First-generation Memory Disaggregation for Cloud Platforms

Huaicheng Li[†], Daniel S. Berger[*], Stanko Novakovic[*], Lisa Hsu[‡], Dan Ernst[‡], Pantea Zardoshti[*], Monish Shah[‡], Ishwar Agarwal[‡], Mark D. Hill[‡], Marcus Fontoura[‡], Ricardo Bianchini[*]

[†]Carnegie Mellon University [*]Microsoft Research [‡]Microsoft Azure

Abstract

In Azure, up to 25% of memory is stranded*, i.e., it is leftover after the servers' cores have been rented to VMs. Memory disaggregation promises to reduce this stranding. However, making disaggregation practical for production cloud deployment remains challenging. For example, RDMA-based disaggregation involves too much overhead for common workloads and transparent latency management is incompatible with virtualization acceleration. The emerging Compute Express Link (CXL) standard offers a low-overhead substrate to build memory disaggregation while overcoming these challenges. This paper proposes a first-generation CXL-based disaggregation system that meets the requirements of cloud providers. Our system includes a memory pool controller, and prediction-based system software and distributed control plane designs. Its predictions of VM latency sensitivity and memory usage allow it to split workloads across local and pooled memory while mitigating the higher pool latency.*

Our analysis of production clusters shows that small pools of 8-32 sockets are sufficient to reduce stranding significantly. It also shows that ∼50% of all VMs never touch 50% of their rented memory. In emulated experiments with 150+ workloads, we show our pooling approach incurs a configurable performance loss between 1-5%. Finally, we show that disaggregation can achieve a 9-10% reduction in overall DRAM, which represents hundreds of millions of dollars in cost savings for a large cloud provider.

1 Introduction

Motivation. DRAM has emerged as a key cost driver for public cloud providers. For example, DRAM makes up 50% of the server cost for Azure [1]. Due to long-standing scaling challenges [2–6], DRAM cost is expected to grow even further in the future. Meanwhile, DRAM alternatives are either in their early stages or hitting production and deployment hurdles [7, 8]. Thus, DRAM efficiency has been a top priority for cloud providers, but remains challenging to achieve. For example, Google reports that only ∼40% of DRAM is used in its production clusters [9, 10].

At Azure, we find that a major contributor to DRAM inefficiency is platform-level *memory stranding*. Memory stranding occurs when a server's cores are fully rented to virtual machines (VMs), but unrented memory remains. With the cores exhausted, the remaining memory is unrentable on its own, and is thus stranded. Surprisingly, we find that up to 25% of DRAM may become stranded at any given moment (see §2). Thus, reducing memory stranding can have an outsized impact on hardware costs as the provider can purchase less DRAM. A promising direction is the pooling of memory through memory disaggregation [11–13].

Unfortunately, memory disaggregation incurs a potential performance penalty in that memory from the pool has higher access latency. To achieve the most DRAM savings, public cloud providers would prefer to hide the additional latency from (1) the vast majority of customers who are not expert developers and (2) workloads that do not manage memory placement and performance explicitly. The challenge is achieving this in a practical and broadly applicable manner.

Memory disaggregation for the public cloud. In more detail, making memory disaggregation practical for the public cloud faces multiple functional challenges. First, we must preserve *customer inertia*, *i.e.*, require no modifications to customer workloads or the guest OS. Second, the system must be *compatible with virtualization acceleration* techniques such as direct I/O device assignment to VMs [14, 15] and SR-IOV [16]. Third, the system must be available as *commodity hardware*.

Due to these requirements, most of the prior memory disaggregation work does not apply: custom hardware-based designs [13, 17–20], systems that require changes to the VM guest [11, 21–37], and implementations that rely on page faults [15] are not deployable in the cloud today (see §4.1).

Fortunately, the emerging Compute Express Link (CXL) interconnect standard [38] greatly facilitates fast and deployable disaggregated memory. CXL enables native load/store (`ld/st`) accesses to disaggregated memory for Intel, AMD, and ARM processors [39–41] and Samsung, Micron, and SK Hynix memory modules [42, 43]. CXL also has a low performance impact: directly-connected CXL memory can be accessed with roughly the same latency as a NUMA hop. CXL has become the best option for memory disaggregation by superceding competing protocols [44, 45].

Open questions. While CXL defines the disaggregation protocol, many questions remain open at every layer of the stack. At the hardware layer, how should the provider construct a memory pool with CXL? How should it balance the

pool size with the higher latency of larger pools? At the system software layer, how should the provider manage and expose the pooled memory to guest OSes? How much additional memory latency can cloud workloads tolerate? At the distributed system layer, how should the provider schedule VMs on machines with CXL memory? Could predictions of memory behavior and latency sensitivity help produce better schedules? If so, how accurate are the predictions?

Though memory disaggregation is related to NUMA and two-tier memory systems, the prior work on these topics does not answer the above questions. NUMA policies focus on moving compute close to memory, *e.g.*, a process or VM is scheduled on cores in the same NUMA domain as the memory pages it accesses [46–48]. This is not applicable to CXL where memory pools do not have local cores. Two-tier systems focus on migrating hot pages to local memory [31, 49–53]. Page migration is not supported in public cloud platforms due to virtualization accelerators (§4). Thus, providers require a new way to overcome the challenges of higher memory access latency.

Our work: CXL-based disaggregation for public clouds. Our work answers the above open questions. First, through extensive measurements (§3.1) in Azuredatacenters, we find that stranding hotspots emerge broadly across clusters. This insight leads us to a small memory pool design between 8–32 sockets to significantly reduce stranding. Larger pools do further reduce stranding but yield diminishing returns.

Second, we introduce the first CXL-based full-stack memory disaggregation design that satisfies the requirements of a cloud platform. Our design consists of the following components at each layer of the datacenter stack.

Hardware layer: We propose a new multi-ported external memory controller directly connected to 8-32 sockets via CXL. Each socket has some fast local memory but can access a portion of the memory pool at slightly higher latency (equivalent to 1 NUMA hop).

System software layer: Our design manages pool memory in large memory slices to achieve practical resource overheads. It exposes pool memory to a VM's guest OS as a zero-core virtual NUMA (zNUMA) node (*i.e.*, a node with memory but no cores, like Linux's CPU-less NUMA [54]), which effectively biases memory allocations away from the zNUMA node and improves performance.

Distributed system software layer: Our control plane relies on predictions of VM memory latency sensitivity and the amount of memory will actually be touched. We call the untouched memory "frigid". VMs that are predicted to be insensitive to memory latency are fully backed by pool DRAM, whereas memory-sensitive VMs are provisioned with a zNUMA node corresponding to their frigid memory. We use the predictions for scheduling at VM deployment time, and asynchronously manages the allocation of pooled memory to servers and changes a VM placement when it detects that its prediction was incorrect.

Our evaluation based on VM traces from 100 production clusters shows that we can achieve a 9-10% reduction in overall DRAM needs with each pool connected to 32 sockets. These savings correspond to hundreds of millions of dollars for a large cloud provider. Using more than 150 workloads in a CXL emulation testbed, we also show that most workloads are insensitive to CXL latency even when their entire working set is allocated in pool memory. However, 25% of them suffer at least a 20% performance loss. In terms of memory access behaviors, our results show that ∼50% of all VMs never touch 50% of their rented memory. Our models predict this percentage accurately, as well as whether a workload is memory latency-sensitive. When workloads are properly split between local and pooled memory, slowdowns are configurable to be lower than 1-5%. Exposing the zNUMA node to the guest OS is very effective at producing this split.

Contributions. Our main contributions are as follows:

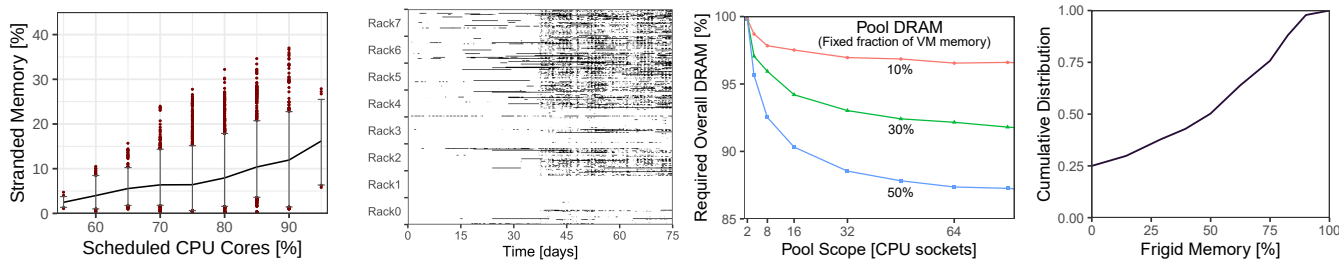
- The first public characterization of memory stranding and frigid memory at a large public cloud provider.
- The first CXL-based full-stack disaggregation design that is practical and performant for broad deployment.
- An accurate prediction model for latency and resource management at datacenter scale.
- An extensive performance evaluation and analysis with over 150 workloads that validates our design.
- To-be-open-sourced research artifacts, including our emulation testbed, production traces, prediction models, etc.

2 Background

Memory stranding. Cloud VMs demand a vector of resources (cpu, memory, etc.) [9, 55–57]. Scheduling a variety of VM types with highly heterogeneous resource demands onto servers leads to a multi-dimensional bin-packing problem [56, 58–60], which remains a theoretical challenge [60]. VM schedulers are also subject to many other constraints, such as spreading VMs across multiple failure domains. As a result, it is difficult to provision servers that closely match the resource demands of the incoming VM mix.

When the DRAM-to-core ratio of VM arrivals and the server resources does not match, tight packing becomes more difficult and resources may end up *stranded*. We define a resource as *stranded* when it is technically available to be rented to a customer, but is practically unavailable as some other resource has exhausted. The typical scenario for *memory stranding* is that all cores have been rented, but there is still memory available in the server.

Reducing stranding via disaggregation. There are multiple techniques that can reduce memory stranding. For example, oversubscribing cores [61, 62] enables more memory to be rented. However, oversubscription only applies to a subset of VMs for performance reasons. Oversubscription also requires a few unrented cores and, thus, cannot address memory stranding. In fact, our stranding measurements at



(a) Stranding averages and outliers (b) Stranding over time (c) Pooling savings (d) Memory usage

Figure 1: **Memory stranding.** *(a) Stranding increases significantly as more CPU cores are scheduled, the error bars indicate the 5^{th} and 95^{th} percentile (outliers in red); (b) stranding dynamics change over time and can affect a broad set of servers, (c) small pools of around 32 sockets are sufficient to significantly reduce overall memory needs; and (d) VMs have a significant amount of frigid memory.*

Azure(§3.1) include multiple clusters that enable oversubscription and still show significant memory stranding.

The approach we target is to disaggregate a portion of memory into a pool that is accessible by multiple hosts [11–13]. This breaks the fixed nature of server hardware configuration. If memory can be dynamically reassigned to different hosts at different times, we can shift memory resources to where they are needed, instead of relying on each individual server to be configured for all cases pessimistically. Thus, we can provision servers close to the average DRAM-to-core ratios and tackle deviations via the memory pool.

Disaggregation via CXL. CXL provides a practical basis for memory disaggregation in the form of a low-latency interconnect protocol and allows CPUs to access CXL memory using regular `ld` and `st` instructions. The bandwidth of a $\times 8$ CXL link approximates that of a DDR5 channel.

While the CXL 2.0 specification [63] standardizes a protocol for memory pooling, the standard leaves open key questions in the design and implementation of a memory pool. For example, while the API defines device discovery and doorbells for host/CXL communication, it does not prescribe how the host manages pool memory. Pool memory management is complicated by hardware resource constraints[1] which forces memory allocation to happen at 1GB granularity (slices). and poses memory fragmentation challenges similar to huge pages [64, 65]. Another example is that, while CXL defines APIs for pool management, it does not define the process of reassigning pool memory between hosts and how to schedule VMs on pool memory.

3 Memory Stranding and Latency Sensitivity

3.1 Stranding at Azure

In this section, we quantify the severity of memory stranding and frigid memory at Azureusing production data.

Dataset. We measure stranding in 100 cloud clusters over a 75-day period. These clusters host mainstream VM workloads and are representative of the majority of the server fleet. We select clusters with similar deployment years, but spanning all major regions on the planet. A trace from each cluster contains millions of per-VM arrival/departure events, with the time, duration, resource demands, and server-id.

Memory stranding. Figure 1a shows the daily average amount of stranded DRAM across clusters, bucketed by the percentage of scheduled CPU cores. In clusters where 75% of CPU cores are scheduled for VMs, 6% of memory is stranded. This grows to over 10% when ∼85% of CPU cores are allocated to VMs. This makes sense since stranding is an artifact of highly utilized nodes, which correlates with highly utilized clusters. Outliers are shown by the error bars, representing 5^{th} and 95^{th} percentiles. At 95^{th}, stranding reaches 25% during high utilization periods. Individual outliers even reach 30% stranding.

Figure 1b shows stranding over time across 8 racks. A workload change (around day 36) suddenly increased stranding significantly. Furthermore, stranding can affect many racks concurrently (*e.g.*, racks 2, 4–7) and it is generally hard to predict which clusters/racks will have stranded memory.

NUMA spanning. Many VMs are small and can fit on a single socket. On two-socket systems, the hypervisor at Azureseeks to schedule such VMs entirely (cores and memory) on a single NUMA node. In rare cases, we see *NUMA spanning* where a VM has all of its cores on one socket and a small amount of memory from another socket. We find that spanning occurs for about 2% of VMs and fewer than 1% of memory pages, on average.

Savings from pooling. Azurecurrently does not pool memory. However, by analyzing its VM-to-server traces, we can estimate the amount of DRAM that could be saved via pooling. Figure 1c presents average reductions from pooling DRAM when VMs are scheduled with a fixed percentage of either 10%, 30%, or 50% of pool DRAM. The pool scope refers to the number of sockets that can access the same DRAM pool. As the pool scope increases, the figure shows that required overall DRAM decreases. However,

[1]Each address range block that is onlined/offlined during pool management requires an invididual entry in the host-managed device memory (HDM) decoder. The number of entries in combination with the pool memory sizes enforces a 1GB entry size.

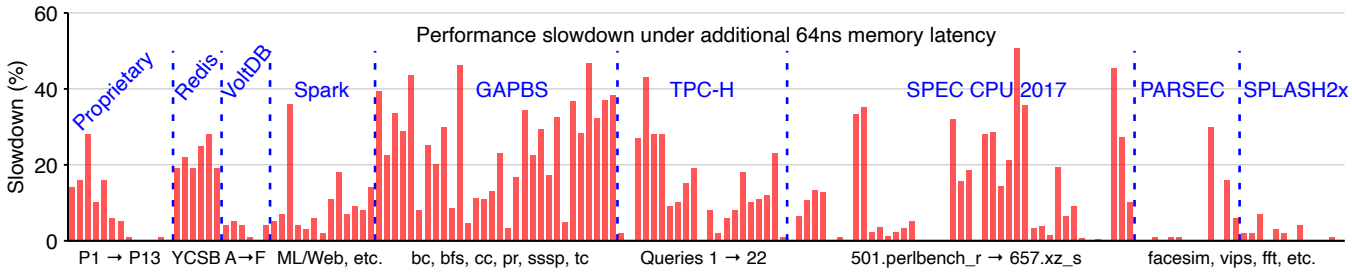


Figure 2: **Workload performance slowdown under additional 64ns memory latency (§3.3).** *This graph shows the performance sensitivity of representative workloads to additional memory latency (as in CXL). X-axis shows 158 representative workloads; Y represents the normalized performance slowdown (i.e., compared to workload performance backed by local DRAM).* **Workloads:** *"Proprietary" denotes Azure's internal production workloads, e.g., databases, web search, machine learning, and analytics. The rest are open-source workloads, such as YCSB (A–F) [66] with in-memory stores (Redisk/VoltDB), in-memory computing Spark workloads in HiBench [67], graph processing (GAPBS) [68], and high-performance computing benchmark sets such as SPEC CPU [69], PARSEC [70], and SPLASH2x [71].*

this effect diminishes for larger pools. For example, with a fixed 50% pool DRAM, a pool with 32 sockets saves 12% of DRAM while a pool with 64 sockes saves 13% of DRAM. Note that allocating a fixed 50% of memory to pool DRAM leads to significant performance loss compared to socket-local DRAM (§6). Our design overcomes this challenge with multiple techniques (§4).

Summary and implications. From this analysis, we draw a few important observations and implications for memory disaggregation:

- We observe 3-16% of stranded memory in production, with some clusters reaching 30%.
- Almost all VMs fit into one NUMA node.
- Pooling memory across 16-32 sockets can reduce cluster memory demand by 10%. This suggests that memory disaggregation can produce significant cost reductions but assumes that a high percentage of DRAM can be allocated on memory pools. When implementing DRAM pools with cross-NUMA latencies, providers must carefully mitigate potential performance impacts.

3.2 VM Memory Usage at Azure

Figure 1d shows the distribution of percentage of frigid memory (computed as 100 minus the *peak* memory usage) across our cloud clusters. We calculate the peak memory usage of each VM as the maximum utilization over its lifetime based on the amount of "committed" memory, spot-checked by scanning the VMs' page-access bits (§5). Generally, we find that while VM memory usage varies across clusters, all clusters have a significant fraction of VMs with frigid memory. Overall, the 50^{th} percentile is 50% frigid memory.

Summary and implications. From this analysis, we draw key observations and implications:

- VM memory usage varies widely.
- In the cluster with the least amount of frigid memory, still over 50% of VMs have more than 20% frigid memory. Thus, there is plenty of frigid memory that can be disaggregated at no performance penalty.
- The challenges are (1) predicting how much frigid memory each VM is likely to have and (2) confining the VM's accesses to local memory. Our design addresses both.

3.3 Workload Sensitivity to Memory Latency

To understand the performance impact of CXL latency for a broad variety of workloads in Azure's datacenters, we evaluate 158 workloads under two scenarios where the workload memory is fully backed by either (1) local DRAM or (2) emulated CXL memory. The experimental setups are detailed in §6.1. Figure 2 presents the workloads' slowdowns when their entire working sets are allocated in CXL memory, compared to having them entirely in local DRAM.

We find that 20% of the 158 workloads experience no slowdown under CXL. An additional 23% of the workloads see less than 5% slowdowns. At the same time, some workloads see severe slowdowns: 25% of the workloads take >20% performance hits and 12% will even suffer more than 30% performance degradations.

Different workload classes are affected differently, *e.g.*, GAPBS (graph processing) workloads generally see higher slowdowns. However, the variability within each workload class is typically much higher than across workload classes. For example, within GAPBS even the same graph kernel reacts very differently to CXL latency, based on different graph datasets. Overall, every workload has at least one workload with less than 5% slowdown and one workload with more than 25% slowdown (except SPLASH2x).

Azure's proprietary workloads are less impacted than the overall workload set. Of the 13 production workloads, 6 do not see noticeable impact (<1%); 2 see ∼5% slowdown; and the remaining half are impacted by 10–28%. This is in part because these production workloads are NUMA-aware and often include data placement optimizations.

Summary and implications. While the performance of some workloads is insensitive to disaggregated memory latency, some are heavily impacted, experiencing up to 51% slowdowns. This motivates our design decision to include

socket-local DRAM alongside pool DRAM to mitigate CXL latency impact for those latency-sensitive workloads.

4 Memory Disaggregation System Design

4.1 Design Goals and Requirements

Cloud platforms rely on hypervisor-based virtualization — including for their container and serverless offerings [72] — to isolate multiple tenants. Deploying disaggregation at scale within this platform faces multiple requirements, which we categorize into performance goals (**PG1–PG3**) and functional requirements (**FR1–FR3**).

PG1: VM performance. It must be comparable or better than today's offerings. Small VMs have their entire memory backed by socket-local DRAM 98% of the time (§3.1). We seek to achieve performance within a small and configurable margin of socket-local DRAM for almost all VMs. We call this margin the *performance degradation margin* (PDM).

PG2: Resource efficiency. Providers seek low overheads within the platform software and hardware. For example, CPU cores lost to virtualization overhead cannot be rented out to users [14, 73, 74]. Another constraint is the number of available memory decoder entries (§2).

PG3: Small blast radius. Providers seek to offer their VMs high availability while minimizing capacity that cannot be used due to component failure. We thus need to minimize the impact (*blast radius*) of failures for additional components introduced for memory disaggregation.

FR1: Customer inertia. Users expect VM compatability to run their OSes/applications without any modifications. Requiring changes limits adoption. Thus, a disaggregation solution must be able to work with opaque and unchanged VMs, guest OSes, and applications.

FR2: Compatibility. Providers rely on virtualization accelerators to improve I/O performance [14, 15, 73, 74]. For example, direct I/O device assignment (DDA) [14, 15] and Single Root I/O Virtualization (SR-IOV) [73, 74] allow I/O requests to bypass the hypervisor stack. These accelerators are enabled by default for all VM types at Azure.

DDA and SR-IOV require the whole address range of a VM to be statically pinned in hypervisor-level page tables [15, 75–79]. As a consequence, memory disaggregation cannot rely on hypervisor page faults, which would be needed to deploy existing RDMA-based systems at the platform level [29–37]. For the same reason, we cannot migrate pages between local memory and a pool as used in existing two-tier memory systems [31, 49–52]. In fact, VM memory allocations can only be changed at VM start or through live migration that temporarily disables acceleration [80].

There are a few options to overcome static pinning. First, the Page Request Interface extension of the PCIe Address Translation Service (ATS/PRI) [81] allows DMA operations to be initiated without requiring memory to be pinned. However, ATS/PRI devices are not available today due to implementation challenges. Second, device vendors have proposed their own standard for page fault handling [15], but this approach does not apply to multiple vendors and device types. Third, virtual IOMMUs [75, 78, 79] allow fine-grained pinning but require significant guest OS changes.

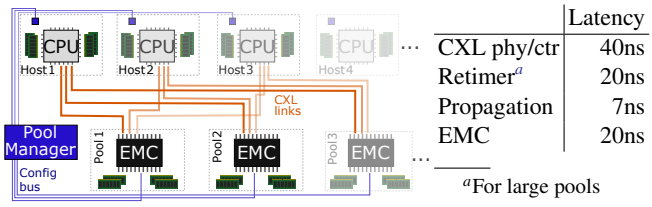


Figure 3: **Our hardware design.** *Each CPU connects to several external memory controllers (EMC). Each EMC offers CXL ports for several sockets and DDR ports for several DIMMs, which constitute the pool. Dashed lines around each CPU and EMC indicate separate failure domains.*

FR3: Commodity hardware. Disaggregation must be deployable on commodity hardware that is available from multiple vendors to avoid single-source supply chain risks.

4.2 Design Overview

Our hardware design (§4.3) comprises a new external memory controller connected via CXL (§2). CXL surfaces pool memory as a load/store device to the CPU, facilitating adding pooled memory to existing applications without software modifications (FR1). Codesigning the hardware with the system software enables us to achieve the functional requirements (FR1-3) and low resource overheads (PG2). We achieve a small blast radius (PG3) via redundant controllers and minimal interleaving (§4.4). We achieve high VM performance (PG1) via an intelligent VM scheduling pipeline and distributed performance monitoring and mitigation components (§4.5). Our scheduling and monitoring pipelines use novel prediction models (§4.6) to ensure that VM performance does not exceed the PDM. Finally, deploying disaggregation in the cloud requires a simple and robust design. Thus, we avoid architectural support for complex features, *e.g.*, cross-pool coherent shared memory.

4.3 Hardware Layer

We introduce a new shared memory module, constructed as an external memory controller (EMC) ASIC with DDR memory. The EMC is a CXL memory device that supports both the configuration and `CXL.mem` [82] protocols for pool memory accesses. Each EMC hosts four 80-bit DDR5 channels of pool DRAM and offers multiple CXL links to allow CPU sockets to access the memory.

As the EMC is a multiported device, it manages arbitration of requests and tracks ownership of memory regions assigned to various hosts. Further, the EMC must support the same reliability, availability and serviceability capabilities [83, 84] of server-grade memory controllers, such as

memory error correction, management, and isolation.

Our memory pool comprises multiple EMCs to minimize the chance of server downtime. Each CPU socket connects to multiple EMCs, for example with a $\times 8$ CXL link which matches the bandwidth of a DDR5 channel. The number of EMCs and attached DDR modules can be scaled independently to meet capacity goals for different clusters. Figure 3 overviews the design and the expected latency contributed by each component. The simplest pool (*e.g.*, 4-8 sockets) would comprise a small set of EMCs, with each directly connected to all sockets in a single large server chassis. For this case, cable lengths are short enough to not require a retimer, which keeps the average-case latency estimate to 67ns (Figure 3). A larger pool (*e.g.*, 32 sockets) can be configured by connecting each EMC device's ports to a different subset of the CPU sockets. This enables sharing across a significantly larger set of hosts to provide higher stranding reduction. However, larger pools will require retimers to extend the signal reach far enough for all sockets. Retimers add about 10ns of latency in each direction [85, 86], bringing the overall access latency to 87ns.

As we observed in §3.1, VMs at Azuredo not typically span multiple sockets. We target this common case by targeting a single-socket coherence domain[2]. The pool implements an ownership model based on 1GB memory slices (§4.4). At the hardware level, the EMC checks all memory accesses for access permission, *i.e.*, whether requestor and owner of the cacheline's address range match. Disallowed accesses result in fatal memory errors.

A single-socket coherency domain has advantages over standard cache-coherent multiprocessor systems. First, cache coherence significantly increases cost and performance overheads, especially for latency-sensitive workloads. Second, separating the pool memory from the local socket memory opens up flexibility in system memory configurations, providing the ability to compose systems in a finer-grained manner that can better match the different DRAM/core averages seen by different clusters (§3.1).

4.4 System Software Layer

Our system software involves 3 components to manage pools, handle failures, and expose pool memory to VMs.

Pool management. A pool comprises multiple hosts with each host running a separate hypervisor and operating system within a separate cache coherent domain (one socket). As pool memory is not cache-coherent across hosts, it is explicitly moved between hosts. We call this task pool management, as shown in Figure 4. It involves two challenges: 1) implementing the control paths for pool-level memory assignment and 2) preventing pool memory fragmentation.

[2]With rising per-socket core counts, industry trends point towards single-socket servers [87–90]. When deploying on two-socket hosts, one would connect both sockets of each host to the EMC to achieve symmetric one-NUMA-hop latency to all DDR memory.

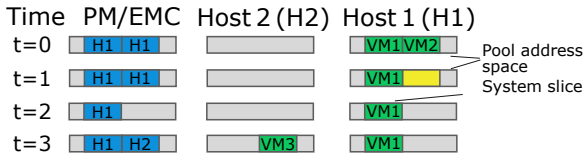


Figure 4: **Typical pool management task.** *Pool memory is assigned to at most one host at a time. This figure shows the flow when a VM (VM2) terminates on a host (H1) at time t=0 and the memory is needed on another host (H2). H1's memory manager first offlines each memory slice previously used by VM2 (t=1). The Pool Manager (PM) then reassigns these slices to H2 (t=2). H2's manager then onlines these slices and starts the new VM (t=3).*

Assignment works by enabling/disabling host memory at runtime from the Pool Manager, which implements the CXL 2.0 fabric manager workflow. The Pool Manager is connected to EMCs and CPU sockets via a low-power management bus (*e.g.*, [91]), as shown in Figure 3. To allocate pool memory, the Pool Manager uses the bus to communicate to the EMC as well as the driver on the host (via interrupts).

From the host's point of view, pool memory is mapped but in an offline state at boot time. The BIOS discovers socket-local and pool capacity through CXL device discovery and maps them to the OS address space through the `ACPI SRAT` tables [92]. Once mapped, the pool address range is marked hot-pluggable and "not enabled". When the host receives an interrupt, our driver reads which address range is hot-plugged and whether this is an onlining or offlining operation. The driver then communicates with the OS memory manager to bring the memory online/offline.

As discussed in §2, pool memory has to be onlined and offlined in 1GB slices due to hardware resource constraints. If these large slices lead to memory fragmentation, we face inefficiencies as the contiguous 1GB address range must be free before we can offline a 1GB slice for reassignment to another host. We overcome fragmentation using two techniques. First, pool memory is allocated to VMs in 1GB-aligned increments (§4.5). While this prevents fragmentation due to VM starts and completions, our experience has shown that memory allocations from host agents and drivers can still fragment slice memory. Our second technique is a special-purpose memory partition for pool memory. The memory manager ensures that only the hypervisor can allocate memory from this partition. Host agents and drivers allocate memory in another memory partition, which effectively contains fragmentation.

With these optimizations, our initial experiments with offlining 1 GB slices on the host show that it takes 10-100 milliseconds per GB. In contrast, onlining memory has insignificant overheads on the order of microseconds per GB. This means that shifting memory after a 128GB VM is completed would require tens of seconds, which is insufficient for VM-start SLOs at Azure. This observation is reflected in our asynchronous offlining process (§4.5).

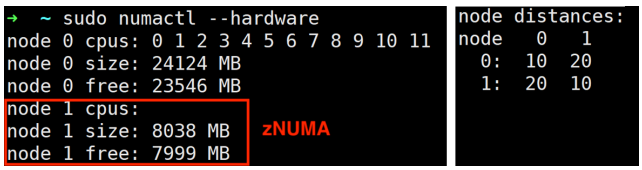


Figure 5: **zNUMA.** *zNUMA as seen from inside a Linux VM*

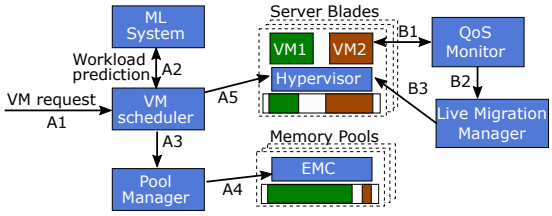


Figure 6: **Control plane workflow (§4.5).** **A)** *The VM scheduler uses ML-based predictions that identify latency-sensitive VMs and their likely amount of frigid memory to decide on VM placement (see Figure 7).* **B)** *The monitoring pipeline reconfigures VMs if quality-of-service (QoS) targets are not met.*

Minimizing blast radius. To minimize the blast radius of any single component and facilitate memory hot-plugging, hosts do not use interleaving across their attached EMCs. Consequently, each socket and each EMC is a separate failure domain, as shown by the dashed boxes in Figure 3. If an EMC fails, only the VMs using that specific component/link will be affected, while VMs running on local DRAM and other EMCs can continue to run. If a CPU socket fails, the associated pool memory can be freed and reallocated to other sockets. If the Pool Manager fails, VMs can continue to run and access memory, but we cannot reallocate pool memory.

Exposing pool memory to VMs. A key building block of our latency management mechanism is to expose pool memory as a zNUMA node. To implement a zNUMA node, the hypervisor adds a memory block (`node_memblk`) without any associated CPU cores (no entry in the `node_cpuid`) in the `ACPI SLIT/SRAT` tables [92]. Figure 5 shows how this appears from the point of view of a Linux VM. The figure also shows that the hypervisor exposes the correct latency in the NUMA distance matrix (`numa_slit`) to facilitate the guest OS's memory management.

When we expose a zNUMA node to any modern guest OS, it preferentially allocates memory from the local NUMA node before going to zNUMA. Recall the observation on VM memory usage in Figure 1d. This means that, if we can predict the amount of frigid memory for a VM (§4.6), and size the zNUMA for that VM, then we can effectively focus the VMs memory allocations on its local and lower-latency NUMA node. The VM's guest OS can also use NUMA-aware memory management [50, 93].

As mentioned above, we avoid fragmentation by backing each zNUMA node with contiguous 1GB-aligned address ranges from the pool memory partition.

4.5 Distributed Control Plane Layer

Figure 6 shows the two tasks performed by our control plane: (A) predictions to allocate memory during VM scheduling and (B) QoS monitoring and resolution.

Predictions and VM scheduling (A). We use ML-based prediction models (§4.6) to decide how much pool memory can be allocated to a VM during scheduling. After a VM request arrives (A1), the scheduler queries the distributed ML serving system (A2) for a prediction on how much local memory to allocate for the VM. The scheduler then informs the Pool Manager about the target host and associated pool memory needs (A3). The Pool Manager triggers a memory onlining workflow using its configuration bus connections to the EMCs and host (A4). As described in §4.4, memory onlining can be completed in less than a millisecond even for large VMs. VM terminations trigger memory offlining, which the Pool Manager performs in the background of VM creation. Finally, the VM scheduler informs the host's hypervisor to start the VM using a zNUMA node matching the onlined memory amount.

QoS monitoring (B). After VMs are allocated, we continuously inspect their performance via the QoS monitor. The monitor queries hypervisor and hardware performance counters (B1) and uses an ML model of latency sensitivity (§4.6) to decide whether the VM is being negatively affected by its pool memory latency. If the impact exceeds a threshold, the monitor triggers a live VM migration [80] via the migration manager (B2), which effects the migration (B3).

4.6 Prediction Models

Our VM scheduling (A) and QoS monitoring (B) algorithms rely on two prediction models as shown in Figure 7.

Predictions for VM scheduling (A). For scheduling, we first check if we can correlate a workload history with the VM requested. This works by checking if there have been previous VMs with the same metadata as the request VM, *e.g.*, the customer's subscription, VM types, and location. This is based on the observation that VMs from the same subscription tend to exhibit similar behaviors [55].

If we have prior workload history, we make a prediction on whether this VM is likely to be memory latency insensitive, *i.e.*, its performance would be within the `PDM` while using only pool memory. (Model details appear below.) Latency-insensitive VMs are allocated entirely on pool DRAM.

If the VM has no workload history or is predicted to be latency-sensitive, we predict the minimum amount of frigid memory over its lifetime. Interestingly, frigid memory predictions with only generic VM metadata such as VM type, guest OS, and location are accurate (§6). VMs without frigid memory are allocated entirely with local DRAM. VMs with a $\texttt{FM} > 0$ percentage of frigid memory are allocated with $\texttt{FM}\%$ of pool memory and a corresponding zNUMA node; the remaining memory is allocated on local DRAM.

If we underpredict `FM`, the VM will not touch the slower

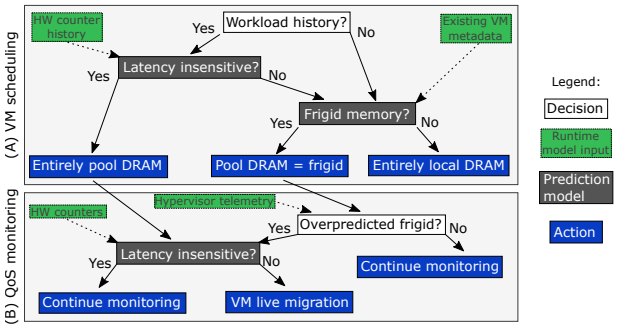


Figure 7: **Our prediction models (§4.6).** *Our two prediction models (dark grey) rely on telemetry (green boxes) that is available for all VM types, including third-party opaque VMs.*

pool memory as the guest OS prioritizes allocating local DRAM. If we overpredict FM, we rely on the QoS monitor for mitigation. Importantly, we always keep a VM's memory mapped in hypervisor page tables at all times. This means that even if our predictions happen to be incorrect, performance does not fall off a cliff.

QoS monitoring (B). For zNUMA VMs, we monitor if it overpredicted the amount of frigid memory during scheduling. For VMs entirely backed with pool DRAM and zNUMA VMs with less frigid memory than predicted, we use the sensitivity model to determine whether the VM workload is suffering excessive performance loss. If not, the QoS monitor initiates a live VM migration to a configuration allocated entirely on local DRAM.

Model details and compatibility with opaque VMs. The two types of prediction models, latency insensitivity and frigid memory, must work for opaque VMs. We achieve this requirement by relying on hardware and hypervisor telemetry. To predict latency sensitivity and performance impact, we rely on core-PMU performance counters motivated by Intel's top-down method for performance analysis (TMA) [94, 95]. We remark that, while TMA was developed for Intel, its relevant parts are available on AMD and ARM as well [96]. In TMA, the CPU's pipeline usage is broken down into hierarchical categories such as "memory bound", which is defined as pipeline stalls due to memory loads and stores. We further subdivide memory boundedness into "DRAM latency bound" and other components. Based on these metrics and sampled measurements of performance slowdowns, we train a model that predicts whether the slowdown exceeds the PDM. Our evaluation discusses the accuracy of this approach (§6) on a set of 150 workloads. In a production deployment, we will further augment our model with labels gathered from internal systems running in Azure's cloud as these systems can make performance metrics available to us.

To train our memory footprint model, we rely on hypervisor telemetry. We conservatively define memory footprint as the peak number of pages that the VM touches during its lifetime. This footprint is available by scanning access bits in the hypervisor page table (§5). Access bti scans are also used in VM working set size estimation [97, 98]; however, we use a more conservative approach that resets access bits only every 30 minutes to minimize VM performance overheads. We further augment the access bit scanning data with an existing approximate memory usage counter that is available for 98% of VMs in Azureas a hypervisor counter.

Parameterization of prediction models. Our latency insensitivity model is parameterized to stay below a *target rate of false positives* (FP), *i.e.*, workloads it incorrectly specifies as latency insensitive but which are actually sensitive to memory latency. This parameter enforces a tradeoff as the percentage of workloads that are labeled as latency insensitive (LI) is a function of FP. For example, a small rate of FP like 0.1% also enforces a small percentage of LI say 5%.

Similarly, our frigid memory model is paramterized to stay below a *target rate of overpredictions* (OP), *i.e.*, workloads that have less frigid memory than predicted and thus will use memory pages on the zNUMA node. This parameter also enforces a tradeoff as the percentage of frigid memory (FM) is a function of OP. For example, a small rate of OP like 0.1% also enforces a small percentage of FM say 3%.

With two models and their respective parameters, we need to decide how to balance FP and OP between the two models, while maximizing the average amount of memory that is allocated on the CXL pool. This optimization problem is instantiated with the given performance degradation margin (PDM) and the percentage of VMs that can be allowed to exceed the degradation goal (X).

$$\begin{aligned} &\text{maximize}(\#\texttt{LI}_{\texttt{PDM}}) + (\%\texttt{FM}) \\ &\text{subject to}(\texttt{FP}_{\texttt{PDM}}) + (\texttt{OP}) \leq X \end{aligned} \tag{1}$$

Note that X essentially defines how often the QoS monitor has to engage and initiate live migrations. Thus, we instatiate X based on a budget for live migrations. Besides X and the PDM, our design has no other parameters as it automatically solves the optimization problem from Eq.(1). The models have no other parameters as we rely on their respective framework's default hyperparameters (§5).

5 Implementation

While we do not yet have hardware for our new EMC, implementing and deploying our design requires extensive testing within Azure's system software and distributed control plane. At the same time, we find that existing two-socket servers from Intel can near-perfectly emulate the latency characteristics of our memory pool. We thus implement and test our software changes on existing servers both in the lab as well as on production nodes.

System software. This implementation comprises three parts. First, we emulate a single-socket system with a CXL pool on a two-socket server by disabling all cores in one

socket, while keeping its memory accessible from the other socket. This memory mimics the pool.

Second, we change Azure's hypervisor to instantiate arbitrary zNUMA topologies. We extend the API between the control plane and the host to pass the desired zNUMA topology to the hypervisor.

Third, we implement support in Azure's hypervisor for the telemetry required for training our models. To associate hardware counters with individual VMs, we extend the virtual core struct with a copy of the virtual core's hardware counter state. This enables us to consistently associate hardware counters with a VM as it gets scheduled on different physical cores. Specifically, the hypervisor copies the virtual hardware counter state to physical core registers when the virtual core is scheduled. When the virtual core is descheduled, we save the hardware register state in the virtual core struct. To enable accurate footprint telemetry, we enable access bit scanning in hypervisor-level address translation tables. Since our implementation only needs to estimate the memory footprint, we scan and reset access bits every 30 minutes, which adds virtually no overhead.

Distributed control plane. We train our prediction models by aggregating daily telemetry into a central database. The latency insensitivity model uses a simple random forest (RandomForest) from Scikit-learn [99] to classify whether a workload exceeds the PDM. The model uses a set of 200 hardware counters as supported by current Intel processors. The frigid memory model uses a gradient boosted regression model (GBM) from LightGBM [100] and makes a quantile regression prediction with a configurable target percentile. After exporting to ONNX [101], the prototype adds the prediction (the size of zNUMA) on the VM request path using a custom inference serving system similar to [102–104]. Azure's VM scheduler incorporates zNUMA requests and pool memory as an additional dimension into its bin packing, similar to other cluster schedulers [56, 105–109].

6 Evaluation

We seek to answer the following questions in the evaluation:
- What is the performance of zNUMA VMs on production and lab nodes? (§6.2 and §6.3)
- How accurate are our prediction models? (§6.4)
- What are the end-to-end DRAM savings? (§6.5)

6.1 Experimental Setup

We evaluate the performance of our prototype using a large set of cloud workloads. Specifically, we evaluate 158 workloads spanning in-memory databases and KV-stores (Redis [110], VoltDB [111], and TPC-H on MySQL [112]), data and graph processing (Spark [67] and GAPBS [68]), HPC (SPLASH2x [71]), CPU and shared-memory benchmarks (SPEC CPU [69] and PARSEC [70]), and a range of Azure's internal workloads (Proprietary). Figure 2 overviews these workloads. We additionally use VM scheduling simulations to quantify the DRAM savings for a deployment at scale.

Prototype setup. We run experiments on production servers at Azureand similarly-configured lab servers. The production servers use two 24-core Intel Skylake 8157M with 256GB of DDR4 on each socket. For socket-local DRAM, we measure an access latency of 78ns and bandwidth exceeding 80GB/s. The zNUMA memory latency is 64ns higher which is slightly faster than a CXL pool. The bandwidth when using only zNUMA memory is around 30GB/s — about 3/4 of the bandwidth of a CXL $\times 8$ link. Our BIOS disables hyper-threading, turbo-boost, and C-states.

We use performance when entirely backed by socket-local DRAM as our *baseline*. We present performance as normalized slowdowns, *i.e.*, the ratio to the baseline. Performance metrics are workload specific, *e.g.*, job runtime, throughput and 99^{th} tail latency, etc.

Each experiment involves running the application with one of 7 zNUMA sizes (as percentages of the workload's memory footprint in Figure 9). With at least three repetitions of each run and 158 workloads, our evaluation spans more than 3,500 experiments and 10,000 machine hours. All of these experiments were performed on lab servers; we repeat spot check for outliers on production servers.

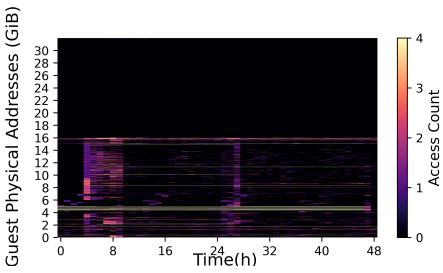
Simulations and model evaluation. Our simulations are based on traces of production VM requests and their placement on servers. The traces are from randomly selected 100 clusters across 34 datacenters globally over 75 days.

The simulator implements different memory allocation policies and tracks each server and each pool's memory capacity at second accuracy. Generally, the simulator schedules VMs on the same nodes as in the trace and changes their memory allocation to match the policy. For rare cases where a VM does not fit on a server, *e.g.*, due to insufficient pool memory, the simulator moves the VMs to another server.

The traces also include resource logs which we use to evaluate our frigid memory prediction model. We observe that 80% of VMs have sufficient history to make a sensitivity prediction. However, the trace does not include the workload's perceived performance (opaque VMs). We thus evaluate our latency sensitivity model based on our 158 workloads.

6.2 zNUMA VMs on Production Nodes

We perform a small-scale experiment on Azureproduction nodes to validate our intuition on zNUMA VMs. The experiment evaluates four internal workloads: an audio/video conferencing application, a database service, a key-value store, and a business analytics service (interactive data visualization and business intelligence). To see the effectiveness of zNUMA, we assume a correct prediction of frigid memory, *i.e.*, the local footprint fits into the VM's local vNUMA node. Figure 8 shows access bit scans over 48 hours from the video workload and a table that shows the traffic to the zNUMA



Workload	Traffic to zNUMA
Video	0.25%
DB	0.06%
KV store	0.11%
Analytics	0.38%

Figure 8: **Effectiveness of zNUMA (§6.2).** *We schedule latency sensitive workloads with two vNUMA nodes. The local vNUMA node is large enough to cover the workload's footprint and the zNUMA nodes holds the VM's remaining memory on the CXL pool. Access bit scans, e.g., for Video (right), show that this configuration indeed minimizes traffic to the zNUMA node.*

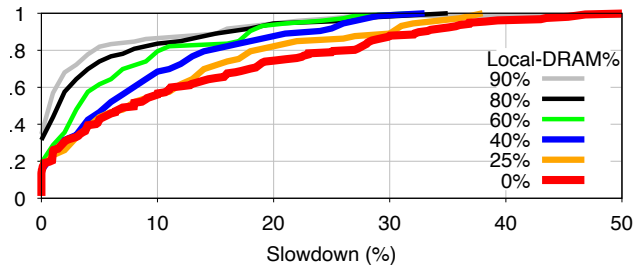


Figure 9: **Slowdown CDFs under mixed local/pool memory.** *This figure shows the CDFs of workload performance slowdown under various mixed local and pool memory, e.g., 25% (orange line) represents 25% of workload memory is backed by local DRAM and the rest 75% by pool memory.*

node for the four workloads.

Finding 1. The access bit scan shows that accesses are spread out within the local vNUMA node but rarely happen on the zNUMA node. The zNUMA nodes are effective at containing the memory access to the local vNUMA node. However, a small fraction of accesses goes to the zNUMA node. We suspect that this is in part due to the guest OS memory manager's metadata that is explicitly allocated on each vNUMA node. We find that the video workload sends fewer than 0.25% of memory accesses to the zNUMA node. Similarly, the other three workloads send 0.06-0.38% of memory access to the zNUMA node.

Implications. With a negligible fraction of memory accesses on zNUMA, we deduce that performance impacts from the higher latency of CXL will likely have negligible effects.

6.3 zNUMA VMs in the Lab

We scale up our evaluation to 158 workloads in a lab setting. Since we fully control these workloads, we can now also explicitly measure their performance. We rerun each workload for differently-sized zNUMA nodes between 0-100% of the workload's footprint. Note that for latency sensitive workloads, we regularly size the zNUMA to match the footprint (100%). Thus, this is effectively a sensitivity study and shows what happens without our frigid memory prediction model. Figure 9 shows a CDF of slowdowns for 0-90% local DRAM.

Finding 2. With a correct prediction of frigid memory, workload slowdowns are usually 0%. For a few workloads, slowdowns are ±2% due to variability of the workload across of repetitions (noise).

Implications. This performance result is expected since very few accesses are issued to the zNUMA node (§6.2). This validates that few accesses translates into minimal performance impact. Our evaluation can thus assume no performance impact under correct predictions of frigid memory (§6.5).

Finding 3. For overpredictions of frigid memory (and correspondingly undersized local vNUMA nodes), performance generally improves with increasing percentage of local DRAM. We find that good performance requires significant amounts of local DRAM. For example with 25% local DRAM (orange line in Figure 9), we see the slowdown distribution largely overlaps with the distribution for 0% local DRAM. Furthermore, we find that a long tail remains even for high percentages of local DRAM. As we increase the percentage of local memory to 40%, and 90%, the slowdown decreases, but the tail remains. For example, with 10% of CXL memory (gray line in the figure), almost 80% of workloads experience less than 5% degradation. However, the last 5% of workloads continue to experience more than 20% performance degradation and the last 1% of workloads see 30%. We use DAMON to verify that these workloads indeed actively access their entire working set.

Implications. Allocating a fixed percentage of pool DRAM to VMs leads to significant performance slowdowns. There are only two strategies to reduce this impact: 1) identify which workloads will see slowdowns and 2) allocate frigid memory on the pool. We employ both strategies.

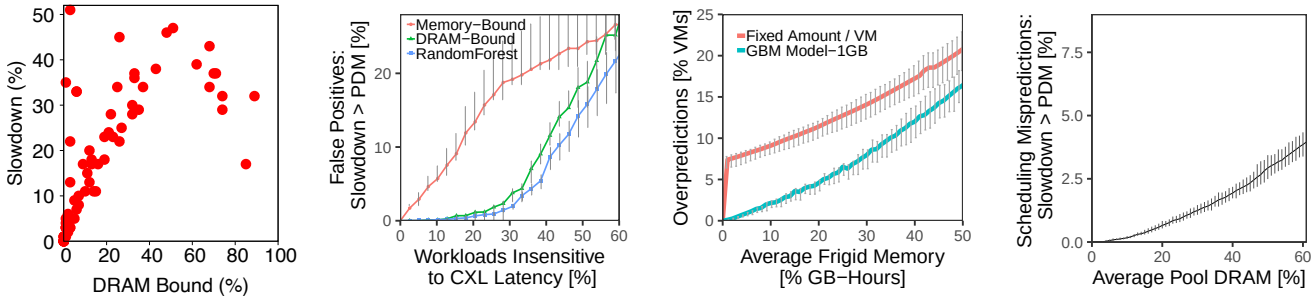
6.4 Performance of Prediction Models

We evaluate our prediction models (§4.6) and its combined prediction model based on Eq.(1).

6.4.1 Predicting Latency Sensitivity

We seek to predict whether a VM is latency insensitive, *i.e.*, whether running the workload on pool memory would stay within the performance degradation margin (`PDM`). We tested the model for `PDM` between 1-10%, but report details only for 5%. Other `PDM` values lead to qualitatively similar results. We compare simple thresholds on memory and DRAM boundedness [94, 95] to our RandomForest (§5).

Figure 10a shows the correlation between DRAM boundedness and performance slowdowns. Figure 10b shows the model's false positive as a function of the percentage of workloads labeled as latency insensitive, similar to a precision-recall curve [113]. Error bars show 99^{th} percentiles from a 100-fold validation based on randomly split-



(a) *DRAM boundedness* (b) *Predicting insensitivity* (c) *Predicting frigid memory* (d) *Combined model*

Figure 10: **Performance of our prediction models.** *(a) shows the correlation between DRAM bound and slowdowns; (b) compares heuristics to a ML-based prediction model for whether a workload exceeds 5% performance degradation; (c) compares a static to a ML-based prediction models for frigid memory; (d) shows the tradeoff between average allocation of pool memory and mispredictions, which have to be mitigated via live migrations.*

ting into equal-sized training and testing datasets.

Finding 4. While DRAM boundedness is correlated with slowdown, we find examples where high slowdown occurs even for a small percentage of DRAM boundedness (points at the top-left corner in Figure 10a). For example, multiple workloads exceed 20% slowdown with just two percent of DRAM boundedness.

Implication. This shows the general hardness of predicting whether workloads exceed the PDM. Predictors will always have some statistical errors.

Finding 5. We find that "DRAM bound" significantly outperforms "Memory bound" (Figure 10b). Our RandomForest performs slightly better than "DRAM bound".

Implication. Our RandomForest can place 30% of workloads on the pool with only 2% of false positives.

6.4.2 Predicting Frigid Memory

We seek to predict the amount of frigid memory over a VM's future lifetime (§4.6). We evaluate this model using metadata and resource usage logs from 100 clusters over 75 days. The model is trained on VMs that started and ended during the first 15 days and evaluated on the subsequent 60 trace days. Figure 10c compares our GBM model to a simple policy where a fixed fraction of memory is assumed to be frigid across all VMs. The figure shows the rate of overpredictions as a function of the average amount of frigid memory (higher is better).

Finding 6. We find that the GBM model is 3.5× more accurate than the static policy, *e.g.*, when labeling 10% of memory as frigid, GBM overpredicts only 2.5% of VMs while the static policy overpredicts 9%.

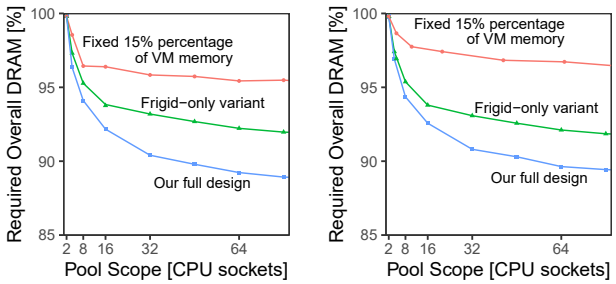
Implication. Our prediction model can identify 20% of frigid memory while only overpredicting 4% of VMs.

6.4.3 Combined Prediction Models

We evaluate our combined prediction model (Eq.(1)) based on the number of "scheduling mispredictions", *i.e.*, the fraction of VMs that will exceed the PDM. This incorporates the overpredictions of frigid memory, how much the model overpredicted, and the probability of this overprediction leading to a workload exceeding the PDM. Figure 10d showing scheduling mispredictions as a function of the average amount of cluster DRAM that is allocated on its pools.

Finding 7. Our combined model outperforms both of the individual models by finding their optimal combination.

Implication. We can schedule 40% of DRAM on pools with only 2% of scheduling mispredictions.



(a) *PDM =5%, X=2%* (b) *PDM =1%, X=6%*

Figure 11: **Memory savings under performance constraints (§6.5).** *Simulated end-to-end evaluation of memory savings under two different performance scenarios: (a) PDM =5% and scheduling mispredictions X=2%. (b) PDM =1% and scheduling mispredictions X=6%.*

6.5 End-to-end Reduction in Stranding

We evaluate our design's end-to-end performance while constraining its rate of scheduling mispredictions. Each scheduling misprediction will have to be mitigated by the QoS monitor and a potential VM live migration.

Figure 11 shows the reduction in overall memory across all clusters as a function of pool scope for "full design", a variant with only frigid-memory predictions, and a strawman static allocation policy. We evaluate two scenarios, Figure 11a shows PDM =5% and scheduling mispredictions X=2% and Figure 11a shows PDM =1% and scheduling mispredictions X=6%.

	HW [13, 17–20, 115, 117]	VMM/OS [29–37]	Runtime [21–23]	App [24–27]	**Our design**
Inertia	✓	✓	✓	✗	✓
Performance	✓	✗	✗	✗	✓
Deployability	✗	✓	✓	✓	✓
VirtAccel	✓	✗	✓	✓	✓
CpuEfficiency	✓	✗	✗	✗	✓

Table 1: **Memory disaggregation approaches (§7).** *Our design satisfies key requirements for public cloud datacenters.*

We compare our design to a static strawman policy. In the first scenario, the strawman statically allocates each VM with 15% of pool DRAM. About 10% of VMs would touch the pool DRAM (Figure 10c). Of those touching pool DRAM, we'd expect that about $\frac{1}{4}$ would see a slowdown exceeding a PDM =5% (Figure 9). So, this strawman would have about 2.5% of scheduling mispredictions. In the second scenario, the strawman statically allocates each VM with 10% of pool DRAM and has about 6% scheduling mispredictions.

Finding 8. At a pool scope of 32 sockets and in the first scenario, we reduce overall DRAM requirements by 10%, while our frigid-only variant reduces DRAM by 7%, and static reduces DRAM by less than 4%. In the second scenario, we reduce DRAM requirements by slightly less.

Implication. We can safely reduce DRAM cost and can make up for a more aggressive performance target (PDM) by relying more on online mitigations using its QoS monitor.

Finding 9. Throughout the simulations, our pool memory offlining speeds remain are below 1GB/s for 99.99% and below 10GB/s for 99.999% of VM starts.

Implication. Our design is practical as it stays within the memory offlining limits of our prototype (10GB/s).

7 Related Work

Table 1 compares our design to state-of-the-art memory disaggregation approaches at the hardware (HW), hypervisor/host (VMM/OS), runtime, and application (App) levels.

Hardware-level disaggregation: Hardware-based disaggregation designs [13, 17–20, 114, 115] are not easily deployable as they do not rely on commodity hardware. For instance, ThymesisFlow [13] proposes an FPGA-based rack-scale memory disaggregation design on top of OpenCAPI [116]. It shares some similar goals with our work, however, its design and performance target are fundamentally different. ThymesisFlow advocates application changes for performance, while we focuse on platform-level ML-driven pool memory management that is transparent to users.

Hypervisor/OS level disaggregation: Hypervisor/OS level approaches [11, 28–37] rely on page faults and access monitoring to maintain the working set in local DRAM. Such OS-based approaches bring significant overhead, jitter, and are incompatible with virtualization acceleration (*e.g.*, DDA).

Runtime and application level disaggregation: Runtime-based disaggregation designs [21–23] propose customized APIs for remote memory access. Similarly, some applications (*e.g.*, databases) are specifically designed to leverage remote memory for scalability and performance [24, 118–120]. While effective, this approach requires developers to explicitly use these mechanisms at the application level.

Memory tiering: Prior works have considered the broader impact of extended memory hierarchies and how to handle them [31, 52, 121–123]. For example, Google achieves 6µs latency via proactive hot/cold page detection and compression [31, 51]. Thermostat [52] uses an online page classification method for efficient application-transparent and huge-page-aware two-tier memory management. Nimble [50] optimizes Linux's page tracking mechanism to tier pages for increased migration bandwidth. HeMem [49] manages page migrations between tiered DRAM/NVM with low-overhead and asynchronous policies (*e.g.*, cpu events sampling). Our work takes a different ML-based approach looking at memory disaggregation design at the platform-level and is generally orthogonal to these prior works.

ML for systems: ML is increasingly applied to tackle systems problems, such as cloud efficiency [55, 61], memory/storage optimizations [124, 125], microservices [126], caching/prefetching policies [127, 128]. We uniquely apply ML methods for frigid memory prediction to support pooled memory provisioning to VMs without jeopardizing QoS.

Coherent memory and NUMA optimizations: Traditional cache coherent NUMA architectures [129] use specialized interconnects to implement a shared address space. There are also many system-level optimizations for NUMA, such as NUMA-aware data placement [130] and proactive page migration [93]. Distributed Shared Memory (DSM) systems offer a coherent global address space across the network (*e.g.*, Ethernet) [131, 132]. AutoNUMA [46] and other NUMA scheduling policies [47, 48] try to balance compute and memory across NUMA nodes, but is not applicable to CXL memory as it has no local cores. Our design does not require coherence as allocated pool memory to a VM will not be shared with others. zNUMA's zero-core nature requires rethinking of existing optimizations which are largely optimized for symmetric NUMA systems.

8 Conclusion

DRAM costs are an increasing cost factor for cloud providers. This paper is motivated by an analysis of stranded and allocated-but-unused memory across 100 production cloud clusters. We proposed the first CXL-based full-stack memory disaggregation design that satisfies the requirements of cloud providers. Our design comprises contributions at the hardware, system software, and distributed system layers. Our results showed that a first-generation memory disaggregation can reduce the amount of needed DRAM by 9-10%. This translates into an overall reduction of 4-5% in cloud server cost.

References

[1] CXL And Gen-Z Iron Out A Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/, 2020.

[2] Onur Mutlu. Memory Scaling: A Systems Architecture Perspective. In *IEEE International Memory Workshop (IMW)*, 2013.

[3] Prashant Nair, Dae-Hyun Kim, and Moinuddin K. Qureshi. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[4] Onur Mutlu. Main Memory Scaling: Challenges and Solution directions. In *More than Moore Technologies for Next Generation Computer Design*. Springer, 2015.

[5] Sung-Kye Park. Technology Scaling Challenge and Future Prospects of DRAM and NAND Flash Memory. In *IEEE International Memory Workshop (IMW)*, 2015.

[6] Shigeru Shiratake. Scaling and Performance Challenges of Future DRAM. In *IEEE International Memory Workshop (IMW)*, 2020.

[7] The Next New Memories. https://semiengineering.com/the-next-new-memories/, 2019.

[8] Micron Ends 3D XPoint Memory. https://www.forbes.com/sites/tomcoughlin/2021/03/16/micron-ends-3d-xpoint-memory/, 2021.

[9] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating Job Packing in Warehouse-scale Computing. In *International Conference on Cluster Computing (Cluster)*, 2014.

[10] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of the 2020 EuroSys Conference (EuroSys)*, 2020.

[11] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[12] 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 *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[13] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and Peter Hofstee. ThymesisFlow: A Software-Defined, HW/SW Co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*, 2020.

[14] Intel Virtualization Technology for Directed I/O. https://software.intel.com/content/dam/develop/external/us/en/documents/vt-directed-io-spec.pdf, 2020.

[15] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page Fault Support for Network Controllers. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[16] Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *Workshop on I/O Virtualization*, 2008.

[17] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[18] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project PBerry: FPGA Acceleration for Remote Memory. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2019.

[19] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale Disaggregated Cloud Data Centers: The dReDBox Project Vision. In *Design Automation and Test in Europe (DATE)*, 2016.

[20] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[21] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[22] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated Managed Runtime. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[23] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[24] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[25] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Regions: a Simple Abstraction for Remote Memory. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[26] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[27] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojicic, and Gustavo Alonso. Farview: Disaggregated Memory with Operator Offloading for Database Engines. *arXiv:2106.07102*, 2021.

[28] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level Implications of Disaggregated Memory. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA-18)*, 2012.

[29] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[30] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[31] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[32] Kwangwon Koh, Kangho Kim, Seunghyub Jeon, and Jaehyuk Huh.

Disaggregated Cloud Memory with Elastic Block Management. *IEEE Transactions on Computers (TC)*, 68(1), January 2019.

[33] 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 *Proceedings of the 2020 EuroSys Conference (EuroSys)*, 2020.

[34] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[35] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud. In *International Conference on Distributed Computing Systems (ICDCS)*, 2020.

[36] Wenqi Cao and Ling Liu. Hierarchical Orchestration of Disaggregated Memory. *IEEE Transactions on Computers (TC)*, 69(6), June 2020.

[37] Youngmoon Lee, Hassan Al Maruf, Mosharaf Chowdhury, and Kang G. Shin. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. *arXiv:1910.09727*, 2019.

[38] Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org, 2020.

[39] Sapphire Rapids Uncovered: 56 Cores, 64GB HBM2E, Multi-Chip Design. https://www.tomshardware.com/news/intel-sapphire-rapids-xeon-scalable-specifications-and-features, 2021.

[40] CXL Consortium Member Spotlight: Arm. https://www.computeexpresslink.org/post/cxl-consortium-member-spotlight-arm, 2021.

[41] Lisa Su. AMD Unveils Workload-Tailored Innovations and Products at The Accelerated Data Center Premiere. https://www.amd.com/en/press-releases/2021-11-08-amd-unveils-workload-tailored-innovations-and-products-the-accelerated, November 2021.

[42] Compute Express Link Industry Members. https://www.computeexpresslink.org/members, 2021.

[43] Samsung Unveils Industry-First Memory Module Incorporating New CXL Interconnect Standard. https://news.samsung.com/global/samsung-unveils-industry-first-memory-module-incorporating-new-cxl-interconnect-standard, 2021.

[44] CXL ABSORBS GEN-Z. https://www.nextplatform.com/2021/11/23/finally-a-coherent-interconnect-strategy-cxl-absorbs-gen-z/, 2021.

[45] Patrick Kennedy. CXL May Have Just Won as AMD Joins CXL. https://www.servethehome.com/cxl-may-have-just-won-as-amd-joins-cxl/, 2019.

[46] Jonathan Corbet. Autonuma: the other approach to numa scheduling. https://lwn.net/Articles/488709/, 2012.

[47] Dulloor Subramanya Rao and Karsten Schwan. vnuma-mgr: Managing vm memory on numa platforms. In *IEEE International Conference on High Performance Computing*, pages 1–10, 2010.

[48] Ming Liu and Tao Li. Optimizing Virtual Machine Consolidation Performance on NUMA Server Architecture for Cloud Workloads. In *Proceedings of the 41th Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[49] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[50] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[51] Linux Memory Management Documentation - zswap. https://www.kernel.org/doc/html/latest/vm/zswap.html, 2020.

[52] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[53] Luiz Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *Proceedings of the 25th International Conference on Supercomputing (ICS)*, 2011.

[54] Christopher Lameter. Flavors of Memory supported by Linux, Their Use and Benefit. https://events19.linuxfoundation.org/wp-content/uploads/2017/11/The-Flavors-of-Memory-Supported-by-Linux-their-Use-and-Benefit-Christoph-Lameter-Jump-Trading-LLC.pdf, 2019.

[55] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[56] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM Allocation Service at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[57] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[58] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for Vector Bin Packing. https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/, 2011.

[59] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[60] Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and Bruce Shepherd. Tight Bounds for Online Vector Bin Packing. In *Proceedings of the 45th ACM symposium on Theory of Computing (STOC)*, 2013.

[61] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the 2021 EuroSys Conference (EuroSys)*, 2021.

[62] Pradeep Ambati, Íñigo Goiri, Felipe Vieira Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[63] Compute Express Link 2.0 White Paper. https://b373eaf2-67af-4a29-b28c-3aae9e644f30.filesusr.com/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf, 2021.

[64] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[65] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[66] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[67] HiBench: The Bigdata Micro Benchmark Suite. https://github.com/Intel-bigdata/HiBench, 2021.

[68] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[69] SPEC CPU 2017. https://www.spec.org/cpu2017, 2021.

[70] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[71] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. PARSEC3.0: A Multicore Benchmark Suite with Network Stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News (CAN)*, 44(5), 2017.

[72] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[73] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[74] Single-Root Input/Output Virtualization. httpp://www.pcisig.com/specifications/iov/single_root, 2019.

[75] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[76] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. On the DMA Mapping Problem in Direct Device Assignment. In *Proceedings of the 3rd ACM International Conference on Systems and Storage (SYSTOR)*, 2010.

[77] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[78] Nadav Amit, Muli Ben-Yehuda, IBM Research, Dan Tsafrir, and Assaf Schuster. vIOMMU: Efficient IOMMU Emulation. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, 2011.

[79] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[80] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. Vm live migration at scale. *ACM SIGPLAN Notices*, 53(3), March 2018.

[81] PCI Express Address Translation Services. https://composter.com.ua/documents/ats_r1.1_26Jan09.pdf, 2009.

[82] CXL Specification. https://www.computeexpresslink.org/download-the-specification, 2020.

[83] Reliability, Availability, and Serviceability (RAS) Integration and Validation Guide for the Intel Xeon Processor E7 Family. https://www.intel.com/content/dam/develop/external/us/en/documents/emca2-integration-validation-guide-556978.pdf, 2015.

[84] AMD EPYC brings new RAS capability. https://www.amd.com/system/files/2017-06/AMD-EPYC-Brings-New-RAS-Capability.pdf, 2017.

[85] CXL Use-cases Driving the Need For Low Latency Performance Retimers. https://www.microchip.com/en-us/about/blog/learning-center/cxl--use-cases-driving-the-need-for-low-latency-performance-reti, 2021.

[86] Enabling PCIe 5.0 System Level Testing and Low Latency Mode for CXL. https://www.asteralabs.com/videos/aries-smart-retimer-for-pcie-gen-5-and-cxl/, 2021.

[87] Google. New tau vms deliver leading price-performance for scale-out workloads. https://cloud.google.com/blog/products/compute/google-cloud-introduces-tau-vms, June 2021.

[88] AWS. Amazon ec2 m6g instances. https://aws.amazon.com/ec2/instance-types/m6g/, 2021.

[89] Tony Harvey. Use Single-Socket Servers to Reduce Costs in the Data Center. https://www.gartner.com/en/documents/3894873/use-single-socket-servers-to-reduce-costs-in-the-data-ce, 2018.

[90] Robert W Hormuth. Why Single-socket Servers Could Rule the Future. https://www.nextplatform.com/2019/04/24/why-single-socket-servers-could-rule-the-future/, 2019.

[91] MIPI I3C Bus Sensor Specification. https://www.mipi.org/specifications/i3c-sensor-specification, 2021.

[92] UEFI. Advanced Configuration and Power Interface Specification, 2021.

[93] Huang Ying. AutoNUMA: Optimize Memory Placement for Memory Tiering System. https://lwn.net/Articles/835402/, 2019.

[94] Ahmad Yasin. A Top-Down Method for Performance Analysis and Counters Architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[95] Top-down Microarchitecture Analysis Method. https://software.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html, 2021.

[96] Mateusz Jarus and Ariel Oleksiak. Top-Down Characterization Approximation Based on Performance Counters Architecture for AMD Processors. *Simulation Modelling Practice and Theory*, 2016.

[97] Anna Melekhova. Machine Learning in Virtualization: Estimate a Virtual Machine's Working Set Size. In *2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, 2013.

[98] Ahmed A Harby, Sherif F Fahmy, and Ahmed F Amin. More Accurate Estimation of Working Set Size in Virtual Machines. *IEEE Access*, 7, 2019.

[99] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research (JMLR)*, 12, 2011.

[100] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A Highly Efficient Gradient Boosting Decision Tree. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[101] ONNX. Open Neural Network Exchange: the Open Standard for Machine Learning Interoperability. https://onnx.ai/, 2021.

[102] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, High-performance ML Serving. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS)*, 2017.

[103] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[104] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[105] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5), 2016.

[106] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4), 2013.

[107] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, 2015.

[108] Malte Schwarzkopf, Andy Konwinski, Michael Abdel-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 2013 EuroSys Conference (EuroSys)*, 2013.

[109] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[110] Redis. https://redis.io, 2021.

[111] VoltDB. https://www.voltdb.com, 2021.

[112] TPC-H Benchmark. http://www.tpc.org/tpch, 2021.

[113] Michael Buckland and Fredric Gey. The Relationship Between Recall and Precision. *Journal of the American Society for Information Science*, 45(1), 1994.

[114] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[115] Jorge Gonzalez, Alexander Gazman, Maarten Hattink, Mauricio G. Palma, Meisam Bahadori, Ruth Rubio-Noriega, Lois Orosa, Madeleine Glick, Onur Mutlu, Keren Bergman, and Rodolfo Azevedo. Optically Connected Memory for Disaggregated Data Centers. In *IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020.

[116] OpenCAPI Consortium. https://opencapi.org/, 2021.

[117] Seung seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[118] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[119] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-Out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[120] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, 2017.

[121] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenters. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[122] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen mei Hwu. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[123] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[124] Giulio Zhou and Martin Maas. Learning on Distributed Traces for Data Center Storage Systems. In *Proceedings of the 4th Conference on Machine Learning and Systems (MLSys)*, 2021.

[125] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[126] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[127] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A Hierarchical Neural Model of Data Prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[128] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying Deep Learning to the Cache Replacement Problem. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[129] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, 1997.

[130] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[131] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[132] Kai Li and Hudak Paul. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4), November 1989.