from that point.

An illustrative example. We demonstrate how the `find()` operation on C++ STL `unordered_map`

can be ported to PULSE. Listing 3.2 shows a simplified version of its implementation in STL — the pointer traversal begins by computing a hash function and determining a pointer to the hash bucket corresponding to the hash. It then iterates through a linked list corresponding to the hash bucket, terminating if the key is found or the linked list ends without it being found.

Listing 3.3 shows the corresponding iterator implementation in PULSE. Much of the implementation is unchanged, with minor restructuring for `init()`, `next()`, and `end()` functions. The main changes are — how the state (the search key) is exchanged across the three functions and how the data is returned back to the user via the `scratch_pad` (an error message if the key is not found, or its value if it is).

3.12.1 Accelerating Pointer Traversals on a Node

PULSE Dispatch Engine

The dispatch engine is a software framework running at the CPU node for two purposes. First, it translates the iterator realization for pointer traversal provided by a data structure library developer (§3.12) into PULSE's ISA. Second, it determines if the accelerator can support the computations performed during the traversal, and if so, ships a request to the accelerator at the memory node. If not, the execution proceeds at the CPU node with regular remote memory accesses.

Translating iterator code to PULSE ISA. To be readily implementable, PULSE plugs into existing compiler toolchains. The dispatch engine generates PULSE ISA instructions using widely known compiler techniques [250]. PULSE's ISA is a stripped-down RISC ISA, only containing operations necessary for basic processing and memory accesses to enable a simple and energy-efficient accelerator design (Table 3.3). There are, however, a few notable aspects to our adapted ISA and the translation of iterator code to it. First, as noted in §3.12, PULSE does not support unbounded loops within a single iteration, i.e., the ISA only supports conditional jumps to points ahead in code. This is similar to eBPF programs [251], where only forward jumps are supported to prevent the program from running infinitely within the kernel. A backward jump can only occur when the next iteration starts; PULSE employs a special `NEXT_ITER` instruction to explicitly mark this point so that the accelerator can begin scheduling the memory pipeline (§3.12.1). Second, again as noted in §3.12, developers can maintain state and return values using a `scratch_pad` of pre-configured size; our ISA

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

Table 3.3: **PULSE adapts a restricted subset of RISC-V ISA** (§3.12.1).

supports register operations directly on the `scratch_pad` and provides special `RETURN` instruction that simply terminates the iterator execution and yields the contents of the `scratch_pad` as the return value.

Finally, we found that the iterator traversal pattern typically can be broken down into two types of computation — fetching data[6] pointed to by `cur_ptr` from memory, and processing the fetched data to determine what the next pointer should be, or if the iterator execution should terminate. If the translation from the iterator code to PULSE's ISA is done naively, it can result in multiple unnecessary loads within the vicinity of the memory location pointed to by `cur_ptr`. For instance, the `unordered_map::find()` realization shown in Listing 3.3 makes references to `cur_ptr->key`, `cur_ptr->value` and `cur_ptr->next` at various points, and if each incurs a separate load, it will slow down execution and waste memory bandwidth. Consequently, PULSE's dispatch engine *infers* the range of memory locations accessed relative to `cur_ptr` in the `next()` and `end()` functions via static analysis and aggregates these accesses into a single large `LOAD` (of up to 256 B) at the beginning of each iteration.

Bounding complexity of offloaded code. While PULSE's interface and ISA already limit the *types* of computation than can be performed per iteration, PULSE also needs to limit the *amount* of computation per iteration to ensure the operations offloaded to PULSE accelerators remain memory-centric. To this end, PULSE's dispatch engine analyzes the generated ISA for the iterator to determine the time required to execute computational logic (t_c) and the time required to perform the single data load at the beginning of the iteration (t_d).

PULSE exploits the known execution time of its accelerators in terms of time per compute in-

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

struction, t_i, to determine $t_c = t_i \cdot N$, where N is the number of instructions per iteration. The CPU node offloads the iterator execution only if $t_c \leq \eta \cdot t_d$, where η is a predefined accelerator-specific threshold. Note that since we only want to offload memory-centric operations, $\eta \leq 1$. As we will show in §3.12.1, the choice of η allows PULSE to maximize the memory bandwidth utilization and ensure processing never becomes a bottleneck for pointer traversals.

Issuing network requests to accelerator. Once the dispatch engine decides to offload an iterator execution, it encapsulates the ISA instructions (`code`) along with the initial value of `cur_ptr` and `scratch_pad` (initialized by `init()`) into a network request. It issues the request, leaving the network to determine which memory node it should be forwarded to (§3.13). To recover from packet drops, the dispatch engine embeds a request identifier (ID) with the CPU node ID and a local request counter in the request packets, maintains a timer per request, and retransmits requests on timeout.

Practical deployability. Our software stack is readily deployable due to its use of real-world toolchains. Our user library adapts implementations of common data structures used in key-value stores [24,138], databases [185–187,189,232], and big-data analytics [135,136,191,192] to PULSE's iterator interface (§3.12). PULSE's dispatch engine is implemented on Intel DPDK-based [252] low-latency, high-throughput UDP stack. PULSE compiler adapts the Sparc backend of LLVM [253] since its ISA is close to PULSE's ISA. Our LLVM frontend applies a set of analysis and optimization passes [254] to enforce PULSE constraints and semantics: the analysis pass identifies code snippets that require offloading, while the optimization pass translates pointer traversal code to PULSE ISA.

PULSE Accelerator Design

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

- **Property 1:** Each iteration involves two clearly separated but sequentially dependent steps: (i) fetching data from memory via a pointer (*e.g.*, a list or tree node), followed by (ii) executing logic on the fetched data to identify the next pointer. The logic cannot be executed concurrently with or before the data fetch, and the next data fetch cannot be performed until the logic execution yields the next pointer.

- **Property 2:** Iterators that benefit from offload spend more time in data fetch (t_d) than logic execution (t_c), i.e., $t_c < \eta \cdot t_d$, where $\eta \leq 1$, as noted in §3.12.1.

Any accelerator for iterator executions must have a *memory pipeline* and a *logic pipeline* to support the execution steps (i) and (ii) above. The strict dependency between the steps (Property 1) renders many optimizations of traditional multi-core processors, such as out-of-order execution, ineffective. Moreover, since each core in such architectures has tightly coupled logic and memory pipelines, the memory-intensive nature of iterators (Property 2) results in the logic pipeline remaining idle most of the time. These two factors combined result in poor utilization and energy efficiency for such architectures. Fig. 3.12 (top) captures this through the execution of 3 iterators (A, B, C), each with 2 iterations (*e.g.*, A1, A2, etc.), on a multi-core architecture. Since each iteration comprises a data fetch followed by a dependent logic execution, one of the pipelines remains idle while the other is busy. While thread-level parallelism permits iterator requests to be spread across multiple cores for increased overall throughput, per-core under-utilization of logic and memory pipelines persists, resulting in suboptimal resource and energy usage.

Disaggregated accelerator design. Motivated by the unique properties of iterators, we propose a novel accelerator architecture that *disaggregates memory and logic pipelines*, using a scheduler to multiplex corresponding components of iterators across them. First, such a decoupling permits an asymmetric number of logic and memory pipelines to maximize the utilization of either pipeline, in stark contrast to the tight coupling in multi-core architectures. In our design, if there are m logic and n memory pipelines, then the accelerator-specific threshold $\eta < 1$ we alluded to in §3.12.1 is $\frac{m}{n}$, i.e., there are fewer logic pipelines than memory pipelines in keeping with Property 2. Fig. 3.12 (bottom) shows an example of our disaggregated accelerator design with one logic pipeline and two memory pipelines (i.e., $m = 1, n = 2$).

Even though data fetch and logic execution within each iterator must be sequential, the disaggregated design permits efficient multiplexing of data fetch and logic execution from different iterators across the disaggregated logic and memory pipelines to maximize utilization. To see how, recall that the logic execution time t_c for each offloaded iterator execution in PULSE is $\leq \eta \cdot t_d$, where t_d is its data fetch time (§3.12.1). Consider the extreme case where $t_c = \eta \cdot t_d$ for all offloaded iterator executions — in this case, it is always possible to multiplex $m + n$ concurrent iterator executions

to fully utilize all m logic and n memory pipelines. While we omit a theoretical proof for brevity, Fig. 3.12 (bottom) illustrates the multiplexed execution — orchestrated by a scheduler in our accelerator — for $t_c = \frac{1}{2} \cdot t_d$ with 3 iterators. This is the ideal case — similar multiplexing is still possible if $t_c \leq \eta \cdot t_d$ with complete utilization of memory pipelines, albeit with lower utilization of logic pipelines (since they will be idle for $\frac{t_c - \eta \cdot t_d}{t_c}$ fraction of time). As such, we provision $\eta = \frac{m}{n}$ to be as close to the expected $\frac{t_c}{t_d}$ for the workload to maximize the utilization of logic pipelines. It is possible to improve the logic pipelines' energy efficiency by dynamically down-scaling frequency [255]; we leave such optimizations to future work.

While the memory pipeline is stateless, the logic pipeline must maintain the state for the iterator it executes. To multiplex several iterator executions, logic pipelines need efficient mechanisms for efficient context switching. To this end, we maintain a dedicated *workspace* corresponding to each iterator's execution. Each workspace stores three distinct pieces of state: `cur_ptr` and `scratch_pad` to track the iterator state described in §3.12, and `data`, which holds the data loaded from memory for `cur_ptr`. A dedicated workspace per iterator allows the logic pipeline to switch to any iterator's execution without delay when triggered by the scheduler, although it requires maintaining multiple workspaces — a maximum of $m + n$ to accommodate any possible schedule due to our bound on the number of concurrent iterators. We divide these workspaces equally across logic pipelines.

PULSE Accelerator Components. PULSE accelerator comprises n memory and m logic pipelines for executing iterator requests, a scheduler that multiplexes requests across the logic and memory pipelines, and a network stack for parsing pointer-traversal requests from the network (Fig. 3.13).

Memory pipeline: Each memory pipeline loads data from the attached DRAM to the corresponding workspace assigned by the scheduler at the start of each iteration. This involves (i) address translation and (ii) memory protection based on page access permissions. We realize range-based address translations (simulated in prior work [256]) in our real-world implementation using TCAM to reduce on-chip storage usage.

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

Logic pipeline: Each logic pipeline runs PULSE ISA instructions other than `LOAD`/`STORE` to determine the `cur_ptr` value for the next iteration or, to determine if the termination condition has been met.

Our logic pipeline comprises an ALU to execute the standard arithmetic and logic instructions, as well as modules to support register manipulation, branching, and the specialized `RETURN` instruction execution (Table 3.3). During a particular iterator's execution, the logic pipeline performs its corresponding instructions with direct reads and updates to its dedicated workspace registers. An iteration's logic can end in one of two possible ways: (i) the `cur_ptr` has been updated to the next pointer, and the `NEXT_ITER` instruction is reached, or (ii) the pointer traversal is complete, and the `RETURN` instruction is reached. In either case, the logic pipeline notifies the scheduler with the appropriate signal.

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

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

While the scheduler assigns memory and logic pipelines to an iterator in steps 1 and 3 in a manner that maximizes utilization of all memory pipelines (i.e., Fig. 3.12 (bottom)), it is possible to implement other scheduling policies.

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

execution (`cur_ptr`, `scratch_pad`).

The network stack uses the same format for both requests and responses, so a response can be sent back to the CPU node on traversal completion or rerouted as a request to a different memory node for continued execution (§3.13).

Implementation. We use an FPGA-based NIC (Xilinx Alveo U250) with two 100 Gbps ports, 64 GB on-board DRAM, 1,728K LUTs, and 70 MB BRAM. Since the board has two Ethernet ports and four memory channels, we partition its resources into two PULSE accelerators, each with a single Ethernet port and two memory channels. Our analysis of common data structures (§3.14) shows their t_c/t_d ratio tends to be < 0.75. As such, we set $\eta = 0.75$, i.e., there are four memory and three logic pipelines and a total of 7 workspaces on the accelerator. We use the Xilinx TCAM IP [257] (for page tables), 100 Gbps Ethernet IP, link-layer IPs [258], and burst data transfers [259] to improve memory bandwidth. The logic and memory pipelines are clocked at 250 MHz, while the network stack operates at 322 MHz for 100 Gbps traffic. Our FPGA prototype showcases PULSE's potential; we believe that ASIC implementations are the next natural step.

3.13 Distributed Pointer Traversals

By restricting pointer traversals to a single memory node (§3.11.2), prior approaches leave applications with two undesirable options. At one extreme, they can confine their data to a single memory, but sacrifice application scalability. Conversely, they can spread their data across multiple nodes but have to return the CPU node whenever the traversal accesses a pointer on another memory node. This affords scalability but costs additional network and software processing latency at the CPU node. To avoid the cost, one may replicate the entire translation and protection state for the cluster at every memory node so they can directly forward traversal requests to other memory nodes. This comes at the cost of increased space consumption for translation, which is challenging to contain within the accelerator's translation and protection tables. Moreover, duplicating this state across memory nodes requires complex protocols for ensuring their consistency (*e.g.*, when the state changes), which have significant performance overheads.

PULSE breaks this tradeoff between performance and scalability by leveraging a programmable network switch to support rack-scale distributed pointer traversals. In particular, if the PULSE accel-

erator on one memory node detects that the next pointer lies on a different memory node, it forwards the request to the network switch, which routes it to the appropriate memory node for continuing the traversal. This cuts the network latency by half a round trip time and avoids software overheads at the CPU node, instead performing the routing logic in switch hardware. Since continuing the traversal across memory nodes is similar to packet routing, the switch hardware is already optimized to support it.

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

Hierarchical translation. For the switch to forward the pointer traversal request to the appropriate memory node, it must be able to locate which memory nodes are responsible for which addresses. To minimize the logic and state maintained at the switch due to its limited resources, PULSE employs hierarchical address translation as shown in Fig. 3.14. In particular, the address space is range partitioned across memory nodes; PULSE only stores the base address to memory node mapping at the switch, while each memory node stores its own local address translation and protection metadata at the accelerator (①), as outlined in §3.12.1. The routing logic at the switch inspects the `cur_ptr` field in the request (②) and consults its mapping to determine the target memory node (③). At the memory node, the traversal proceeds until the accessed pointer is not present in the local table (as in ①); it then sends the request back to the switch (§3.12.1), which can re-route the request to the appropriate memory node (④-⑥), or notify the CPU node if the pointer is invalid.

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

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

Table 3.4: **Workloads used in our evaluation (§3.14).** t_c and t_d correspond to compute and memory access time at the PULSE accelerator.

3.14 Evaluation

Compared systems. We compare PULSE against: (1) a **Cache-based** system that relies solely on caches at CPU nodes to speed up remote memory accesses; we use Fastswap [31] as the representative system, (2) an **RPC** system that offloads pointer-traversals to a CPU on memory nodes, (3) **RPC-ARM**, an RPC system that employs a wimpy ARM processors at memory nodes, and (4) a **Cache+RPC** approach that employs data structure-aware caches; we use AIFM [28] as the representative system. (1, 4) use a cache size of 2 GB, while (2, 3) use a DPDK-based RPC framework [260].

Our experimental setup comprises two servers, one for the CPU node and the other for memory nodes, connected via a 32-port switch with a 6.4 Tbps programmable Tofino ASIC. Both servers were equipped with Intel Xeon Gold 6240 Processors [261] and 100 Gbps Mellanox ConnectX-5 NICs. For a fair comparison, we limit the memory bandwidth of the memory nodes to 25 GB/s (FPGA's peak bandwidth) using Intel Resource Director [262] and report energy consumption of the **minimum** number of CPU cores needed to saturate the bandwidth. We use Bluefield-2 [263] DPU as our ARM-based SmartNICs with 8 Cortex-A72 cores and 16 GB DRAM. For PULSE, we placed two memory nodes on each FPGA NIC (one per port, a total of 4 memory nodes). Our results translate to larger setups since PULSE's performance or energy efficiency are independent of dataset size and cluster scale.

Applications & workloads. We consider 3 applications with varying data structure complexity, compute/memory-access ratio, and iteration count per request (Table 3.4): (1) *Web Service* [28] that processes user requests by retrieving user IDs from an in-memory hash table, using these IDs to fetch 8KB objects, which are then encrypted, compressed and returned to the user. Requests are generated using YCSB A (50% read/50% update), B (95% read/5% update), and C (100% read) workloads with Zipf distribution [139]. (2) *WiredTiger Storage Engine* (MongoDB backend [264])

uses B+Trees to index NoSQL tables. Our frontend issues range query requests over the network to WiredTiger and plots the results. Similar to prior work [28, 265], we model user queries using the YCSB E workload with Zipf distribution [139] on 8B keys and 240B values. (3) *BTrDB Time-series Database* [239] is a database designed for visualizing patterns in time-series data. BTrDB reads the data from a B+Tree-based store for a given user query and renders the time-series data through an interactive user interface [266]. We run stateful aggregations (sum, average, min, max) for time windows of different resolutions, from 1s to 8s, on the Open μPMU Dataset [267] with voltage, current, and phase readings from LBNL's power grid [239].

3.14.1 Performance for Real-world Applications

Since AIFM [28] does not natively support B+-Trees or distributed execution, we restrict the Cache+RPC approach to the Web Service application on a single node.

Single-node performance. Fig. 3.15 demonstrates the advantages of accelerating pointer-traversals at disaggregated memory. Compared to the Cache-based approach, PULSE achieves 9–34.4$\times$ lower latency and 28–171$\times$ higher throughput across all applications using only one network round-trip per request. RPC-based systems observe 1–1.4$\times$ lower latency than PULSE due to their 9$\times$ higher CPU clock rates. We believe an ASIC-based realization of PULSE has the potential to close or even overcome this gap. Cache+RPC incurs higher latency than RPC due to its TCP-based DPDK stack [28, 268] and does not outperform RPC, indicating that data structure-aware caching is not beneficial due to poor locality.

Latency depends on the number of nodes traversed during a single request and the response size. WebService experiences the highest latency due to large 8KB responses and long traversal length per request. In BTrDB, the latency increases (and the throughput decreases) as the window size grows due to the longer pointer traversals (see Table 3.4). Interestingly, the Cache-based approach performs significantly better for BTrDB than WebService and WiredTiger due to the better data locality in time-series analysis of chronologically ordered data. However, its throughput remains significantly lower than both PULSE and RPC since it is bottlenecked by the swap system performance, which could not evict pages fast enough to bring in new data. This is verified in our analysis of resource utilization (deferred to Appendix for brevity); we find that RPC, RPC-

ARM, Cache+RPC, and PULSE can utilize more than 90% of the memory bandwidth across the applications, while the Cache-based approach observes less than 1 Gbps network bandwidth. The other systems — PULSE, RPC, RPC-ARM, and Cache+RPC — can also saturate available memory bandwidth (around 25 GB/s) by offloading pointer traversals to the memory node, consuming only 0.5%–25% of the available network bandwidth.

Distributed pointer traversals. Fig. 3.15 shows that employing multiple memory nodes introduces two major changes in performance trends: (1) the latency increases when the pointer traversal spans multiple memory nodes, and (2) throughput increases with the number of nodes since the systems can exploit more CPUs or accelerators. WebService is an exception to the trend: since the hash table is partitioned across memory nodes based on primary keys, the linked list for a hash bucket resides in a single memory node.

PULSE observes lower latency than the compared systems due to in-network support for distributed pointer-traversals (§3.13). The latency increases significantly from one to two memory nodes for all systems since traversing to the next pointer on a different memory node adds 5–10 μs network latency. Also, even across two memory nodes, a request can trigger multiple inter-node pointer traversals incurring multiple network round-trips; for WiredTiger and BtrDB, 10%–30% of pointer traversals are inter-node. However, in-network traversals allow PULSE to reduce latency overheads by 33–98%, with 1.1–1.36$\times$ higher throughput than RPC.

Energy consumption. We compared energy consumed per request for PULSE and RPC schemes at a request rate that ensured memory bandwidth was saturated for both. We measure energy consumption using Xilinx XRT [233] for PULSE (all power rails) and Intel RAPL tools [234] for RPC on CPUs [261] (CPU package and DRAM only). For RPC-ARM on ARM cores, since there is no power-related performance counter [269] or open-source tool available, we adapt the measurement approach from prior work [227]. Specifically, we calculate the CPU package's energy using application CPU cycle counts and DRAM power using Micron's estimation tool [270]. Finally, we conservatively estimate ASIC power using our FPGA prototype: we scale down the ASIC energy only for PULSE accelerator using the methodology employed in prior research [271] while using the unscaled FPGA energy for other components (DRAM, third-party IPs, etc.). As such, we measure an *upper bound* on PULSE and PULSE-ASIC energy use, and a *lower bound* for RPC, RPC-ARM,

and Cache+RPC.

Fig. 3.16 shows that PULSE achieves a 4.5–5× reduction in energy use per operation compared to RPCs on a general-purpose CPU, due to its disaggregated architecture (§3.12.1). Our estimation shows that PULSE's ASIC realization can conservatively reduce energy use by an additional 6.3 − 7× factor. Finally, RPC-ARM's total energy consumption per request can exceed that of standard cores, as seen in the WebService workload. This observation aligns with prior studies [227], which attribute the increased energy use to their longer execution times, resulting in higher aggregate energy demands.

3.14.2 Understanding PULSE Performance

Distributed pointer traversals. We evaluate the impact of distributed pointer traversals (§3.13) by comparing PULSE against PULSE-ACC, a PULSE variant that sends requests back to the CPU node if the next pointer is not found on the memory node. Fig. 3.17 shows that while both have identical performance on a single memory node, PULSE-ACC observes 1.02–1.15× higher latency for two nodes. On the other hand, their throughput is the same since, under sufficient load, memory node bandwidth bottlenecks the system for both.

Latency breakdown for PULSE accelerator. Fig. 3.18 shows the latency contributions of various hardware components at the PULSE accelerator for the WebService application. The network stack first processes the pointer traversal request in about 430 ns, after which the WebService payload is processed by the scheduler and dispatched to an idle memory access pipeline in 5.1 ns. Then, the memory pipeline takes ∼132 ns to perform address translation, memory protection, and data fetch from DRAM. Finally, the logic pipeline takes 10 ns to check the termination conditions and determine the next pointer to look up. This process repeats until the termination condition is met. The time to send a response back over the network stack is symmetric to the request path.

3.15 Future Trends and Research

While PULSE is implemented atop Ethernet, its design is interconnect-agnostic and could be realized in ASIC-based or FPGA-attached memory devices over emerging interconnects like CXL [10, 231, 272]. We have verified these benefits in simulation atop detailed memory access and processing

traces of our evaluated applications and workloads. The simulator maintains 2GB of cache in local (CPU-attached) DRAM, while the entire working set is stored on remote CXL memory. Following prior work [37], we model 10–20ns L3 cache latency, 80ns local DRAM latency, 300ns CXL-attached memory latency, and 256B access granularity. We simulate both a four-memory-node setup, which uses a CXL switch with PULSE logic and a PULSE accelerator at each memory node, and a single-node setup with no switch. We assume a conservative overhead for PULSE, using our hardware programmable Ethernet switch and FPGA accelerator latencies.

Fig. 3.19 shows the average slowdown for executing our evaluated workloads on CXL memory relative to running it completely locally (i.e., the entire application working set fits in local DRAM) — with and without PULSE. In the four-node setup, PULSE reduces CXL's slowdown by 19–33% across all applications.

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

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

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

Listing 3.2: C++ STL realization for `unordered_map::find()`.

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

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

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

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

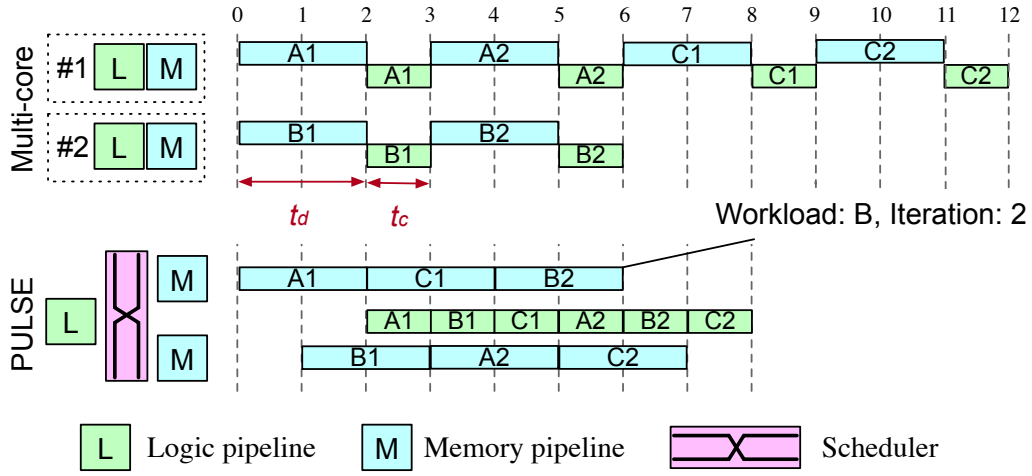


Fig. 3.12: **PULSE accelerator architecture.** (top) Traditional multi-core architectures with tightly coupled logic and memory pipelines result in low utilization and longer execution times. (bottom) PULSE accelerator's *disaggregated* design with an unequal number of logic and memory pipelines efficiently multiplexes concurrent iterator executions across them for near-optimal utilization and performance.

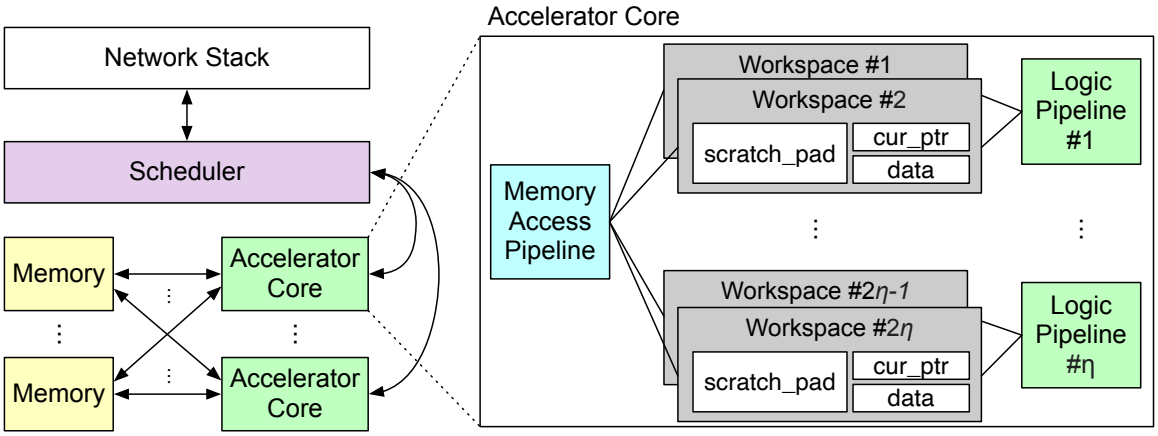


Fig. 3.13: **PULSE accelerator overview.** See §3.12.1 for details.

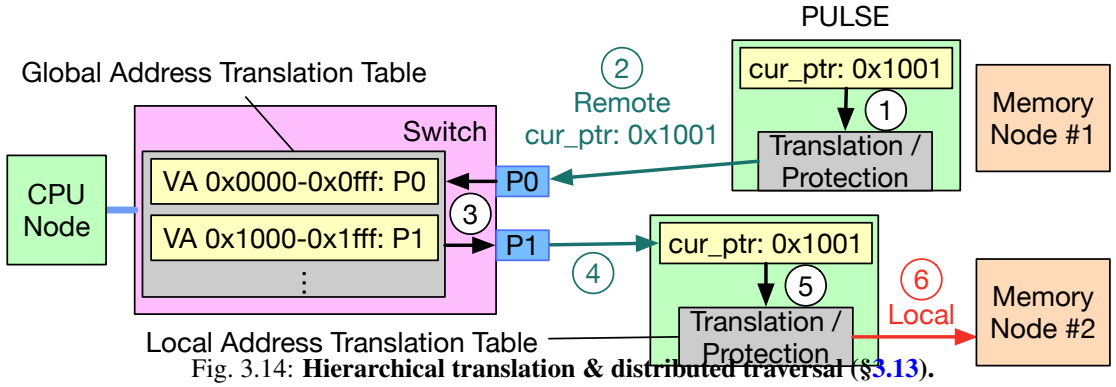


Fig. 3.14: **Hierarchical translation & distributed traversal (§3.13).**

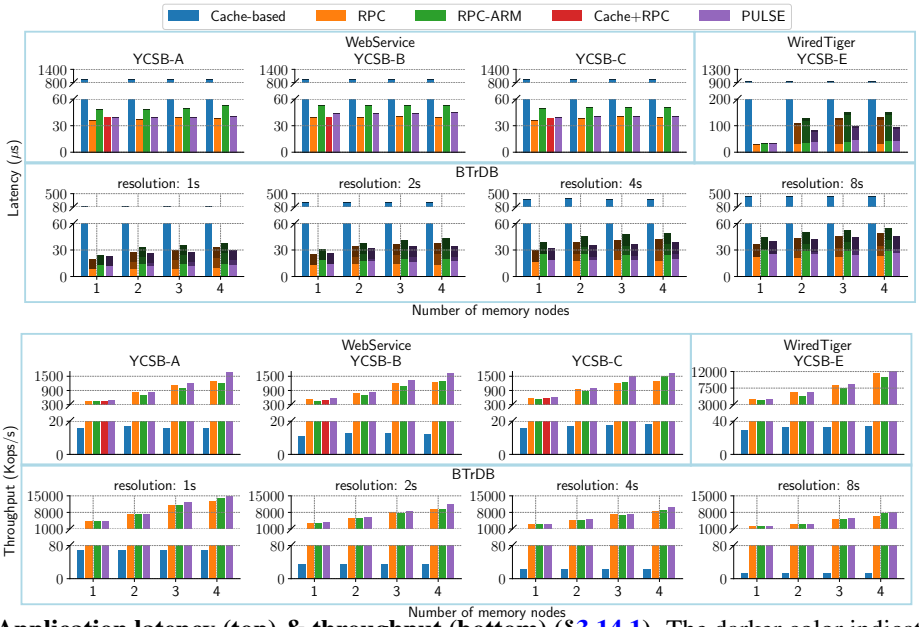


Fig. 3.15: **Application latency (top) & throughput (bottom) (§3.14.1).** The darker color indicates the time spent on cross-node pointer traversals, which increases with the number of memory nodes in WiredTiger and BTrDB.

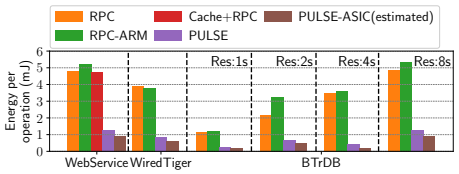


Fig. 3.16: **Application energy consumption per operation (§3.14.1).**

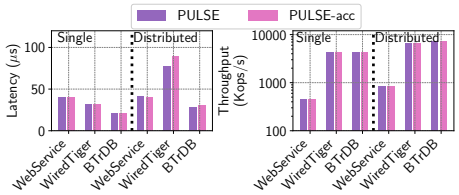


Fig. 3.17: **Impact of distributed pointer traversals (§3.14.2).**

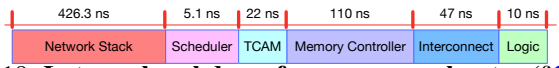


Fig. 3.18: **Latency breakdown for PULSE accelerator (§3.14.2).**

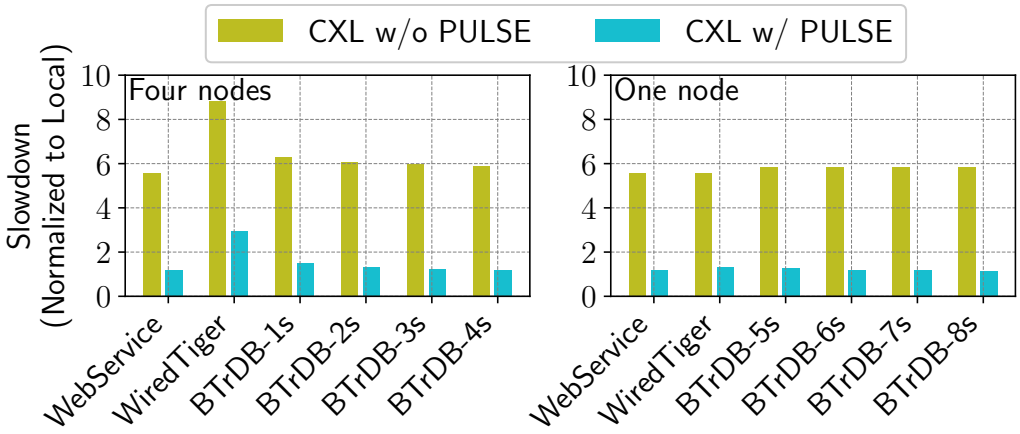


Fig. 3.19: **Slowdown with simulated CXL interconnect (§3.15).**

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 [273] from Nvidia and Compute Express Link (CXL) [10] from Intel. CXL, in particular, has been introduced as a promising solution to expand memory capacity and bandwidth by attaching external memory devices to PCIe slots, offering a dynamic and heterogeneous computing environment.

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

hardware.

Despite extensive research on CXL [272, 274, 275], 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 [275, 276], 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) [277], 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 [278] has investigated promoting hot pages from slower-tiered memory at the kernel level to enhance performance while maintaining application transparency.

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

4.2 Introduction

In an age marked by the surge of memory-intensive applications, such as machine learning tasks and High-Performance Computing (HPC) applications, there is an urgent need for expanding the memory capacity and bandwidth [279–281]. For instance, a machine learning application with 175 B model requires 700 GB of memory to hold its parameters only, not to mention memory requirements for intermediate results and others. That is, the memory requirements of modern applications could easily exceed the memory capability of a single machine due to physical constraints, such as availability of DDR DIMM slots and thermal issues, as well as cost considerations of employing high-density DIMMs [280, 281].

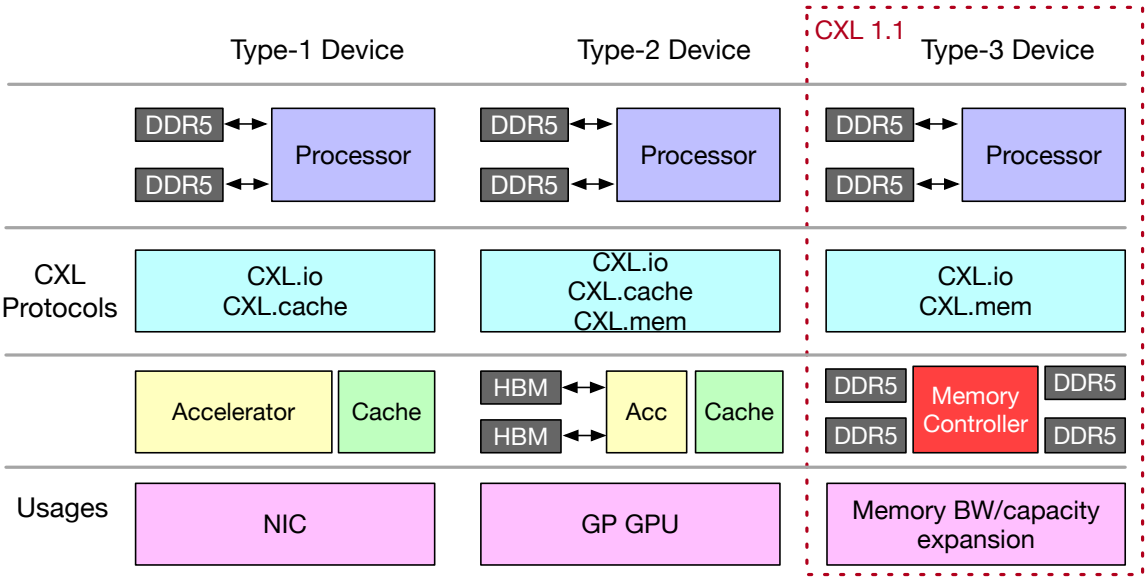


Fig. 4.1: **CXL Overview.** In this study, we focus on commercial CXL 1.1 Type-3 devices, leveraging CXL.io and CXL.mem protocols for memory expansion in single-server environments.

To meet such urgent demands, Compute Express Link (CXL) [10,272,274,281] is introduced as a groundbreaking interconnect technology. CXL promises significant expansion of memory capacity and bandwidth by attaching external memory devices (e.g., DRAM, Flash or persistent memory) to PCIe slots. Unlike its predecessors, CXL enables a more dynamic and heterogeneous computing environment, leading to various design trade-offs for performance and cost gains. Commercially debuting with version 1.1, CXL allows direct attachment of external memory devices to the host machine, enabling a unified and coherent memory address space. In such configuration, CXL is predominantly used as a way of memory expansion. For example, AsteraLabs' A1000 [282] CXL memory expansion card supports up to 4xDDR5 RDIMMs, enabling up to 2 TB of additional memory for a single server.

Although substantial studies on CXL memory have been performed in the past [37, 272, 274, 275, 278, 281, 283, 284], there remains a significant gap of employing these studies to guide the integration of CXL practically. In particular, we observe the following issues: (1) Much of the current literature has focused on evaluating CXL hardware through simulations [37, 274] or using FPGA-based setups [275, 284]. Although a limited number of studies have begun to assess the raw performance of ASIC-based CXL hardware [275, 285], there remains a gap in understanding how different system configurations influence the performance of data center applications using CXL memory. Furthermore, the specific applications that could substantially benefit from CXL memory

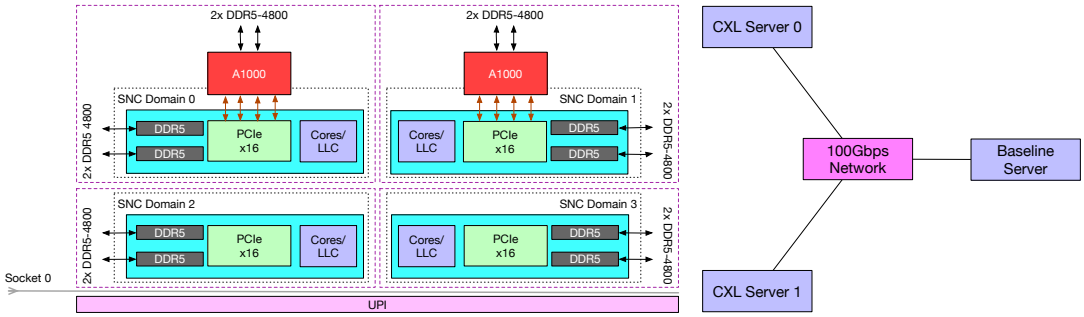
expansion are not yet fully identified. (2) While existing studies have begun to explore the cost implications of employing CXL technology, such as the work on memory pooling cost models presented in [286], a critical gap remains in understanding the cost-effectiveness of migrating particular types of applications or services to memory expansions facilitated by CXL. (3) Given the restricted availability of CXL ASIC hardware, the research community faces a notable scarcity of open-source empirical data. This limitation hinders efforts to fully comprehend the performance capabilities of such hardware or to develop performance models based on empirical evidence.

Our study aims to fill existing knowledge gaps by conducting detailed evaluations of CXL 1.1 for memory-intensive applications, leading to several *intriguing observations*: Contrary to the common perception that CXL memory, due to its higher latency, should be considered a separate, slower tier of memory [37,278], **we find that shifting some workloads to CXL memory can significantly enhance performance**, even if local memory's capacity and bandwidth are underutilized. This is because using CXL memory can decrease the overall memory access latency by alleviating bandwidth contention on DDR channels, thereby improving application performance. From our analysis of application performance, we have formulated an abstract cost model (§4.7) that predicts substantial cost savings in practical deployments.

In summary, the major contributions of this paper are:

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

The paper organizes as follows. §4.3 introduces basic information of CXL and environment setup for the evaluations. §4.4 presents basic performance characteristic of CXL memory expansion. §4.5 and §4.6 presents findings and suggestions of using CXL as the expansion of memory capacity



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

Fig. 4.2: **CXL Experimental Platform.** (a) Each CXL server is equipped with two A1000 memory expansion cards. SNC-4(§4.4.1) is enabled only for the raw performance benchmarks(§4.4) and bandwidth-bound benchmarks(§4.6), and each SNC Domain is equipped with two DDR5 channels. (a) illustrates Socket 0; Socket 1 shares a similar setup except for the absence of CXL memory. (b) Our platform comprises two CXL servers and one baseline server. The baseline server replicates the same configuration but lacks any CXL memory cards.

and bandwidth on data center workloads. §4.7 provides a detailed analysis on the potential cost benefits brought by CXL. §3.8 discusses how our insights are applicable to future generations of CXL. §3.9 describes related work, and §**??** concludes the paper.

4.3 Background and Methodology

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

4.3.1 Compute Express Link (CXL) Overview

Compute Express Link (CXL) [**?**] is a standardized interconnect technology that facilitates communication between processors and various devices, including accelerators, memory expansion units, and smart I/O devices. CXL is built upon the physical layer of PCI Express® (PCIe®) 5.0 [287], providing native support for x16, x8, and x4 link widths with data rates of 32.0 GT/s and 64.0 GT/s. The CXL transaction layer is implemented through three protocols: CXL.io, CXL.cache, and CXL.mem, as depicted in Fig. 4.1. *CXL.io* protocol is based on PCIe 5.0 and handles device discovery, configuration, initialization, I/O virtualization, and direct memory access (DMA). *CXL.cache* enables CXL devices to access the host processor's memory. *CXL.mem* allows the host to access memory attached to devices using load/store commands.

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

The commercially available version of CXL is 1.1, where a CXL 1.1 device can only serve as a single logical device accessible by one host at a time. Future generations of CXL, like CXL 2.0, are expected to support the partitioning of devices into multiple logical units, enabling up to 16 different hosts to access different portions of memory [288]. In this paper, our focus is on commercially available CXL 1.1 Type-3 devices, specifically addressing single-host memory expansion.

4.3.2 Hardware Support for CXL

Recent announcements have introduced CXL 1.1 support for Intel Sapphire Rapids processors (SPR) [289] and AMD Zen 4 EPYC "Genoa" and "Bergamo" processors [290]. While commercial CXL memory modules are provided by vendors such as Asteralabs [282], Montage [291], Micron [270], and Samsung [285], CXL memory expanders are predominantly in prototype stages, with only limited samples available, making access difficult for university labs. Consequently, due to the scarcity of CXL hardware, research into CXL memory has largely depended on NUMA-based emulation [37, 278] and FPGA implementations [275, 284], each with inherent limitations:

NUMA-based emulation. Given the cache coherent nature and comparable transfer speed of CXL and UPI/xGMI interconnects, NUMA-based emulation [37, 278] is widely adopted to enable fast application performance analysis and software prototyping as the CXL memory is exposed as a remote NUMA node. However, NUMA-based emulation fails to accurately capture the performance characteristics of CXL memory due to differences from CXL and UPI/xGMI interconnects [292], as shown in previous research [275].

FPGA-based implementation. Intel and other hardware vendors use FPGA hardware to imple-

ment CXL protocols [276], bypassing the performance inconsistencies of NUMA-based emulation. However, FPGA-based CXL memory falls short in fully utilizing memory chip performance due to its lower operating frequency compared to ASICs [293]. FPGAs prioritize flexibility over performance and are suitable for early-stage CXL memory validation but not production deployment. Intel's recent evaluation [275] uncovered performance issues in FPGA implementations, including reduced memory bandwidth during concurrent thread execution. This hampers rigorous evaluations for memory capacity- and bandwidth-bound applications, which are key use cases for CXL memory expanders. Further discussion on the performance disparity between CXL ASIC and FPGA controllers is in §4.4.

To the best of our knowledge, we are one of the pioneers in uncovering the performance characteristics of actual ASIC prototypes designed for CXL memory expansion. The ASIC CXL memory controller we have employed is the A1000 [282] developed by AsteraLabs, which implements the CXL interface at speeds of up to 32 GT/s per lane, supporting up to 16 lanes in total. This controller has the capability to accommodate up to 4 DDR5-5600 RDIMM slots, providing a total memory capacity of 2TB.

4.3.3 Software Support for CXL

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

N:M Interleave Policy for Tiered Memory Nodes.

Traditional memory interleave policies distribute data evenly across memory banks, often using a 1:1 ratio. However, the advent of tiered memory systems, which feature CPU-less memory nodes with diverse performance traits, demands more nuanced strategies for optimizing memory bandwidth, especially for bandwidth-heavy applications. The interleave patch [294] introduces an innovative N:M interleave policy to address this, allowing for an allocation scheme where N pages are directed to high-performance (top-tier) nodes and M pages to lower-tier nodes. For example, us-

ing a 4:1 ratio directs 80% of traffic to top-tier nodes and 20% to low-tier nodes, adjustable through the `vm.numa_tier_interleave` parameter. While the patch showcases compelling evaluation results [294], it's crucial to note that optimal memory distribution depends on specific hardware and application characteristics. Given the higher latency of CXL memory, as demonstrated in §4.4, performance-sensitive applications should undergo thorough profiling and benchmarking to maximize the advantages of interleaving and mitigate potential performance trade-offs.

NUMA Balancing & Hot Page Selection.

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

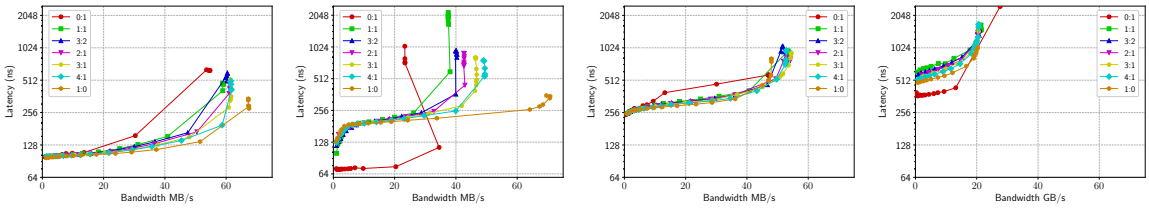
1. The *NUMA-balancing* patch [295] uses a latency-aware page migration strategy, focusing on promoting recently accessed pages (MRU). It scans NUMA balancing page tables and hints page faults. However, it may not accurately identify high-demand pages due to extended scanning intervals, potentially causing latency issues for some workloads.

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

Additionally, research prototypes like TPP [278] share a similar concept with optimizations and are being considered for integration into the Linux Kernel [296]. However, we faced challenges with TPP when running memory-bandwidth-intensive applications, resulting in unexplained performance degradation. Hence, we rely on the well-tested kernel patches integrated into Linux Kernel since version 6.1.

4.3.4 Experimental Platform Description

The evaluation testbed, as illustrated in Fig. 4.2(b), consists of three servers. Two of these servers are designated as CXL experiment servers. Each of these servers is equipped with dual Intel Xeon



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

Fig. 4.3: **Overall effect of read-write ratio on MMEM and CXL across different distances.** The workloads are represented by read:write ratios (e.g., 0:1 for write-only, 1:0 for read-only). Accessing CXL memory locally incurs higher latency compared to MMEM but is more comparable to accessing MMEM on a remote socket. MMEM bandwidth peaks at 67 GB/s, versus 54.6 GB/s for CXL memory. Performance significantly declines when accessing CXL memory on a remote socket (§4.4.2). In specific scenarios, such as the write-only workload (0:1) in (b), the plot may show instances where bandwidth decreases and latency increases with heavier loads. The Y-axis is on a logarithmic scale.

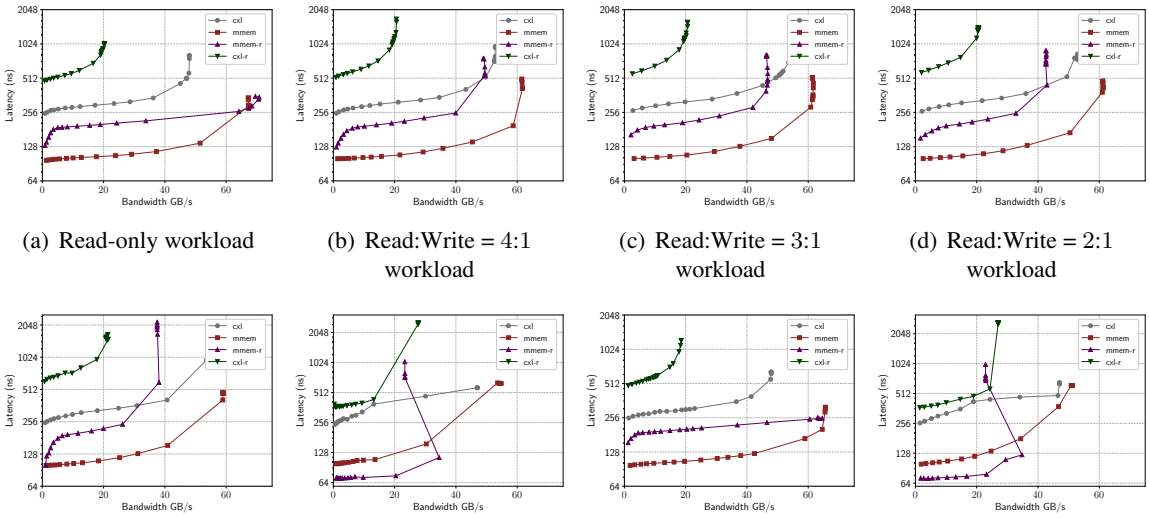
4th Generation CPUs (Sapphire Rapids, or SPR), 1 TB of 4800 MHz DDR5 memory, two 1.92 TB SSDs, and a pair of A1000 CXL Gen5 x16 ASIC memory expanders modules from AsteraLabs, each with 256 GB of 4800MHz memory (resulting in a total of 512 GB memory per server). Both A1000 memory modules are attached to socket 0. The third server serves as the baseline and is configured identically to the CXL experiment servers, except for the absence of the CXL memory expanders. It is designated for initiating client requests and running workloads that strictly utilize the main memory during the application assessments. All servers are interconnected via 100 Gbps Ethernet links.

4.4 CXL 1.1 Performance Characteristics

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

4.4.1 Experimental Configuration

For each dual-channel A1000 ASIC CXL memory expander [282], we connect two DDR5-4800 memory channels, achieving a total capacity of 256 GB. To provide a fair comparison between MMEM and CXL-attached DDR5 memory, we utilize the Sub-NUMA Clustering (SNC) [297] feature to ensure the number of memory channels is the same in both settings.



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

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

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

Sub-NUMA Clustering(SNC). Sub-NUMA Clustering (SNC) serves as an enhancement over the traditional NUMA architecture. It decomposes a single NUMA node into multiple smaller semi-independent sub-nodes (domains). Each sub-NUMA node possesses its own dedicated local memory, L3 caches, and CPU cores. In our experimental setup (Fig. 4.2(a)), we partition each CPU into four sub-NUMA nodes. Each sub-NUMA node is equipped with two DDR5 memory channels connected to two 64 GB DDR5-4800 DIMMs. Enabling SNC requires setting the IMC (Integrated Memory Controllers) to 1-way interleaving. According to the specifications, a single DDR5-4800 channel has a theoretical peak bandwidth of 38.4 GB/s [274]. Therefore, each sub-NUMA node has a combined memory bandwidth of up to 76.8 GB/s.

Intel Memory Latency Checker (MLC). We leverage Intel's Memory Latency Checker (MLC) to examine loaded-latency for various read-write workloads, adopting a 64-byte access size same as prior work [275]. We deploy 16 MLC threads, and it's important to note that while the thread count is a configurable parameter in MLC, it doesn't directly dictate memory request concurrency. MLC assigns separate memory segments for each thread to access simultaneously. Specifically, when evaluating loaded latency, MLC incrementally increases the operation rate of each thread.

Our findings indicate that employing 16 threads with MLC precisely measures both the idle and loaded latency and the point at which bandwidth becomes saturated. MLC accommodates a broad spectrum of workloads including those with varied read-write mixes and non-temporal writes.

Our study is focused on addressing the following research questions:

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

4.4.2 Basic Latency and Bandwidth Characteristics

This section outlines our findings on memory access latency and bandwidth for different memory configurations: local-socket main memory (MMEM), remote-socket main memory (MMEM-r), CXL memory (CXL), and remote-socket CXL memory (CXL-r). Figure 4.3(a) shows the loaded latency curve for MMEM under varied read-write mixes. The read-only workload hits a peak bandwidth of roughly 67 GB/s, reaching 87% of its theoretical maximum. Yet, as write operations increase, bandwidth dips, with write-only tasks dropping to 54.6 GB/s. We note an initial memory latency of about 97 ns, which spikes exponentially as bandwidth nears full capacity, a sign of bandwidth contention [298, 299]. Interestingly, latency starts to significantly increase at 75%-83% of bandwidth utilization, surpassing prior estimates of 60% from earlier studies [298].

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

Moreover, latency escalation occurs earlier in remote socket memory accesses than in local ones, primarily due to queue contention at the memory controller.

Fig. 4.3(c) illustrates the latency curve for CXL memory expansion, demonstrating a minimum latency of 250.42 ns. Interestingly, despite additional PCIe and CXL memory controller overhead on the datapath, accessing CXL follows the same "Bandwidth contention" trend as MMEM. The latency of accessing CXL on the same socket remains relatively stable as bandwidth increases, with a maximum bandwidth of around 56.7 GB/s, achieved when the workload is 2:1 read-write ratio. The reduction in maximum bandwidth compared to DRAM is attributed to PCIe overhead, such as extra headers. The maximum bandwidth for read-only workloads is smaller due to PCIe bi-directionality, preventing full bandwidth utilization. Fig. 4.3(d) reveals the latency-bandwidth plot for accessing CXL from a remote socket, incurring an exceptionally high idle latency of 485 ns. In addition, the maximum memory bandwidth is unexpectedly halved, reaching just 20.4 GB/s for 2:1 read-write ratio, which is a much more severe performance drop compared to accessing MMEM from the remote NUMA node in Fig. 4.3(d). Since running a read-only towards a CXL Type-3 device on the remote socket does not generate substantial coherence traffic, initial speculation regarding cache coherence is ruled out. Further investigation utilizing the Intel Performance Counter Monitor (PCM) [300] also confirms that the UPI utilization is consistently below 30%. Discussions with Intel suggest this performance bottleneck is likely due to limitations in the Remote Snoop Filter (RSF) on the current CPU platform, anticipated to be addressed in the next-generation processors [301].

4.4.3 Different Read-Write Ratios & Access Pattern

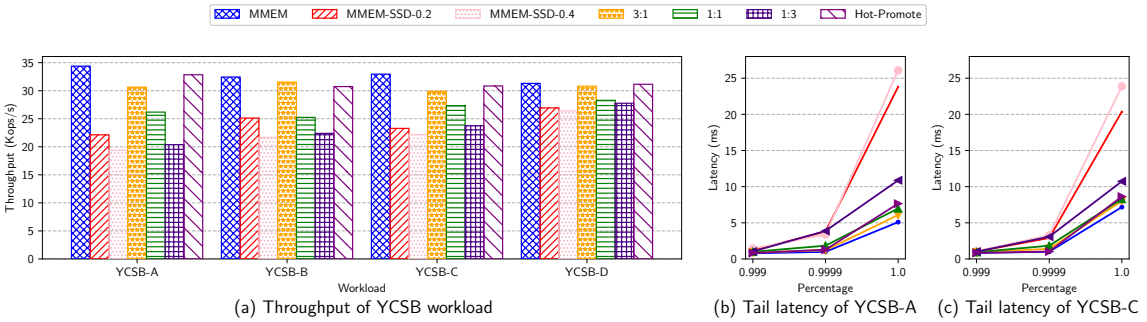
Fig. 4.4(a)-4.4(f) present a performance comparison for a specific workload with varying read-write ratios. The results align with our observation that accessing CXL from a remote socket introduces exceptionally high latency and low bandwidth. When accessing CXL from the same socket, latency is 2.4-2.6 $\times$ that of local DDR and 1.5-1.92 $\times$ that of remote socket DDR. This suggests that running applications directly on CXL may significantly drop performance. However, when workloads span multiple NUMA nodes within the same socket, accessing CXL locally is comparable to accessing remote NUMA node memory. Additionally, the latency-bandwidth knee-point shifts to the left as the proportion of write operations in the workload increases. Fig. 4.4(g) and 4.4(h) display the results of running both read-only and write-only workloads, utilizing random access patterns instead

of sequential access. Notably, we do not observe any significant performance disparities under these conditions.

4.4.4 Key insights

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

Bandwidth Contention Previous research [274, 299] has brought attention to issues related to bandwidth contention. We further examine how memory latency varies with varying read-write ratios under bandwidth contention. While latency remains relatively stable at low to moderate bandwidth utilization levels, it increases exponentially as bandwidth approaches higher levels, primarily due to queuing delays in the memory controller [298]. Furthermore, the knee-point in latency shifts to lower memory bandwidth when there is a higher proportion of write operations in the workload. Interestingly, CXL-attached memory has often been characterized by industry and research community as 'tiered memory' [275, 294, 296], suggesting that it serves as a slower and less performant memory layer to be considered only when MMEM is fully utilized. However, we argue against this simplistic view of CXL-memory. Allocators and kernel-level page placement policies should consider the available bandwidth in MMEM. Even if a substantial portion of memory bandwidth in MMEM remains unused, e.g., 30%, offloading a portion of the workload, e.g., 20%, to CXL memory can lead to overall performance improvements. Our recommendation is to regard CXL memory as a valuable resource for load balancing, even when local DRAM bandwidth is not fully utilized. Subsequent real-world evaluations support these insights (§4.6).



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

Fig. 4.5: **KeyDB YCSB latency and throughput under different configurations.** (a) Average throughput of four YCSB workload under different system configuration. (b) Tail latency of YCSB-A (c) Tail latency CDF of YCSB-C, both reported by the YCSB client [302].

Comparison with FPGA-based CXL implementations. Intel recently disclosed latency and bandwidth performance metrics for their FPGA-based CXL prototype [275]. While they provided insights into relative latency and bandwidth efficiency for soft and hard IP implementations, performance under load was not shared. Our measurements indicate that the ASIC CXL solution only introduces a less than $2.5x$ overhead in access latency compared to MMEM, surpassing most of Intel's measurements. However, the FPGA-based solution achieved only 60% of the PCIe bandwidth due to the inefficiency of the memory controller, while the Asteralabs A1000 prototype reached an impressive 73.6% bandwidth efficiency, clearly outperforming Intel's FPGA-based solution.

4.5 Memory Capacity-bound Applications

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

4.5.1 In-memory key-value stores

Redis [24] is an open-source in-memory key-value store and one of the most popular NoSQL databases. Redis employs a user-defined parameter, `maxmemory`, to limit its memory allocation for storing user data. Like traditional memory allocators (e.g., malloc()), Redis may not return memory to the system after key deletion, particularly if deleted keys were on a memory page with

active ones. This necessitates memory provisioning based on peak demand, making memory capacity the major bottleneck for Redis deployments [303] in data centers. Google Cloud suggests keeping memory usage below 80% [304], whereas other sources recommend a limit of 75% [303].

Due to the substantial infrastructure costs for memory-only deployment, Redis Enterprise [305] is the commercial variant extensively supported by leading cloud platforms (e.g., AWS, Google Cloud, or Azure). It introduces "Auto Tiering" [306] to allow data overflow to SSDs, offering an economically viable option for database expansion beyond the limits of RAM capacity. Given that Redis Enterprise is not accessible on our experiment platform, we employ KeyDB as an alternative. KeyDB extends Redis's capabilities by adding KeyDB Flash, which uses RocksDB for persistent storage. The FLASH feature enables all data is written to the disk for persistence, with hot data remaining in memory as well as disk.

Methodology and Software Configurations.

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

Configuration	Description
`MMEM`	Entire working set in main memory.
`MMEM-SSD-0.2`	20% of the working set is spilled to SSD.
`MMEM-SSD-0.4`	40% of the working set is spilled to SSD.
`3:1`	Entire working set in memory (75% MMEM + 25% CXL, 3:1 interleaved).
`1:1`	Entire working set in memory (50% MMEM + 50% CXL, 1:1 interleaved).
`1:3`	Entire working set in memory (25% MMEM + 75% CXL, 1:3 interleaved).
`Hot-Promote`	Entire working set in memory (50% MMEM + 50% CXL), with hot page promotion kernel patches discussed in §4.3.

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

D. The total amount of working set data is 512 GB.

Analysis.

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

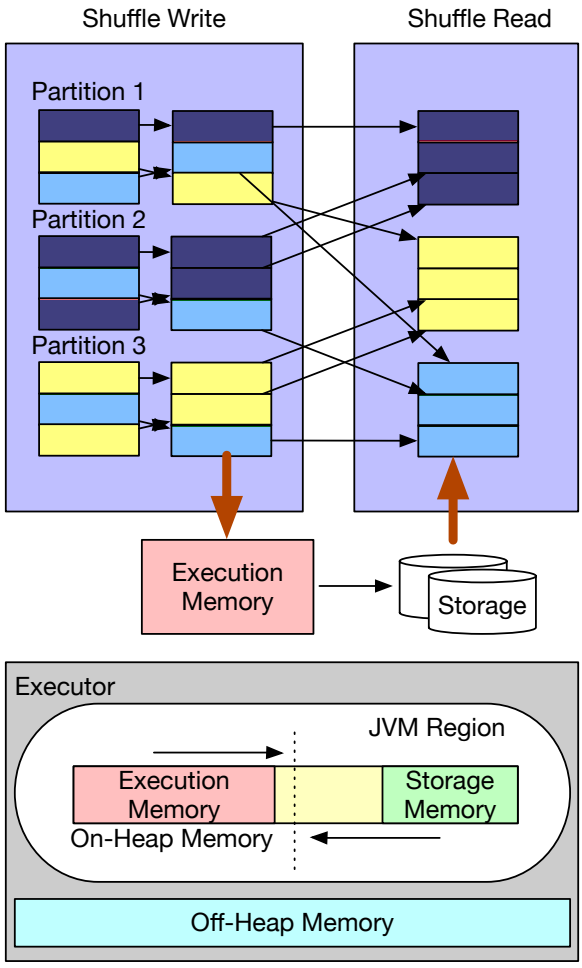


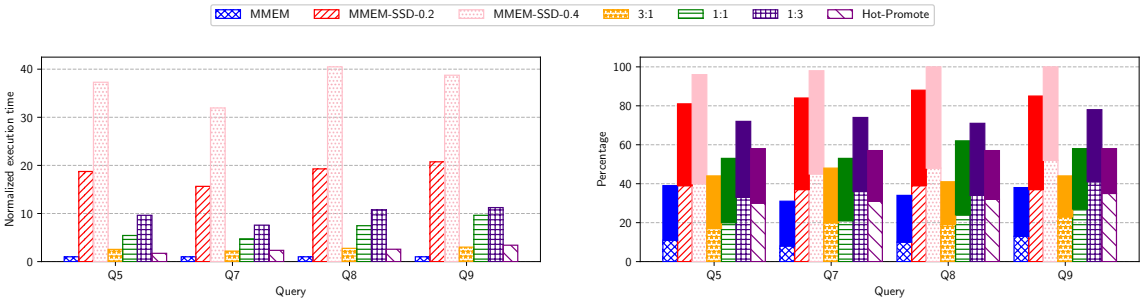
Fig. 4.6: **Spark memory layout and shuffle spill.** Each Spark executor possesses a fixed-size On-Heap memory, which is dynamically divided between execution and storage memory. If there is insufficient memory during shuffle operations, the Spark executor will spill the data to the disk.

Insights.

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

4.5.2 Spark SQL

Big Data plays a crucial role in the workloads managed by data centers. Due to the scale of data involved in Big Data analytical applications, memory capacity often becomes a bottleneck to the performance [26]. Take Spark [81], one of the common Big Data platforms, as an exam-



(a) Normalized execution time of TPC-H queries

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

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

ple: A typical query requires shuffling data from multiple tables for processing in the next stage. Operations like *reduceByKey()* first partition the data according to the key and then execute reduce operators on each key. Such shuffling operation involves disk I/O and network communication between multiple nodes, posing significant overhead on the query. In some cases, the performance of shuffling could dominate the performance of the workload [307]. During the shuffling process(Fig. 4.6), memory usage could grow beyond the capacity or certain threshold (e.g. `spark.shuffle.memoryFraction`). When this happens, Spark can be configured to spill data to disk to avoid the risk of out-of-memory failure. Since disk I/O is of magnitudes slower than memory, this could significantly impact the workload's performance.

Methodology and Software Configurations.

In our experiment, we aim to test if we could reduce the number of servers needed for a specific workload with minimal effect on overall performance. Therefore, we compared the performance of Spark running TPC-H [308] on three servers without CXL memory expansion vs. on two servers but with CXL memory expansion. We assume the maximum amount of MMEM that could be used on each server is 512 GB, therefore with three servers, we have 1.5 TB MMEM and 1 TB CXL memory in total. In order to trigger data spill within the workload, we configured 150 Spark executors. Each Spark executor contains 1 core and 8 GB of memory. Therefore the total Spark application occupies 150 cores and 1.2 TB of memory. We generate a total of 7 TB TPC-H initial dataset. We continue to adhere to the configuration settings detailed in Table 4.1 as follows:

- MMEM only: We allocate 50 Spark executor and 400 GB on each of the **three** servers. In

this case there is no data spilled to disk as each executor have sufficient amount of memory.

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

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

- Hot-Promote: same as prior experiment (§4.5.1).

We chose four specific queries ($Q5$, $Q7$, $Q8$, and $Q9$) from the TPC-H benchmark [308], recognized for their intensive data shuffling demands from prior studies [307], to evaluate our setup. Importantly, our measurements focused solely on the time to execute these queries, excluding any data preparation or server setup durations. We disabled SNC on all servers.

Analysis.

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

Year	CPU	Max vCPU per server	Memory channels per socket	Max memory \TB	Required Memory (1 : 4) \TB
2021	IceLake-SP [309]	160	8xDDR4-3200	4	0.64
2022 (delayed)	Sapphire Rapids [310]	192	8xDDR5-4800	4	0.768
2023 (delayed)	Emerald Rapids [311]	256	8xDDR5-6400	4	1
2024+	Sierra Forest [312]	1152	12	4	4.5
2025+	Clearwater Forest [313]	1152	TBD	4	4.5

Table 4.2: **Intel Processor Series.**

A notable difference between the KeyDB and Spark experiments is the performance of Hot-Promote. While it performs better in KeyDB, the Spark SQL experiment shows a more than 34% slowdown compared to MMEM. Unlike the Zipfian distribution in which the hottest keys are moved from CXL to DDR, there is a considerable amount of thrashing behavior within the kernel in the Spark SQL tests. We identify the root cause after thoroughly investigating the kernel patch implementation. In the initial version of the hot page selection patch [**?**], a sysctl knob "`kernel.numa_balancing_promote_rate_limit_MBps`" is used to control the maximum promoting/demoting throughput. Subsequent versions introduced an automatic threshold adjustment feature to this patch, aiming to strike a balance between the speed of promotion and migration costs. Nevertheless, this automatic adjustment mechanism appears to fall short in our Spark SQL evaluations. The TPC-H workload on Spark, which demonstrates reduced data locality, challenges the kernel's efficiency in promoting frequently accessed pages. This finding aligns with similar issues highlighted in prior research [275].

Insights.

Our research indicates that utilizing CXL memory expansion offers a cost-efficient approach for data-center applications. We postpone our detailed theoretical examination of the Abstract Cost Model to §4.7. Concurrently, although the hot-promote patch demonstrates significant advantages in key-value store workloads, its performance is notably lacking in Spark experiments. As system developers begin to enhance software support for CXL within the kernel, it is crucial to proceed with caution. System-wide policies can have varied impacts on applications, depending on their unique characteristics.

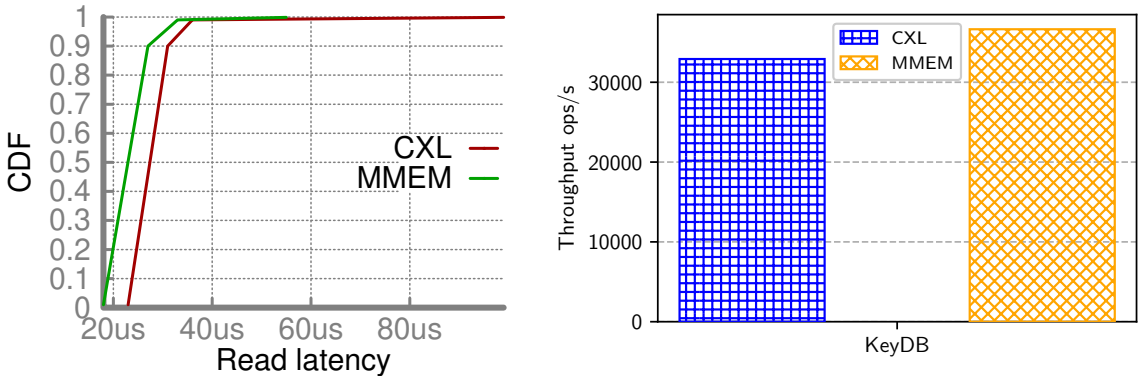
4.5.3 Spare Cores for Virtual Machine

One widely-used application within Infrastructure-as-a-Service (IAAS) is Elastic Computing [314]. Here, cloud service providers (CSPs) offer computational resources to users through virtual machines or container instances. Given the diverse needs of users, CSPs traditionally offer a variety of instance types, each characterized by different configurations of CPU cores, memory, disk, and network capacities. Generally, an "optimal" CPU-to-memory ratio, often cited as 1:4, is employed to balance computational and memory requirements (as per AWS guidelines [315,316]). For example, an instance with 128 vCPUs would typically feature 512 GB of DDR memory. Advancements in server processor architecture and chiplet technology have spurred rapid increases in the number of cores available in a single processor package, driven in large part by the CSPs' aim to lower per-core costs. Consequently, 2-socket servers have seen their vCPU counts grow from 160 to 256 within the past two years (Table 4.2). This trend is projected to continue, reaching as many as 1152 vCPUs per server by 2025.

The surge in vCPUs exacerbates memory capacity bottlenecks, constrained by DDR slot limits, DRAM density, and the cost of high-density DIMMs. Intel's Sierra Forest Xeon, for example, supports 1152 vCPUs but is limited by motherboard design to less than 4 TB of memory, falling short of the typical 4.5 TB needed for VM provisioning [317]. This discrepancy makes maintaining a cost-effective vCPU-to-memory ratio challenging, resulting in underutilized vCPUs and lost revenue for CSPs. CXL memory expansion provides a solution by enabling memory capacity to scale beyond DDR limitations, ensuring optimal vCPU utilization and mitigating revenue losses for CSPs.

Methodology and Software Configurations.

To assess the performance impact when an application operates exclusively on CXL memory, we replicate the KeyDB configuration from previous experiments (§4.5.1). We utilize *numactl* to allocate the KeyDB instance exclusively to MMEM or CXL memory. For our evaluation, the workload employed is YCSB-C, characterized by 1 KB key-value pairs and a total dataset size of 100 GB. SNC is disabled.



(a) CDF of KeyDB YCSB-C

(b) Throughput of KeyDB YCSB-C

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

Analysis.

The CDF of read latency (Fig. 4.8(a)) indicates that applications running on CXL experience a latency penalty of $9\% - 27\%$ which is less than the raw data fetching numbers in our previous measurements in §4.4. This is due to the processing latency within Redis. The throughput of running the entire workload on CXL memory is around 12.5% less compared to MMEM as show in Fig. 4.8(b).

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

Insights.

Given the sheer scale of Elastic Computing Service (ECS) applications in public clouds, the potential benefits of CXL memory expansion could be substantial. However, the challenge of maintaining an optimal virtual CPU (vCPU) to memory ratio, traditionally at 1:4, becomes more complex with the rapid increase in processor cores. This ratio, although standard, is under scrutiny for its applicability in future cloud computing paradigms. Notably, Bytedance's Volcano Engine Cloud [318] illustrates the variability in resource allocation by offering different ratios: 1:4 for general purposes,

1:2 for compute-intensive tasks, and 1:8 for memory and storage-intensive workloads. The impact of CXL memory expansion and pooling on these established ratios presents an intriguing avenue for exploration, raising questions about the adaptability of cloud providers to evolving hardware capabilities and the subsequent effect on resource allocation standards.

4.6 Memory Bandwidth-Bound applications

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

Recent work on LLM [319] shows that LLM inference is hungry for memory capacity and bandwidth. The limited capacity of GPU memory restricts the batch size of the LLM inference job and reduces computing efficiency since LLM models are memory-demanding. On the other hand, while CPU memory is high in capacity, it has lower bandwidth than GPU memory. The extra bandwidth and capacity offered by CXL memory make it a promising option for alleviating this bottleneck. For example, a CPU-based LLM inference job can benefit from the extra bandwidth brought by CXL memory, and a CXL-enabled GPU device can also use the extra memory capacity from a disaggregated memory pool. Due to the lack of CXL support in current GPU devices, we experiment with LLM inference on CPU to study the implications of CXL memory's extra bandwidth. We also note that as LLM inference applications are agnostic to the underlying memory technologies, the findings and implications from our experiments are also applicable to the upcoming CXL 2.0/3.0 devices.

LLM Inference Framework. Mainstream Large Language Model (LLM) inference frameworks, such as vLLM [320] and LightLLM [321], do not support CPU inference. Recently, Intel introduced an LLM model named Q8chat [322], trained using their 4th Generation Intel Xeon® Scalable Processors. However, the inference code for Q8chat is not yet publicly available. To address this gap, we have developed our inference framework based on the open-source LightLLM framework [321] by replacing the backend with a CPU inference backend. Figure 4.9 illustrates our implementation. In our framework, the HTTPserver frontend receives LLM inference requests and forwards the tokenized requests to a router. The router is responsible for distributing these requests to different CPU backend instances. Each CPU backend instance is equipped with a Key-Value (KV) cache [323],

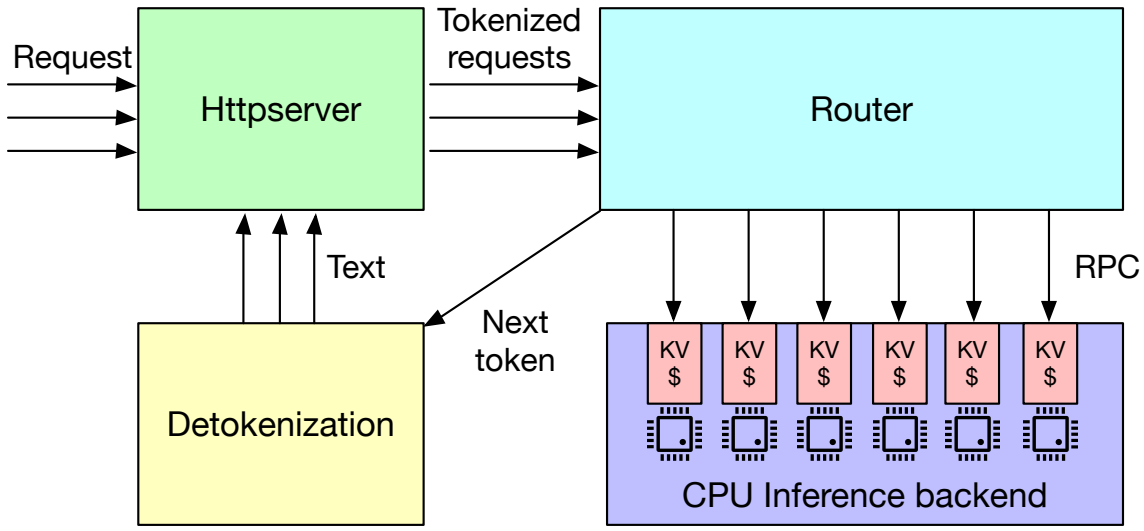
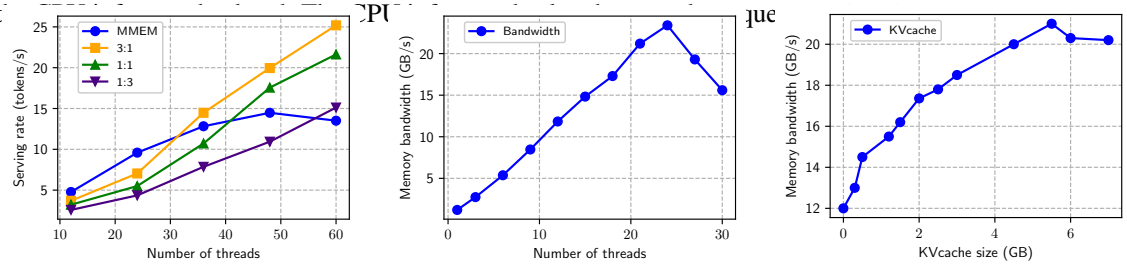


Fig. 4.9: **LLM inference framework.** The Httpserver receive requests and forward the tokenized requests to t[illegible]CPU[illegible]que



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

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

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

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

a widely used technique in large language model inference. It's worth noting that KV caching, despite its name, differs from the traditional 'key-value store' in system architecture. KV caching occurs during multiple token generation steps, specifically within the decoder. During the decoding process, the model starts with a sequence of tokens, predicts the next token, appends it to the input, and repeats this generation process. This is how models like GPT [319] generate responses. The KV cache stores key and value projections used as intermediate data within this decoding process to avoid recomputation for each token generation. Prior research [323] has shown that KV caching is typically memory-bandwidth bound, as it is unique for each sequence in the batch, and different requests typically do not share the KV cache since the sequences are stored in separate contiguous memory spaces [324].

4.6.1 Methodology and Software Configurations

To investigate the benefits of CXL memory extension for applications with high memory bandwidth demands and limited MMEM bandwidth availability, we employ the SNC-4 configuration to divide a single CPU into four sub-NUMA nodes. Each node is equipped with two DDR5-4800 memory channels, facilitating an early memory bandwidth saturation of 67 GB/s (§4.4). We examine three distinct interleaving policies (3:1, 1:1, 1:3), detailed in Table 4.1. The CPU inference backend is configured with 12 CPU threads, and memory allocation is strictly bound to a single sub-NUMA domain. This domain includes two DDR5-4800 channels and a 256 GB A1000 CXL memory expansion module via PCIe. By binding allocations to a single node, we ensure the initial saturation of the DDR5 channels. Our experiments utilize the Alpaca 7B model [325], an advancement of the LLaMA 7B model, requiring 4.1GB of memory. The workload, derived from the LightLLM framework [321], includes a wide range of chat-oriented questions. A single-threaded client machine on a baseline server sends HTTP requests with various LLM queries to mimic real-world conditions. The client ensures continuous operation of the CPU inference backends by maintaining a constant stream of requests. The prompt context is set to 2048 bytes to guarantee a minimum inference response size. We progressively increase the CPU inference backend count to monitor the LLM inference serving rate (in tokens/s).

4.6.2 Analysis

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

Interestingly, among the interleaving policies, configurations with a higher proportion of data in main memory demonstrate superior inference performance. Contrary to expectations, we observe that operating entirely on main memory is 14% less effective than a MMEM:CXL ratio of 1:3 beyond 64 threads. This outcome is notable given CXL's inherently higher latency and reduced

memory bandwidth (§ 4.4). Fig. 4.10(b) charts the memory bandwidth utilization, as measured by the Intel Performance Counter Monitor (PCM) [300], with increasing CPU thread counts within a single CPU inference backend. Initially, bandwidth utilization grows linearly with thread count, plateauing at 24.2 GB/s for 24 threads. This trend allows us to estimate a bandwidth of approximately 63 GB/s at 60 threads, reaching 82% of the theoretical maximum. Our microbenchmark findings, as detailed in §4.4, indicate that this level of bandwidth utilization may lead to significant latency spikes. These results corroborate the hypothesis that bandwidth contention plays a crucial role in the observed performance degradation.

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

4.6.3 Insights

Interestingly, existing tiered memory management in the kernel does not consider memory bandwidth contention. Considering a workload that uses high main memory bandwidth(e.g., 70%), existing page migration policy(§4.3) tends to move data from slower tiered-memory (CXL) into MMEM, supposing that there is still enough memory capacity. As more data is written into the main memory, the memory bandwidth will continue to increase (e.g., 90%). In this case, the access latency will grow exponentially, resulting in an actual slowdown of the workload. This scenario will not be uncommon, especially for memory-bandwidth-bound applications (e.g., LLM inference). Therefore, the definition of tiered memory requires rethinking.

4.7 Cost Implications

Our comprehensive analysis in prior sections (§4.5, §4.6) reveals that the adoption of CXL memory expansion offers substantial benefits for data center applications, including comparable performance

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

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

with operational cost savings. However, a significant hurdle in embracing such innovative technology as CXL lies in determining its Return on Investment (ROI). Despite having access to detailed technical specifications and benchmark performance results, accurately forecasting the Total Cost of Ownership (TCO) savings remains challenging. The complexity of simulating benchmarks at production scale, compounded by the limited availability of CXL hardware, exacerbates this issue. Traditional cost models in prior work [286], which could offer such forecasts, demand extensive internal and sensitive information that is often inaccessible. To overcome this barrier, we propose an Abstract Cost Model designed to estimate TCO savings independently of internal or sensitive data. This model leverages a select set of metrics obtainable through microbenchmarks, alongside a handful of empirical values that are simpler to approximate or access, providing a viable means to evaluate the economic viability of CXL technology implementation.

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

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

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

- Baseline performance (P_s): Measure the throughput when (almost) all working set is spilled to SSD. The absolute number is not used in our cost model. Instead, we then normalize it to 1 in our cost model.
- Relative performance when the entire working set is in MMEM (R_d): Using the same workload, we measure the throughput when the entire working set is in MMEM and normalize it to P_s to get the relative performance (i.e., how much faster compared to the baseline).
- Relative performance when the entire working set is in CXL memory (R_c): Using the same workload, we measure the throughput when the entire working set is in CXL memory, and normalize it to P_s to get the relative performance.

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

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

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

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

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

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

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

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

TCO saving can then be formulated as follows.

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

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

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

Extending Cost Model for more realistic scenarios. In line with previous research [286], our Abstract Cost Model is designed to be adaptable, allowing for the inclusion of additional practical infrastructure expenses such as the cost of CXL memory controllers, CXL switches (applicable in CXL 2.0/3.0 versions), PCBs, cables, etc., as fixed constants. However, a notable constraint of our current model is its focus on only one type of application at a time. This becomes a challenge when a data center provider seeks to evaluate cost savings for multiple distinct applications, each with unique characteristics, especially in environments where resources are shared (for instance, through CXL memory pools). This scenario introduces complexity and presents an intriguing challenge, which we acknowledge as an area for future investigation.

Chapter 5

Future Work

5.1 Introduction

Autoregressive large language models (LLMs) generate output tokens sequentially, where the generation of each token involves the attention computation using key-value (KV) of its preceding tokens [326–328]. This sequential dependency makes LLM inference both compute- and memory-intensive. LLM inference typically includes two stages: the prefill stage, where all input tokens are processed to generate the initial output token, and the decode stage, where the rest of the output tokens are generated one by one until the model generates an end-of-sequence token [329–331].

For applications such as chatbot and coding assistant, LLM serving systems aim to minimize the time to finish the prefill stage, or time to first token (TTFT). In production, service-level objective (SLO) for TTFT is typically 400ms [330]. To meet such SLO, LLM serving systems often cache the previously-computed KV data of the preceding tokens (i.e., prefix) in GPU memory, to avoid re-computing them for future requests that have the same prefix [330, 332, 333]. Storing KV cache reduces the overall computational load and significantly improves throughput by trading memory for computation.

In production chatbot applications that support large context windows, the demand for KV cache storage grows rapidly by the number of inference requests from users, which cannot be fully accommodated by the limited and expensive GPU memory [334]. Researchers thus developed techniques to offload KV cache to CPU memory, leveraging the larger CPU memory capacity to reduce GPU memory pressure [333, 335, 336]. However, as larger LLMs and support for long-context inference

requests continue to emerge, the approach of storing KV cache to CPU memory is still insufficient. For example, in LLaMA-2-7B, KV cache of token in FP32 precision is 1024KB; KV cache of a single request with 4096 tokens (maximum context length) is 4GB [337]. The memory demand from serving many concurrent long-context requests can easily overwhelm even high-end memory servers [332, 338].

Practitioners increasingly turn to more scalable memory architectures, such as Compute Express Link (CXL) memory [37,339,340], to address the growing memory demands of large-scale systems. CXL expands memory capacity by connecting additional DRAM to servers via PCIe, while maintaining low-latency access. It offers a promising solution to the KV cache storage demand in LLM serving.

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

- We present the first measurement of CXL-GPU interconnect and evaluate its feasibility for KV cache storage. We show that the data-transfer latency and bandwidth on CXL-GPU interconnect is on par with CPU-GPU interconnect.
- We present our design of CXL-based KV cache storage interface and evaluate its performance improvement to LLM serving, on our platform that is the first to successfully integrate ASIC-CXL device and GPU. Our results show competitive TTFT achieved by CXL-based prefix caching.
- We examine the cost-efficiency in using CXL for KV cache storage in production via Return on Investment (ROI) modeling. Estimates show a promising reduction in GPU compute cost when using CXL for KV cache storage. We also identify promising future research directions.

5.2 CXL-based KV Cache Storage

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

Design and implementation. Our goal is to develop a CXL storage interface, named *KVExpress*,

which can be integrated into existing LLM serving systems for saving and loading KV cache of inference requests. *KVExpress* provides two external APIs to its upper-level serving system: `save` and `load`. The `save` takes a unique identifier of a token chunk as input, and copies its KV cache from GPU to CXL memory. The `load` takes a unique identifier of a token chunk as input, and finds if its KV cache exists in CXL memory, if so, copies the KV cache from CXL memory to GPU. A token chunk can consist of one or more tokens. The unique identifier of a token chunk t_i for a sequence is the hash of the content of t_i and the hash of its prefix $\langle t_0, ..., t_{i-1} \rangle$. If the prefix of a sequence of a current request has been computed and saved into CXL, *KVExpress* will load the KV cache of the prefix from CXL and use it when computing for this request [332].

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

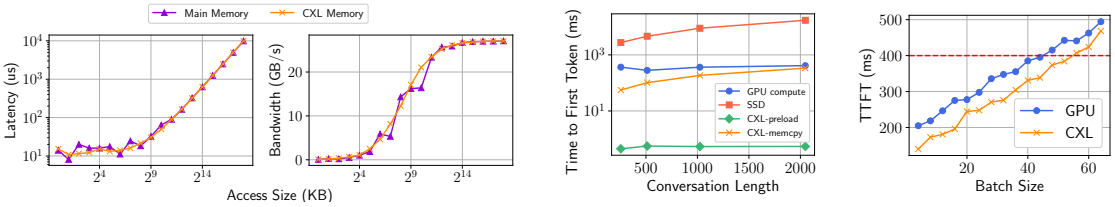
We implement our design of *KVExpress* in gpt-fast [341], a simple and low-latency text generation system with support on a number of widely-used inference optimizations [342–344] and open-source LLMs [337, 345]. We further modify gpt-fast to support our evaluation on batched inference.

Hardware platform. Our single socket server is equipped with Intel Xeon Platinum processors [346], 1TB of 4800 MHz DDR5 memory, an NVIDIA H100 GPU with 96GB HBM, and a CXL memory expansion card with 256 GB of DDR5 memory at 4800 MHz [340]. While prior works [347–349] have explored utilizing CXL for accelerators, to our knowledge, our work is the first implementation to successfully integrate a real ASIC-CXL device and a GPU within a single inference server.

5.3 Performance Evaluation

In Section 5.3.1, we measure the latency and bandwidth of CXL-GPU interconnect for data transfer to assess the feasibility of storing KV cache on CXL devices. In Section 5.3.2, we compare the TTFT of KV re-compute, prefix caching with CXL, and prefix caching with GPU, to understand if *KVExpress* can achieve similar TTFT as existing approaches for prefill requests under varying context lengths. In Section 5.3.3, we study the maximum batch size achieved while retaining a

given SLO on TTFT between KV re-compute and prefix caching with CXL.



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

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

5.3.1 Measurements on CXL-GPU interconnect performance

KV cache storage requires low-latency access (e.g., from host memory to GPU memory). Although prior studies [339,340] show that accessing CXL memory from the host CPU is over 2× slower than accessing local memory, none of their measurements involves any interaction with the GPU. In this paper, we evaluate the performance characteristics of the CXL-GPU interconnect by measuring the latency and bandwidth of copying data from CXL memory to the GPU. Transferring in the reverse direction yields similar results [350]. Since CXL memory devices are exposed to the system as NUMA nodes without CPUs by default [340], we allocate a set of host buffers on the CXL NUMA node and use `cudaMemcpyAsync` to copy data between the host buffers and GPU device buffers allocated via the CUDA API [351]. We evaluated transferring data of sizes ranging from 1KB to 256MB.

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

5.3.2 Evaluation on TTFT under varying input context length

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

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

We measure the TTFT of the aforementioned approaches on conversation requests of input length ranging from 256 to 2048 tokens from the ShareGPT-Vicuna-Unfiltered dataset [352]. We use the LLaMA-2-13B as the underlying model for our evaluation. Figure 5.1(b) shows the TTFT (y-axis in log-scale) achieved by the evaluated approaches for requests of varying input context length (x-axis).

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

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

5.3.3 Evaluation on serving throughput while adhering SLO

By storing the KV cache of the inference request prefix in CXL memory and thus reducing re-computation during the prefill stage, we can effectively reduce the computational load on the GPU. The saved GPU compute can be re-allocated to handle a larger number of concurrent inference requests. In other words, the LLM serving system can achieve a higher serving throughput, by handling a larger batch size of inference requests using the saved GPU compute, while maintaining the same SLO on TTFT [330].

Figure 5.1(c) shows the TTFT achieved by KV re-compute and prefix caching with CXL under varying batch size. The horizontal red-dashed line indicates our SLO limit–the maximum TTFT that can be tolerant in production. The typical SLO is 400ms used for LLaMA-2 [353]. As shown in Figure 5.1(c), with KV re-compute, the evaluated serving system (§5.2) can handle a maximum batch size of 44 before hitting the SLO limit. On the other hand, when leveraging CXL for storing KV cache, the system can handle a maximum batch size of 57, which is a **30% increase** compared to KV re-compute. Our initial evaluation on SLO-adhering serving throughput highlights the performance benefits of utilizing CXL memory for KV cache storage, particularly in scenarios that require efficient scaling under strict latency requirements.

5.4 Cost-Efficiency Modeling

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

By replacing computation with memory accesses, we reduce the overall computational load, thereby lowering the demand for FLOP/s while still meeting the same SLO. This results in significant cost savings for LLM inference (Figure 5.2):

- **Assumption:** Assume a GPU has a com-

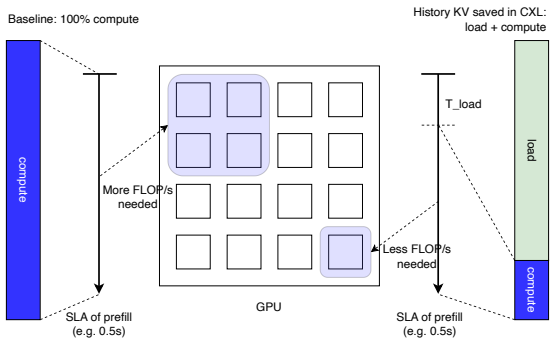


Fig. 5.2: Example of ROI modeling: replace computation with memory access

putational power of 100 TFLOP/s, an average prefill request requires 25 TFLOP of computation, and the SLO for prefill is 0.5 seconds.

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

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

5.5 Conclusion and Future Work

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

Table 5.1: ROI Modeling

C_0	Avg. FLOPs needed by a prefill request in an initial request. Can be estimated as $C_0 = 2ML$, where M is the model parameters and L is the avg. sequence length.
C_1	FLOPs needed by new prompt in a follow-up request. Can be estimated as $C_1 = rC_0$, where r is the avg. ratio of the new prompt (e.g., 10%).
T_{slo}	SLO of prefill (e.g., 0.5s).
T_{load}	Avg. time to load KV cache from memory (e.g., 0.1s).
P	Computation power (FLOP/s) of the GPU.
P_0	FLOP/s needed for the initial request. $P_0 = C_0/T_{slo}$
P_1	FLOP/s needed for the new prompt. $P_1 = C_1/(T_{slo} - T_{load})$
R_{gpu}	Request per second (RPS) a single GPU can support. $R_{gpu} = P/(P_0(1-h) + P_1h)$, where h is the ratio of multi-round requests.
N_{cxl}	Number of GPUs needed using our CXL memory scheme. $N_{cxl} = \lceil R/R_{gpu} \rceil = \lceil \frac{R}{P/(P_0(1-h)+P_1h)} \rceil$
$N_{baseline}$	Number of GPUs needed without any KV cache stored (i.e., all data discarded after prefill). $N_{baseline} = \lceil \frac{R}{P/P_0} \rceil$

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

Appendix A

Appendix

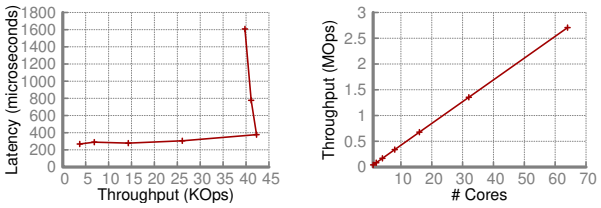
A.1 Jiffy: Additional Evaluation

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

A.1.1 Controller Performance

Jiffy adds several components at the controller compared to Pocket, including all of metadata management, lease management and handling requests for data repartitioning. As such, we expect its performance to be lower than Pocket's metadata server. We deem this to be acceptable as long as it can still handle control plane request rates typically seen for real world workload, *e.g.*, a peak of a few hundred requests per second, including lease renewal requests, for all of our evaluated workloads and those evaluated in [39].

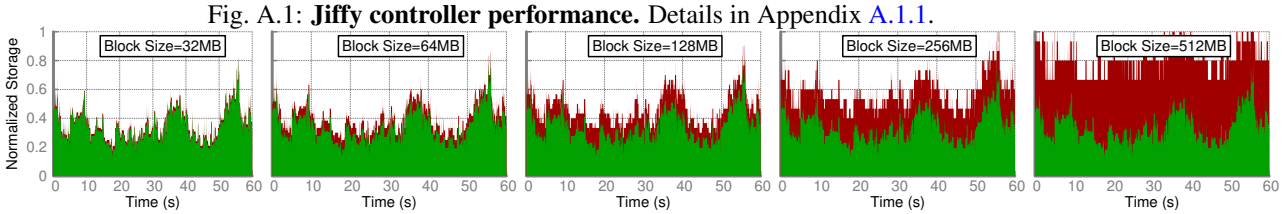
Figure A.1(a) shows the throughput-vs-latency curve for Jiffy controller operations on a single CPU core of an m4.16xlarge EC2 instance. The controller throughput saturates at roughly 42 KOps, with a latency of 370us. While this throughput is lower than Pocket (~ 90KOps per core), it is more than sufficient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



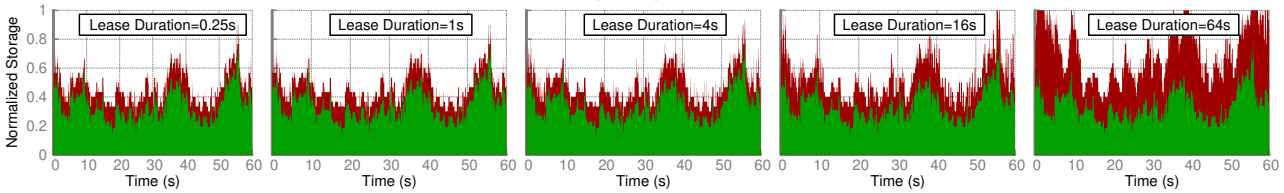
(a) Controller throughput vs. latency on a single CPU core.

(b) Controller throughput scaling with multiple cores.

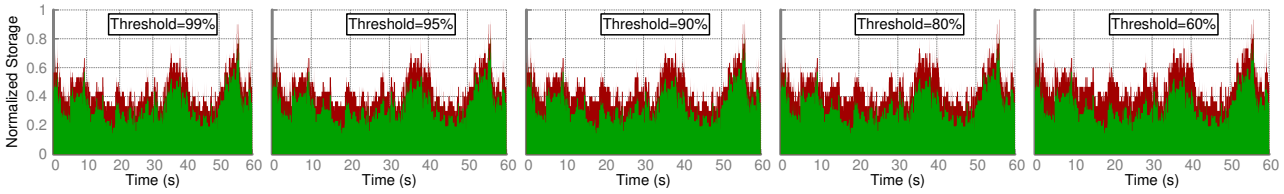
Fig. A.1: **Jiffy controller performance.** Details in Appendix A.1.1.



(a) Sensitivity analysis for block size



(b) Sensitivity analysis for lease duration



(c) Sensitivity analysis for (high) repartition threshold

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

A.1.2 Sensitivity Analysis

We now perform sensitivity analysis for various system parameters in Jiffy, including block size (§2.2.1), lease duration (§2.2.2) and thresholds for data repartitioning (§2.2.3). We use files as our underlying data structure, and use the Snowflake workload from Figure 2.2. These results can be contrasted directly with Figure 2.11(a) (center), which corresponds to our default system parameters (128MB blocks, 1s lease duration and 95% of block occupancy as repartition threshold). For each parameter that we vary, the other to remain fixed at their default values.

Block size (Figure A.2(a)). As discussed in §2.2.1, the block size in Jiffy exposes a tradeoff between the amount of metadata that needs to be stored at the control plane, and resource utilization.

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

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

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

Supplementary Materials

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

We claimed in §3.12.1 that if $t_c = \eta \cdot t_d$ for all offloaded iterator executions, it is always possible to multiplex $m + n$ concurrent iterator executions and fully utilize all memory and logic pipelines. We prove our claim by providing a staggered scheduling algorithm (Algorithm 1) that ensures such multiplexing across $m + n$ iterator executions. The scheduler processes $m + n$ iterator execution requests, assigning each a memory pipeline, a logic pipeline, and staggered start times. These requests are then executed in the respective memory pipelines. Through this staggered scheduling approach, Jiffy fully utilizes the n memory pipelines and m logic pipelines, ensuring no resources are wasted. Note that this algorithm is a simplified version to illustrate the potential for full pipeline saturation under the given condition. Jiffy's scheduler implements a real-time algorithm to multiplex incoming requests on the fly.

Algorithm 1 Staggered-Scheduling

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

A.1 PULSE Empirical Analysis

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

B. PULSE Supported Data Structures

We adapt 13 data structures across 4 popular open-sourced libraries to PULSE's iterator abstraction (§3.12). In particular, we outline how the data structure implementations for certain operations can be expressed using `init()`, `next()`, and `end()`. For simplicity and readability, (i) we assume that the data structure developer defines a macro, `SP_PTR_(variable_name)`, as the address of the variable resides on the `scratch_pad`, and (ii) we omit obvious type conversions for de-referenced pointers.

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

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

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

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

Application	% of time spent in pointer traversal
GraphChi [135]	$\sim 93\%$
MonetDB [235]	$70\% - 97\%$
GC in Spark [81]	$\sim 72\%$
VoltDB [236]	Up to 49.55%
MemC3 [237]	Up to 21.15%
DBx1000 [238]	$\sim 9\%$
Memcached [138]	$\sim 7\%$

(a) Survey from prior studies

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

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

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

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

B.1 List data structure in STL library

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

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

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

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

```
class list_find : chase_iterator {

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

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

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

B.2 List data structure in Boost library

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

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

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

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

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

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

    bool end() {
```

B.3 Tree data structure in Google library

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

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

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

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

```

B.4 Tree data structure in STL library

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

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

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

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

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

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

B.5 Tree data structure in Boost library

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

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

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

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

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

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

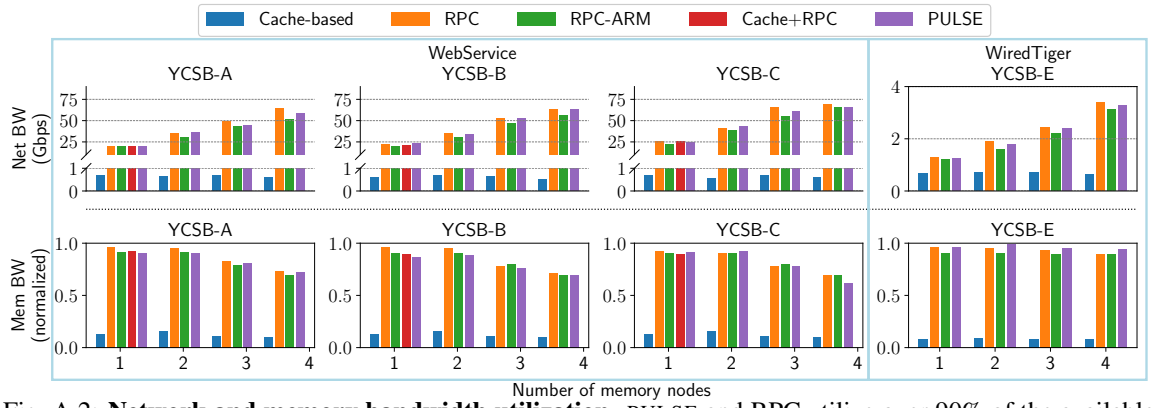


Fig. A.2: **Network and memory bandwidth utilization.** PULSE and RPC utilize over 90% of the available memory bandwidth, while the cache-based approach suffers from swap system overhead. In Webservice, the network bandwidth becomes the bottleneck due to large 8 KB data transfers.

C. PULSE Additional Evaluation Results

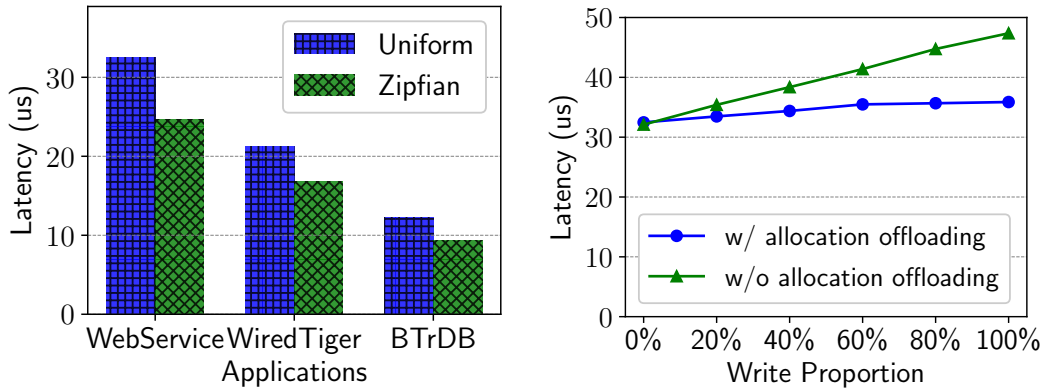
In this section, we provide additional evaluation results for PULSE.

C.1 Traditional Core Architecture vs. PULSE

We evaluate the impact of the PULSE architectural design (§3.12.1) by comparing PULSE against PULSE-CORE, an in-order processor built on PULSE's components. We denote *C_x* as in tightly-coupled core architecture, where *x* is the number of cores. We denote *P_x_y* as PULSE architecture, where *x* is the number of logic pipelines and *y* is the number of memory pipelines. Table A.2 shows the power, performance, and area usage of various configurations. The performance metrics are obtained by executing the WebService application with various configurations. In PULSE's disaggregated architecture, when the number of logic and memory pipelines is equal to that of a traditional core architecture, power and area usage are higher due to additional logic and buffering in the interconnect and scheduler. However, due to the nature of pointer traversal operations (§3.12.1), PULSE requires fewer logic pipelines to achieve similar performance. For example, to fully saturate the memory bandwidth of a single node, PULSE uses only one logic pipeline and four memory pipelines, while a traditional core architecture requires four cores. As a result, PULSE saves 20.12% in power with only a 7.2% latency overhead, primarily due to the additional scheduler and data movement between workspaces.

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

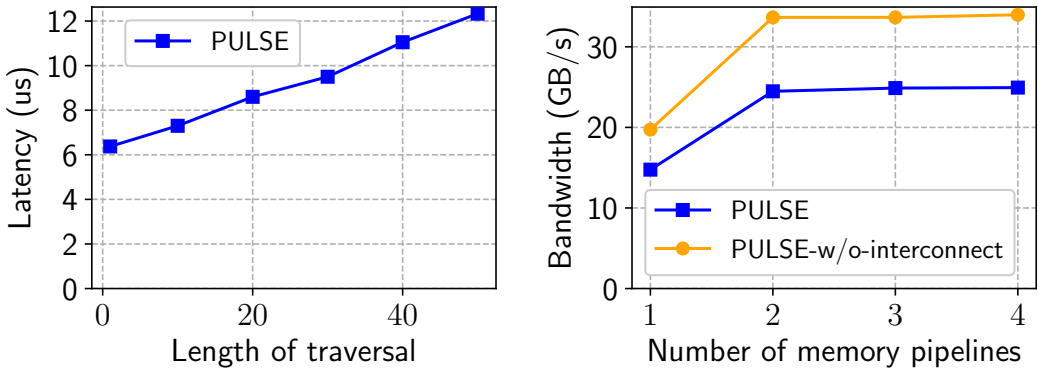
Table A.2: Comparison between traditional core architecture and PULSE architecture.



(a) Impact of access pattern

(b) Impact of modifications

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



(a) Traversal length

(b) Number of memory pipelines

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

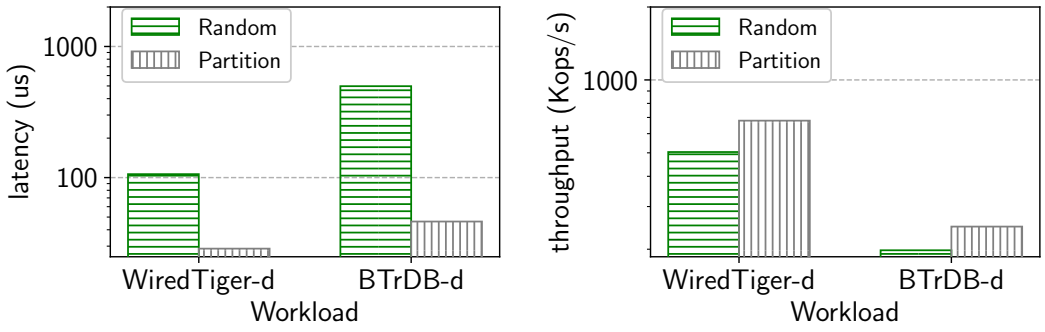
C.2 Network and Memory Bandwidth Utilization

We evaluate the network and memory bandwidth utilization of the three applications in Fig. A.2. For WiredTiger, PULSE and RPC utilize over 90% of the available memory bandwidth, while the Cache-based approach suffers from low network bandwidth and memory utilization due to swap system overhead. For WebService, the large 8 KB data transfers saturate the maximum bandwidth that the DPDK stack can sustain [260]. As a result, network bandwidth becomes the bottleneck, reducing PULSE and RPC memory bandwidth utilization under 3 and 4 memory nodes. The memory bandwidth is normalized, where 1.0 corresponds to 25 GB/s per node.

C.3 PULSE Sensitivity Analysis

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

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



(a) Latency (b) Throughput

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

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

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

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

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

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

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

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] 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.

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

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

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

[39] 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.

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

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

[42] 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.

[43] 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.

[44] 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.

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

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

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

[48] State of the Serverless Community Survey Results. https://serverless.com/blog/state-of-serverless-community.

[49] 2018 Serverless Community Survey: huge growth in serverless usage. https://bit.ly/2Mu5TCR.

[50] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[63] K. Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[64] B. Fleisch and G. Popek. *Mirage: A coherent distributed shared memory design*, volume 23. ACM, 1989.

[65] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):109–133, 1988.

[66] P. Dasgupta, R. J. LeBlanc, M. Ahamad, and U. Ramachandran. The clouds distributed operating system. *Computer*, 24(11):34–44, 1991.

[67] J. B. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: Where we are and where we should be headed. In *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 119–122. IEEE, 1995.

[68] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*.

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

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

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

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

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

[74] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management (IWMM)*, 1992.

[75] D. I. Bevan. Distributed garbage collection using reference counting. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 1987.

[76] K. G. Cassidy. Feasibility of automatic storage reclamation with concurrent program execution in a lisp environment. master's thesis. 1985.

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

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

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

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

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

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

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

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

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

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

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

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

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

[90] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote Memory in the Age of Fast Networks. In *SoCC*, 2017.

[91] A. Carbonari and I. Beschasnikh. Tolerating Faults in Disaggregated Datacenters. In *HotNets*, 2017.

[92] High Throughput Computing Data Center Architecture. http://www.huawei.com/ilink/en/download/HW_349607.

[93] The Machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/.

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

[95] Facebook's Disaggregated Racks Strategy Provides an Early Glimpse into Next Gen Cloud Computing Data Center Infrastructures. https://dcig.com/2015/01/facebooks-disaggregated-racks-strategy-provides-early-glimpse-next-gen-cloud-computing.html.

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

[97] In Bid for Major Carriers and Service Providers, Dell EMC Rack Scale Infrastructure Offers 'Hyperscale Principles'. https://www.enterpriseai.news/2017/09/12/bid-major-carriers-service-providers-dell-emc-rack-scale-infrastructure-offers-hyperscale-principles/.

[98] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network Support for Resource Disaggregation in Next-Generation Datacenters. In *HotNets*, 2013.

[99] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *ATC*, 2015.

[100] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*, 2014.

[101] 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.

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

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

[104] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR*, 2014.

[105] R. S. Fabry. Capability-based addressing. *CACM*, 1974.

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

[107] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[108] P4. https://p4.org/.

[109] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu. DC.P4: Programming the Forwarding Plane of a Data-Center Switch. In *SOSR*, 2015.

[110] 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.

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

[112] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 2007.

[113] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, 2010.

[114] R. A. Maddox, R. J. Safranek, and G. Singh. *Weaving High Performance Multiprocessor Fabric: Architectural Insights Into the Intel QuickPath Interconnect*. Intel Press, 2009.

[115] I. Corporation. An Introduction to the Intel QuickPath Interconnect. White paper, 2009.

[116] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *PPOPP*, 1990.

[117] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. Technical report, 1993.

[118] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *SoCC*, pages 323–337, 2017.

[119] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 1992.

[120] MESI Protocol. https://en.wikipedia.org/wiki/MESI_protocol.

[121] MESIF Protocol. https://en.wikipedia.org/wiki/MESIF_protocol.

[122] MOESI Protocol. https://en.wikipedia.org/wiki/MOESI_protocol.

[123] MOSI Protocol. https://en.wikipedia.org/wiki/MOSI_protocol.

[124] A. Bhattacharjee, D. Lustig, and M. Martonosi. *Architectural and Operating System Support for Virtual Memory*. Synthesis Lectures on Computer Architecture. 2017.

[125] M. Parasar, A. Bhattacharjee, and T. Krishna. SEESAW: Using Superpages to Improve VIPT Caches. In *ISCA*, 2018.

[126] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Energy-Efficient Address Translation. In *HPCA*, 2016.

[127] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *ISCA*, 2014.

[128] Boot Memory Allocator. https://www.kernel.org/doc/gorman/html/understand/understand022.html.

[129] Page Migrations. https://www.kernel.org/doc/html/latest/vm/page_migration.html.

[130] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. Range Translations for Fast Virtual Memory. *IEEE Micro*, 2016.

[131] The GNU Allocator. https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html.

[132] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[133] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[134] A. Krizhevsky, V. Nair, and G. Hinton. CIFAR-10 (Canadian Institute for Advanced Research).

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

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

[137] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA, 2010. Association for Computing Machinery.

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

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

[140] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.

[141] R. K. Jain, D.-M. W. Chiu, W. R. Hawe, et al. A Quantitative Measure of Fairness and Discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[142] D. R. K. Ports and J. Nelson. When Should The Network Be The Computer? In *HotOS*, 2019.

[143] 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.

[144] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *NSDI*, 2017.

[145] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*, 2018.

[146] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *SOSR*, 2016.

[147] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *SIGCOMM*, 2017.

[148] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. In *SOSR*, 2015.

[149] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking*, 2020.

[150] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*, 2016.

[151] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *NSDI*, 2015.

[152] T. Jepsen, L. P. de Sousa, M. Moshref, F. Pedone, and R. Soulé. Infinite Resources for Optimistic Concurrency Control. In *NetCompute*, 2018.

[153] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *SOSP*, 2017.

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

[155] Mellanox Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). https://www.mellanox.com/products/sharp.

[156] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *HotNets*, 2017.

[157] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation, 2020.

[158] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *SIGCOMM*, 2018.

[159] A. Lerner, R. Hussein, and P. Cudré-Mauroux. The Case for Network Accelerated Query Processing. In *CIDR*, 2019.

[160] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*, 2017.

[161] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *ASPLOS*, 2017.

[162] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *OSDI*, 2020.

[163] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *FAST*, 2021.

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

[165] S. Hong, W.-O. Kwon, and M.-H. Oh. Hardware implementation and analysis of gen-z protocol for memory-centric architecture. *IEEE Access*, 8:127244–127253, 2020.

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

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

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

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

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

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

[172] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi. Co-architecting controllers and dram to enhance dram process scaling. In *The memory forum*, volume 14, 2014.

[173] S.-H. Lee. Technology scaling challenges and opportunities of memory devices. In *International Electron Devices Meeting (IEDM)*, 2016.

[174] S. Shiratake. Scaling and performance challenges of future dram. In *International Memory Workshop (IMW)*, 2020.

[175] C. Reiss. *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, EECS Department, University of California, Berkeley, 2016.

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

[177] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37. IEEE, 2014.

[178] V. Young, C. Chou, A. Jaleel, and M. Qureshi. Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 328–339. IEEE, 2018.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[220] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO-46, page 468–479, New York, NY, USA, 2013. Association for Computing Machinery.

[221] 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.

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

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

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

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

[226] 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.

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

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

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

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

[231] 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.

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

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

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

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

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

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

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

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

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

[241] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In *21st USENIX Conference on File and*

Storage Technologies (FAST 23), pages 99–114, Santa Clara, CA, February 2023. USENIX Association.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[261] Intel Xeon Gold 6240 Processor datasheet. https://ark.intel.com/content/www/us/en/ark/products/192443/intel-xeon-gold-6240-processor-24-75m-cache-2-60-ghz.html.

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

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

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

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

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

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

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

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

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

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

[272] 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.

[273] 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.

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

[275] 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.

[276] 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.

[277] 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.

[278] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[310] 4th Gen Intel Xeon Processor Scalable Family, sapphire rapids. (n.d.). Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html#gs.3m5uv2.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

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

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

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

[336] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

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

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

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

[340] 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.

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

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

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

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

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

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

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

[348] S. Sano, Y. Bando, K. Hiwada, H. Kajihara, T. Suzuki, Y. Nakanishi, D. Taki, A. Kaneko, and T. Shiozawa. Gpu graph processing on cxl-based microsecond-latency external memory. In

Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, page 962–972, New York, NY, USA, 2023. Association for Computing Machinery.

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

[350] 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.

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

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

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