MIND: In-Network Memory Management for Disaggregated Data Centers

Seung-seob Lee, Yanpeng Yu, Yupeng Tang,
Anurag Khandelwal, Lin Zhong, Abhishek Bhattacharjee
Yale University

Abstract

Memory disaggregation promises transparent elasticity, high resource utilization and hardware heterogeneity in data centers by physically separating memory and compute into network-attached resource "blades". However, existing designs achieve performance at the cost of resource elasticity, restricting memory sharing to a single compute blade to avoid costly memory coherence traffic over the network.

In this work, we show that emerging programmable network switches can enable an efficient shared memory abstraction for disaggregated architectures by placing memory management logic *in the network fabric*. We find that centralizing memory management in the network permits bandwidth and latency-efficient realization of in-network cache coherence protocols, while programmable switch ASICs support other memory management logic at line-rate. We realize these insights into MIND[1], an in-network memory management unit for rack-scale disaggregation. MIND enables transparent resource elasticity while matching the performance of prior memory disaggregation proposals for real-world workloads.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Networks → Programmable networks**.

Keywords: Memory disaggregation, Programmable networks

1 Introduction

Data center network bandwidth is approaching that of intra-server resource interconnects [1, 2], and is soon poised to surpass it [3]. This has driven significant academic [4–14] and industry [15–20] 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 [10–13] 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 (§2.2). Most approaches require application modifications due to changes in hardware [6, 16, 17, 21], programming model [22, 23], or memory interface [24–26]. Recent approaches that enable transparent access to disaggregated memory [10–12] 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 [27–29] 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 [27], 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

[1] **M**MU **I**n-**N**etwork for **D**isaggregated architectures

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483561

ASICs [30] 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 (§4.1).
- MIND features domain-based memory protection inspired by capability-based schemes [31–33] 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 (§4.2).
- MIND adapts directory-based MSI coherence [34] 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 (§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 (§4.3). We address this through a novel Bounded Splitting algorithm (§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 (§7).

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 [35], 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 §8.

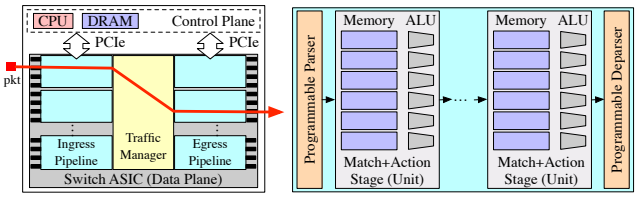


Fig. 1. Enabling technologies for MIND. (left) Programmable switch architecture and (right) Switch ingress/egress pipeline.

Table 1. In-network technology tradeoffs. See §2.1 for details.

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

2 Background

This section motivates MIND. We discuss key enabling technologies (§2.1), followed by challenges in realizing memory disaggregation goals using existing designs (§2.2).

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 [4–10, 13], we restrict our scope to *partial* memory disaggregation: while most of the memory is network-attached, CPU blades possess a small amount (few GBs) of local DRAM as cache.

2.1 Enabling Technologies

We now briefly describe MIND's enabling technologies.

Programmable switches. In recent years, programmable switches have evolved along two well-coordinated directions: development of P4 [30, 36, 37], a flexible programming language for network switches, and design of switch hardware that can be programmed with it [28, 29, 38, 39]. These switches host an application-specific integrated circuit (ASIC), along with a general purpose CPU with DRAM, as shown in Figure 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 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) [38], it is possible to realize MIND using FPGAs, custom ASICs, or even general

purpose CPUs. While each of them exposes different tradeoffs (Table 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 [40–44] and distributed shared memory (DSM) architectures [35, 45–48]. 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 [34, 49–52] are required to ensure each node operates on the latest version of a block. In popular directory-based invalidation protocols like MSI [34] (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 §8.

2.2 Disaggregated Memory Designs and Challenges

As outlined in §1, 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 §9.

Table 2. Parallels between memory & networking primitives.

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

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 §2.1 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 [9–11, 13, 15, 16], i.e., while compute blades cache data locally, different compute blades do not share memory to avoid sending coherence messages over the network. As such, these proposals achieve memory performance only by limiting transparent compute elasticity for an application to the resources available on a single compute blade, requiring application modifications if they wish to scale beyond a compute blade.

3 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 [27–29] 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 2), allowing us to leverage decades of innovation in network hardware and protocol design for disaggregated memory

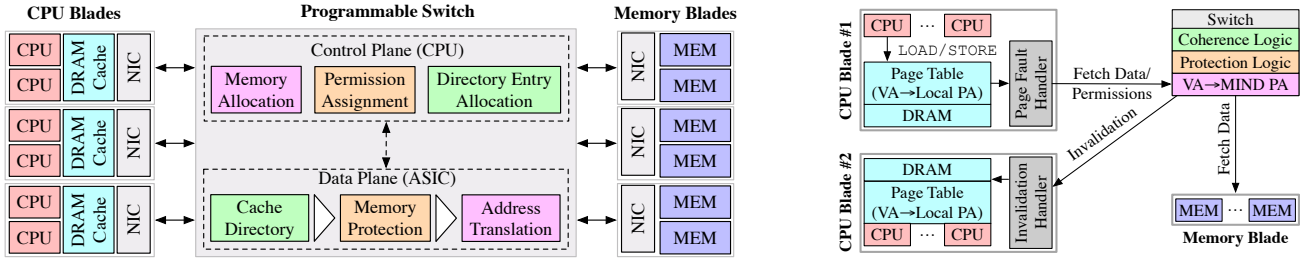


Fig. 2. (left) High-level MIND architecture, and, (right) data flow for memory accesses in MIND. See §3.2 for details.

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 §4.

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.

3.1 Design Principles

MIND follows three principles to meet the goals for memory disaggregation outlined in §1:

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 (§4.1), address translation from memory protection (§4.2), cache accesses and eviction from coherence protocol execution (§4.3.2), 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 [53–55].

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 [11, 12], or a separate global controller [10].

Figure 2 (left) provides an overview of MIND design, while Figure 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 (§4.3). The cache is virtually addressed[2], 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 (§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 (§4.3, §5). In parallel, the data plane also ensures the requesting process has permissions to access the requested page (§4.2). If no CPU blade cache holds the page, the data plane translates the virtual addresses to physical addresses (§4.1), forwarding the request to the appropriate memory

[2]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 2 (right)).

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 [10], MIND leverages one-sided RDMA operations [24] 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.

4 In-Network Memory Management

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 (§4.1), memory protection (§4.2), and cache coherence (§4.3), while operating under the constraints of programmable switches. Finally, we detail how MIND handles failures (§4.4).

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 (§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 [23, 32, 33, 35, 56], 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 §7.

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 [57]. 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 §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 §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 [58], i.e., moving pages from one memory blade to another. MIND maintains separate *range-based* address translations [59] 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.

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 (§7). This is because the first-fit allocator and Linux's `glibc` allocation requests [60] 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 [31, 56], 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> → 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 [56].

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.

4.3 Caching & Cache Coherence

In MIND design, while the caches[3] 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 §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.

[3]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.

4.3.1 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 §7.3.

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 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 [10–12]). 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 §5.

4.3.2 In-Network Coherence Protocol

Due to the limited computational capability at the switch ASIC, MIND employs the simple directory-based MSI coherence protocol [34]. While we defer the implementation details of the coherence protocol to §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.

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 [10, 11, 14] 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.

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 §4.3.

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 §7. 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 §7 that appropriate epoch sizing allows us to ensure both assumptions for our evaluated workloads.

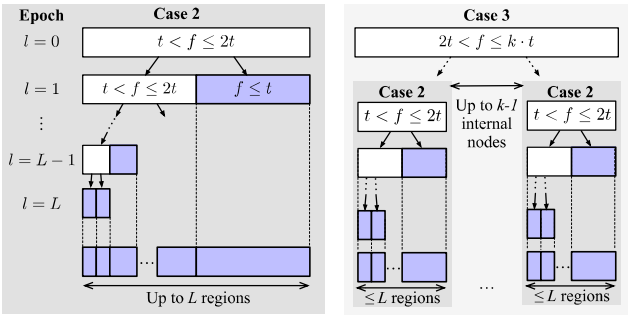
5.2 Performance Bounds

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.

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

Theorem 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



(a) (b)

Fig. 3. Splitting process for cache blocks depicted as a binary tree. Note that $L = \log_2 M$; see §5.2 for details.

$\log_2 M$ times, across as many epochs. Figure 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 = 2MB. 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 (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 (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.,

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

where c is a constant parameter.

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

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

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 §7.

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 §7. 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%.

6 Implementation Details

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.

6.1 CPU Blade

MIND assumes a partial disaggregation model, where the CPU blades possess a small amount of local DRAM as cache (§2.1). 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 [10], 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 [35]. 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 §7.1. We discuss possible architectural changes to address this in §8.

6.2 Memory blade

Unlike prior disaggregated memory systems [10, 11] or distributed shared memory systems [35], 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.

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 (§4.1) and memory protection (§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 §4.

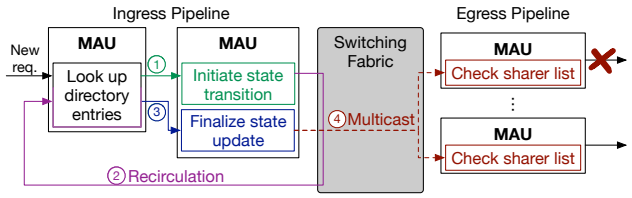


Fig. 4. Performing directory state transitions on switch ASIC.

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 accordingly all at once, due to their limited compute capability. As such, we split the logic for (i-ii) across two MAUs (① in Figure 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 §4.3.2. 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.

7 Evaluation

We evaluate MIND to answer the following questions:

- Can MIND enable transparent elasticity over performant disaggregated memory for real-world workloads (§7.1)?
- What are MIND's performance/resource bottlenecks (§7.2)?
- How does bounded splitting perform in isolation (§7.3)?

Compared systems. We compare MIND against two extreme designs in disaggregated memory (§2.2): 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 [35], a software-based DSM, to our disaggregated setting where cache directory is implemented at the compute blades. For the latter, we employ FastSwap [12], 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 §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 [10], 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: TensorFlow [61] with ResNet-50 [62] on CIFAR-10 [63] (denoted as TF), GraphChi [64] with PageRank [65] on Twitter graph [66] (denoted as GC), and Memcached [67] with YCSB [68] workload A (50% read, 50% write split, denoted as M_A) and workload C (100% reads,

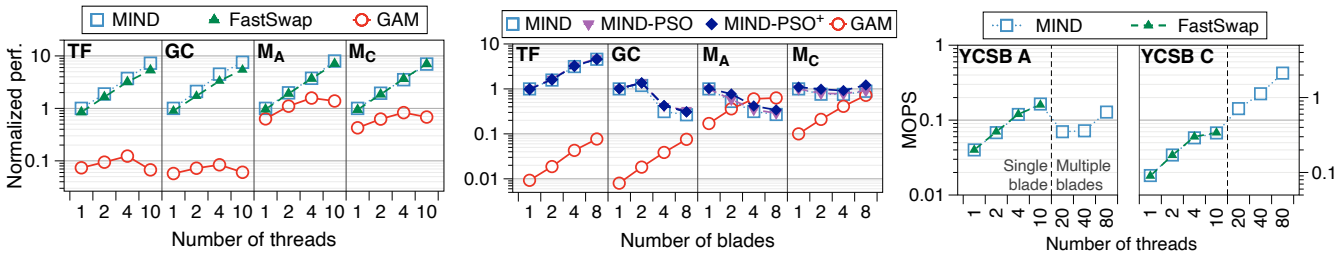


Fig. 5. Performance scaling (left) on a single compute blade, (center) across compute blades, and (right) for Native-KVS. For (center), each blade runs 10 threads. Performance is normalized by MIND's performance at 1 thread for (left) and 1 blade for (center); the runtimes in seconds for TF, GC, M_A 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.

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 [69], 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.

7.1 Performance Scaling for Real-World Workloads

We start by evaluating MIND's performance scalability.

Intra-blade scaling. Figure 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 5 (center) shows our results; here, while MIND's default memory consistency model is strict (TSO, §6), 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+ results 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 [10, 11].

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 ~1.67×, with a 59× 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 ~2.5× more data in shared pages than TF, generating significantly more state transitions to modified (**M**) state, and incurring frequent invalidations (§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× 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 §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

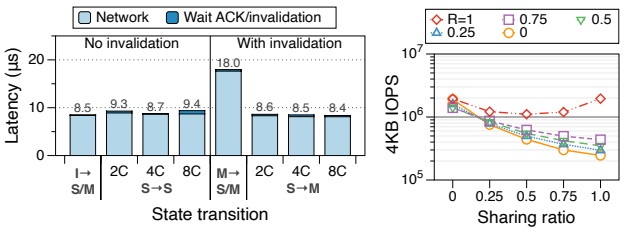


Fig. 6. Performance bottlenecks. (left) Network latency for state transitions, (right) memory throughput vs. read-write/sharing ratios.

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

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.

Latency for cache state transitions. Figure 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 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 (~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 ~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 ~18 μs latency. Note that since the latency for requests with invalidations is 2× 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 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 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 × 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 ~10× 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 7 (left) shows the number of cache directory entries stored in the switch data plane in MIND over time for the workloads evaluated in §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 §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 7 (center) shows that the number of match action rules

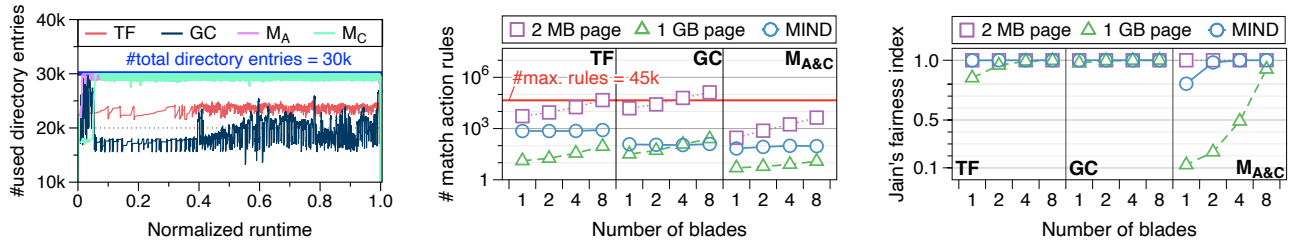


Fig. 7. MIND switch resource bottlenecks. (left) Directory entries, (center) match-action entries for heap, (right) load balancing for heap.

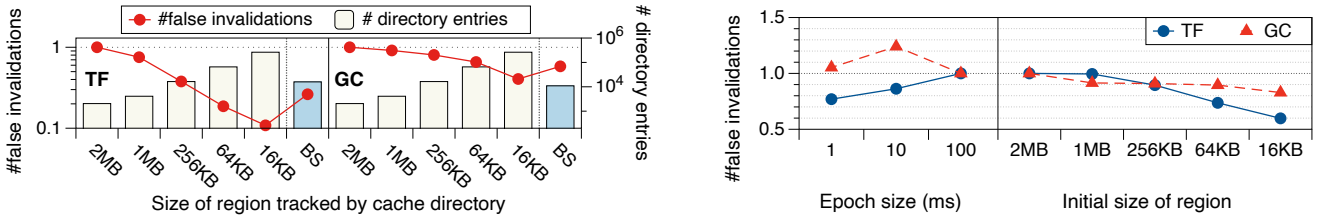


Fig. 8. Evaluating MIND's bounded splitting algorithm. (left) Navigating switch storage vs. performance tradeoff. (right) Impact of epoch & initial region sizing. The number of false invalidations is normalized by value at 2 MB for region size and 100 ms for epoch size.

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 [70, 71]. 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 (§4.1), as shown via Jain's fairness index metric [72] in Figure 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 ($M_{A/C}$).

7.3 Evaluating MIND's Bounded Splitting Algorithm

We now evaluate MIND's bounded splitting algorithm.

Storage vs. performance tradeoff. Recall from §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 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 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 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.

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 §6 and §7, 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.

9 Related Work

While we discussed prior disaggregated memory approaches in §2.2, 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 [73–90]. Most focus was on offloading *application* logic and state to the network, *e.g.*, key-value caches [91, 92] and metadata [93]. Perhaps the most relevant to MIND are NOPaxos [81] and Concordia [94]. 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 [14, 95]. 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 [96] 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 [96] 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 [97], CXL [98] and OpenCAPI [99] target *intra*-server settings, Gen-Z [100] 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 [100, 101], 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.

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.

Acknowledgements

We would like to thank our shepherd Yiying Zhang and anonymous SOSP reviewers for their valuable comments and insightful feedback. We are also grateful to Daehyeok Kim, Victor Gomez and Jonathan Kraft for inputs at various stages of the work. This work is supported in part by NSF Awards `2047220`, `2016422` and `1916817` and their REU supplements.

References

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

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Memory in the Age of Fast Networks. In *Proc. ACM SoCC*, 2017.

[3] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proc. USENIX ATC*, 2019.

[4] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Proc. ACM ASPLOS*, 2014.

[5] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. Memory Disaggregation: Research Problems and Opportunities. In *Proc. IEEE ICDCS*, 2019.

[6] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proc. ACM/IEEE ISCA*, 2009.

[7] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *Proc. IEEE HPCA*, 2012.

[8] Ahmad Samih, Ren Wang, Christian Maciocco, Mazen Kharbutli, and Yan Solihin. *Collaborative Memories in Clusters: Opportunities and Challenges*. 2014.

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

[10] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proc. USENIX OSDI*, 2018.

[11] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *Proc. USENIX NSDI*, 2017.

[12] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proc. EuroSys*, 2020.

[13] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *Proc. USENIX OSDI*, 2016.

[14] Amanda Carbonari and Ivan Beschasnikh. Tolerating Faults in Disaggregated Datacenters. In *Proc. ACM HotNets*, 2017.

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

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

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

[18] 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.

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

[20] 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/.

[21] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network Support for Resource Disaggregation in Next-Generation Datacenters. In *Proc. ACM HotNets*, 2013.

[22] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proc. USENIX OSDI*, 2010.

[23] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proc. USENIX ATC*, 2015.

[24] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proc. USENIX NSDI*, 2014.

[25] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 2010.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proc. ACM SIGCOMM*, 2014.

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

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

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

[30] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *Proc. ACM SIGCOMM*, 2014.

[31] Robert S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17(7):403–412, 1974.

[32] Maurice Vincent Wilkes and Roger Michael Needham. *The Cambridge CAP Computer and Its Operating System*. 1979.

[33] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Trans. on Comput. Syst.*, 1994.

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

[35] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB*, 2018.

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

[37] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. Dc.p4: Programming the forwarding plane of a data-center switch. In *Proc. ACM SOSR*, 2015.

[38] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. ACM SIGCOMM*, 2013.

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

[40] James Laudon and Daniel Lenoski. The SGI Origin: A CcNUMA Highly Scalable Server. In *Proc. ACM/IEEE ISCA*, 1997.

[41] Pat Conway and Bill Hughes. The AMD Opteron Northbridge Architecture. *IEEE MICRO*, 2007.

[42] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE MICRO*, 2010.

[43] Robert A Maddox, Robert J Safranek, and Gurbir Singh. *Weaving high performance multiprocessor fabric: architectural insights into the Intel QuickPath Interconnect*. Intel Press, 2009.

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

[45] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. ACM PPoP*, 1990.

[46] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. Technical report, 1993.

[47] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proc. ACM SoCC*, 2017.

[48] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *Computer*, 1992.

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

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

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

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

[53] Abhishek Bhattacharjee and Daniel Lustig. *Architectural and Operating System Support for Virtual Memory*. Synthesis Lectures on Computer Architecture. 2017.

[54] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. SEESAW: Using Superpages to Improve VIPT Caches. In *Proc. ACM/IEEE ISCA*, 2018.

[55] Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman S Unsal. Energy-efficient address translation. In *Proc. IEEE HPCA*, 2016.

[56] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proc. ACM/IEEE ISCA*, 2014.

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

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

[59] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman S Ünsal. Range Translations for Fast Virtual Memory. *IEEE MICRO*, 2016.

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

[61] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. In *Proc. USENIX OSDI*, 2016.

[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[63] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10. `http://www.cs.toronto.edu/~kriz/cifar.html`.

[64] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proc. USENIX OSDI*, 2012.

[65] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, 1999.

[66] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proc. WWW*, 2010.

[67] MemCached. `http://www.memcached.org`.

[68] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. ACM SoCC*, 2010.

[69] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. ACM PLDI*, 2005.

[70] Peter Hornyack, Luis Ceze, Steve Gribble, Dan Ports, and HM Levy. A study of virtual memory usage and implications for large memory. *Univ. of Washington, Tech. Rep*, 2013.

[71] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting Virtual Memory with Midgard. In *Proc. ACM/IEEE ISCA*, 2021.

[72] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[73] Dan R. K. Ports and Jacob Nelson. When Should The Network Be The Computer? In *Proc. ACM HotOS*, 2019.

[74] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *Proc. ACM SIGCOMM*, 2016.

[75] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *Proc. USENIX NSDI*, 2017.

[76] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *Proc. USENIX NSDI*, 2018.

[77] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proc. ACM SOSR*, 2016.

[78] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. ACM SIGCOMM*, 2017.

[79] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Proc. ACM SOSR*, 2015.

[80] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a Network Service. *IEEE Trans. on Networking*, 2020.

[81] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proc. USENIX OSDI*, 2016.

[82] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proc. USENIX NSDI*, 2015.

[83] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. Infinite Resources for Optimistic Concurrency Control. In *Proc. ACM NetCompute*, 2018.

[84] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proc. ACM SOSP*, 2017.

[85] Zhuolong Yu, Yiwen Zhang, Vladimir Bravermann, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proc. ACM SIGCOMM*, 2009.

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

[87] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proc. ACM HotNets*, 2017.

[88] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation, 2020.

[89] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proc. ACM SIGCOMM*, 2018.

[90] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. The Case for Network Accelerated Query Processing. In *Proc. CIDR*, 2019.

[91] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proc. ACM SOSP*, 2017.

[92] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proc. ACM ASPLOS*, 2017.

[93] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *Proc. USENIX OSDI*, 2020.

[94] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *Proc. USENIX FAST*, 2021.

[95] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the Application. In *Proc. USENIX HotCloud*, 2020.

[96] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proc. USENIX ATC*, 2020.

[97] CCIX Consortium. `https://www.ccixconsortium.com/`.

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

[99] OpenCAPI Consortium. `https://opencapi.org/`.

[100] Gen-Z Consortium. `https://genzconsortium.org/`.

[101] Seokbin Hong, Won-Ok Kwon, and Myeong-Hoon Oh. Hardware Implementation and Analysis of Gen-Z Protocol for Memory-Centric Architecture. *IEEE Access*, 2020.