Pond: CXL-Based Memory Pooling Systems for Cloud Platforms

Huaicheng Li
Virginia Tech
Carnegie Mellon University

Daniel S. Berger
Microsoft Azure
University of Washington

Lisa Hsu
Unaffiliated

Daniel Ernst
Microsoft Azure

Pantea Zardoshti
Microsoft Azure

Stanko Novakovic
Google

Monish Shah
Microsoft Azure

Samir Rajadnya
Microsoft Azure

Scott Lee
Microsoft

Ishwar Agarwal
Intel

Mark D. Hill
Microsoft Azure
University of Wisconsin-Madison

Marcus Fontoura
Stone Co

Ricardo Bianchini
Microsoft Azure

ABSTRACT

Public cloud providers seek to meet stringent performance requirements and low hardware cost. A key driver of performance and cost is main memory. Memory pooling promises to improve DRAM utilization and thereby reduce costs. However, pooling is challenging under cloud performance requirements. This paper proposes Pond, the first memory pooling system that both meets cloud performance goals and significantly reduces DRAM cost. Pond builds on the Compute Express Link (CXL) standard for load/store access to pool memory and two key insights. First, our analysis of cloud production traces shows that pooling across 8-16 sockets is enough to achieve most of the benefits. This enables a small-pool design with low access latency. Second, it is possible to create machine learning models that can accurately predict how much local and pool memory to allocate to a virtual machine (VM) to resemble same-NUMA-node memory performance. Our evaluation with 158 workloads shows that Pond reduces DRAM costs by 7% with performance within 1-5% of same-NUMA-node VM allocations.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Hardware** → **Emerging architectures**.

KEYWORDS

Compute Express Link; CXL; memory disaggregation; memory pooling; datacenter; cloud computing.

ACM Reference Format:
Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3578835

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3578835

1 INTRODUCTION

Motivation. Many public cloud customers deploy their workloads in the form of virtual machines (VMs), for which they get virtualized compute with performance approaching that of a dedicated cloud, but without having to manage their own on-premises datacenter. This creates a major challenge for public cloud providers: achieving excellent performance for opaque VMs (*i.e.*, providers do not know and should not inspect what is running inside the VMs) at a competitive hardware cost.

A key driver of both performance and cost is main memory. The gold standard for memory performance is for accesses to be served by the same NUMA node as the cores that issue them, leading to latencies in tens of nanoseconds. A common approach is to preallocate all VM memory on the same NUMA node as the VM's cores. Preallocating and statically pinning memory also facilitate the use of virtualization accelerators [4], which are enabled by default, for example, on AWS and Azure [12, 14]. At the same time, DRAM has become a major portion of hardware cost due to its poor scaling properties with only nascent alternatives [73]. For example, DRAM can be 50% of server cost [7].

Through analysis of production traces from Azure, we identify *memory stranding* as a dominant source of memory waste and a potential source of massive cost savings. Stranding happens when all cores of a server are rented (*i.e.*, allocated to customer VMs) but unallocated memory capacity remains and cannot be rented. We find that up to 25% of DRAM becomes stranded as more cores become allocated to VMs.

Limitations of the state of the art. Despite this significant amount of stranding, reducing DRAM usage in the public cloud is challeng-

ing due to its stringent performance requirements. For example, existing techniques for process-level memory compression [51, 78] require page fault handling, which adds microseconds of latency, and moving away from statically preallocated memory.

Pooling memory via memory disaggregation is a promising approach because stranded memory can be returned to the disaggregated pool and used by other servers. Unfortunately, existing pooling systems also have microsecond access latencies and require page faults or changes to the VM guest [51, 59].

Our work. This work describes Pond, *the first system to achieve both same-NUMA-node memory performance and competitive cost for public cloud platforms.* To achieve this, Pond combines hardware and systems techniques. It relies on the Compute Express Link (CXL) interconnect standard [5], which enables cacheable load/store (`ld/st`) accesses to pooled memory on Intel, AMD, and ARM processors [11, 16, 23] at nanosecond-scale latencies. CXL access via loads/stores is a game changer as it allows memory to remain statically preallocated while physically being located in a shared pool. However, even with loads/stores, CXL accesses still face higher latencies than same-NUMA-node accesses. Pond introduces systems support for CXL-based pooling that dramatically reduces the impact of this higher latency.

Pond is feasible because of four key insights. First, by analyzing traces from 100 production clusters at Azure, we find that pool sizes between 8-16 sockets lead to sufficient DRAM savings. The pool size defines the number of CPU sockets able to use pool memory. Further, analysis of CXL topologies lead us to estimate that CXL will add 70-90ns to access latencies over same-NUMA-node DRAM with a pool size of 8-16 sockets, and add more than 180ns for rack-scale pooling. We conclude that grouping 8 dual-socket (or 16 single-socket) servers is enough to achieve most of the benefits of pooling.

Second, by emulating either 64ns or 140ns of additional memory access overheads over same-NUMA-node memory latency, we find that 43% and 37% of 158 workloads are within 5% of the performance on same-NUMA-node DRAM when entirely allocated in pool memory. However, more than 21% of workloads suffer a performance loss above 25%. This emphasizes the need for small pools and shows the challenge with achieving same-NUMA-node performance. This characterization also allows us to train a machine learning (ML) model that can identify a subset of insensitive workloads ahead of time to be allocated on the Pond memory pool.

Third, we observe through measurements at Azure that ~50% of all VMs touch less than 50% of their rented memory. Conceptually, allocating untouched memory from the pool should not have any performance impact even for latency-sensitive VMs. We find that — while this concept does not hold for the uniform address spaces assumed in prior work (*e.g.*, [51, 57]) — it does hold if we expose 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 [52]. Our experiments show zNUMA effectively biases memory allocations away from the zNUMA node. Thus, a VM with a zNUMA sized to match its untouched memory will indeed not see any performance impact.

Fourth, Pond can allocate CXL memory with same-NUMA-node performance using correct predictions of **a)** whether a VM will be latency-sensitive and **b)** a VM's amount of untouched memory. For incorrect predictions, Pond introduces a novel monitoring system that detects poor memory performance and triggers a mitigation that migrates the VM to use only same-NUMA-node memory. Further, we find that all inputs which are needed to train and run Pond's ML models can be obtained from existing hardware telemetry with no measurable overhead.

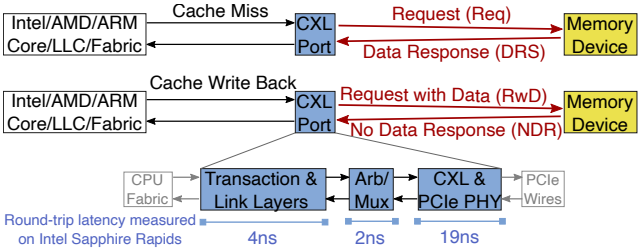


Figure 1: CXL Request Flow (§2). *CPU cache misses and write-backs to addresses mapped to CXL devices are translated to requests on a CXL port by the HDM decoder. Intel measures the round-trip port latency to be 25ns.*

Artifacts. CXL is still a year from broad deployment. Meanwhile, deploying Pond requires extensive testing within Azure's system software and distributed software stack. We implement Pond on top of an emulation layer that is deployed on production servers. This allows us to prove the key concepts behind Pond by exercising the VM allocation workflow, zNUMA, and by measuring guest performance. Additionally, we support the four insights from above by reporting from extensive experiments and measurements in Azure's datacenters. We evaluate the effectiveness of pooling using simulations based on VM traces from 100 production clusters. An open-source version of Pond's emulation layer and a sample of VM traces are available at https://github.com/vtess/Pond.

Contributions. Our main contributions are:

- The first public characterization of memory stranding and untouched memory at a large public cloud provider.
- The first analysis of the effectiveness and latency of different CXL memory pool sizes.
- Pond, the first CXL-based full-stack memory pool that is practical and performant for cloud deployment.
- An accurate prediction model for latency and resource management at datacenter scale. These models enable a configurable performance slowdown of 1-5%.
- An extensive evaluation that validates Pond's design including the performance of zNUMA and our prediction models in a production setting. Our analysis shows that we can reduce DRAM needs by 7% with a Pond pool spanning 16 sockets, which corresponds to hundreds of millions of dollars cost-savings for a large cloud provider.

2 BACKGROUND

Hypervisor memory management. Public cloud workloads are virtualized. To maximize performance and minimize overheads, hypervisors perform minimal memory management and rely on virtualization accelerators to improve I/O performance [4, 8, 56]. Examples of common accelerators are direct I/O device assignment (DDA) [8] and Single Root I/O Virtualization (SR-IOV) [4]. Acceler-

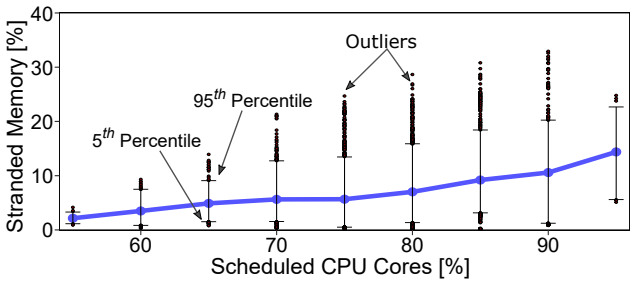


Figure 2: Memory stranding (§3.1). *Stranding increases significantly as more CPU cores are scheduled. Error bars indicate the 5^{th} and 95^{th} percentiles (outliers in dots).*

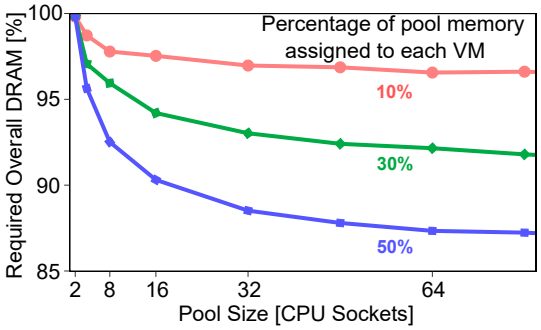


Figure 3: Impact of pool size (§3.1). *Small pools of 32 sockets are sufficient to significantly reduce memory needs.*

ated networking is enabled by default on AWS and Azure [12, 14]. As pointed out in prior work, virtualization acceleration requires statically preallocating (or "pinning") a VM's entire address space [55].

Memory stranding. Cloud VMs demand a vector of resources (*e.g.*, CPUs, memory, etc.) [41, 48]. Scheduling VMs thus leads to a multi-dimensional bin-packing problem [32, 45, 48] which is complicated by constraints such as spreading VMs across multiple failure domains. Additionally, it is difficult to provision servers that closely match the resource demands of future incoming VM mixes at design time. When the DRAM-to-core ratio of VM arrivals and the server resources do not match, tight packing becomes more difficult. 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. Multiple techniques can reduce memory stranding. For example, oversubscribing cores [30, 77] enables more memory to be rented. However, oversubscription only applies to a subset of VMs for performance reasons. Our measurements at Azure (§3.1) include 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. This breaks the fixed hardware configuration of servers. By dynamically reassigning memory 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.

Pooling via CXL. CXL contains multiple protocols including `ld/st` memory semantics (CXL.mem) and I/O semantics (CXL.io). CXL.mem maps device memory to the system address space. Last-level cache (LLC) misses to CXL memory addresses translate into requests on a CXL port whose reponses bring in the missing cachelines (Figure 1). Similarly, LLC write-backs translate into CXL data writes. Neither action involves page faults or DMAs. CXL memory is virtualized using hypervisor page tables and the memory-management unit and is thus compatible with virtualization acceleration. The CXL.io protocol facilitates device discovery and configuration. CXL 1.1 targets directly-attached devices, CXL 2.0 [6, 15] adds switch-based pooling, and CXL 3.0 [28, 70] standardizes switch-less pooling (§4) and higher bandwidth.

CXL.mem uses PCIe's electrical interface with custom link and transaction layers for low latency. With PCIe 5.0, the bandwidth of a bidirectional ×8-CXL port at a typical 2:1 read:write-ratio matches a DDR5-4800 channel. CXL request latencies are largely determined by the CXL port. Intel measures round-trip CXL port traversals at 25ns [69] which, when combined with expected controller-side latencies, leads to an end-to-end overhead of 70ns for CXL reads in a basic topology, compared to NUMA-local DRAM reads. While FPGA-based prototypes report higher latency [44, 60], 70ns-latency-overheads match industry-expectations for ASIC-based memory controllers [28, 60, 69].

3 MEMORY STRANDING & WORKLOAD SENSITIVITY TO MEMORY LATENCY

3.1 Stranding at Azure

This section quantifies the severity of memory stranding and untouched memory at Azure using production data.

Dataset. We measure stranding in 100 cloud clusters over a 75-day period. These clusters host mainstream first-party and third-party VM workloads. They 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 2 shows the distribution of hourly snapshots of the amount of stranded DRAM bucketed by the percentage of CPU cores that are scheduled to host VMs. We find that 6% of memory is stranded in the median snapshot (blue line) where 75% of CPU cores are scheduled for VMs. The median 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. The figure also indicates outliers and error bars, representing 5^{th} and 95^{th} percentiles. Variability is largely due to different VM mixes. For example, the VM mix in some snapshots is biased towards compute-heavy VM types that do not require much DRAM. This can create high stranding, even at low utilization. At the 95^{th} percentile, stranding reaches 25% during high utilization periods. Individual outliers even reach 30% stranding.

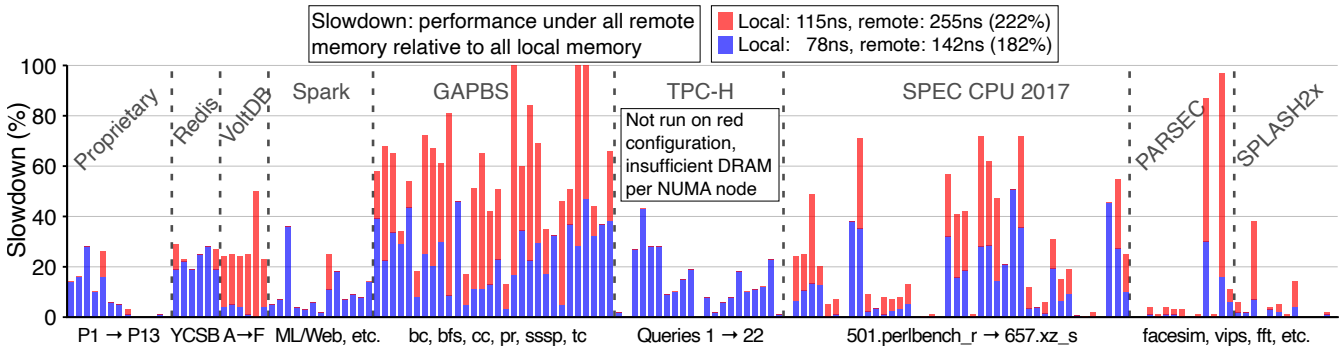


Figure 4: Performance slowdowns when memory latency increases by 182-222% (§3.3). *Workloads have different sensitivity to additional memory latency (as in CXL). X-axis shows 158 representative workloads; Y is the normalized performance slowdown, i.e., performance under higher (remote) latency relative to all local memory. "Proprietary" denotes production workloads at Azure.*

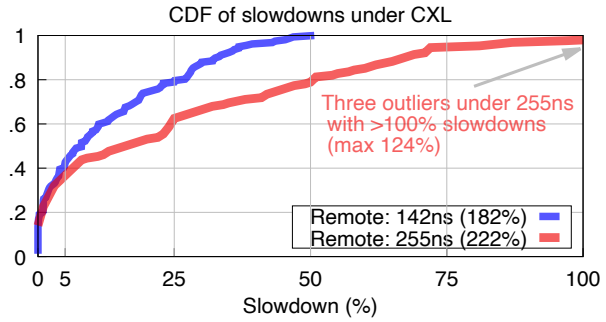


Figure 5: CDF of slowdowns (§3.3). *Higher remote latency (red) only slightly affects the head of the distribution (workloads with less than 5% slowdown). The body and tail of the distribution see significantly higher slowdowns.*

NUMA spanning. Many VMs are small and can fit on a single socket. On two-socket systems, the hypervisor at Azure seeks to schedule such that VMs fit 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 2-3% of VMs and fewer than 1% of memory pages, on average.

Savings from pooling. Azure currently 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 3 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 size refers to the number of sockets that can access the same DRAM pool. As the pool size increases, the figure shows that required overall DRAM decreases. However, 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 sockets 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). Pond overcomes this challenge with multiple techniques (§4).

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

- We observe 3-27% of stranded memory in production at the 95^{th} percentile, with some outliers at 36%.
- 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 pooling 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

We use Pond's telemetry on opaque VMs (§4.2) to characterize the percentage of untouched memory across our cloud clusters. Generally, we find that while VM memory usage varies across clusters, all clusters have a significant fraction of VMs with untouched memory. Overall, the 50^{th} percentile is 50% untouched memory.

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

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

3.3 Workload Sensitivity to Memory Latency

To characterize the performance impact of CXL latency for typical workloads in Azure's datacenters, we evaluate 158 workloads under two scenarios of emulated CXL access latencies: 182% and 222% increase in memory latency, respectively. We then compare the workload performance to NUMA-local memory placement. Experimental details are in §6.1. Figures 4 and 5 show workload slowdowns relative to NUMA-local performance for both scenarios.

Under a 182% increase in memory latency, we find that 26% of the 158 workloads experience less than 1% slowdown under CXL.

An additional 17% of workloads see less than 5% slowdowns. At the same time, some workloads are severely affected with 21% of the workloads facing >25% slowdowns.

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 class 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.

Under a 222% increase in memory latency, we find that 23% of the 158 workloads experience less than 1% slowdown under CXL. An additional 14% of workloads see less than 5% slowdowns. More than 37% of workloads face >25% slowdowns. Generally, we find that higher latency magnifies the effects seen under lower latency: workloads performing well under 182% latency also tend to perform well under 222% latency; workloads severely affected by 182% are even more affected by 222%.

Summary and implications. While the performance of some workloads is insensitive to disaggregated memory latency, some are heavily impacted. This motivates our design decision to include socket-local DRAM alongside pool DRAM to mitigate CXL latency impact for those latency-sensitive workloads. Memory pooling solutions can be effective if they are effective at identifying sensitive workloads.

4 POND DESIGN

Our measurements and observations at Azure (§2–3) lead us to define the following design goals.

G1 Performance comparable to NUMA-local DRAM
G2 Compatibility with virtualization accelerators
G3 Compatibility with opaque VMs and unchanged guest OSes/ applications
G4 Low host resource overhead

To quantify (G1), we define a *performance degradation margin* (PDM) for a given workload as the allowable slowdown relative to running the workload entirely on NUMA-local DRAM. Pond seeks to achieve a configurable PDM, *e.g.*, 1%, for a configurable tail-percentage (TP) of VMs, *e.g.*, 98% (§3.1). To achieve this high performance, Pond uses a small but fast CXL pool (§4.1). As Pond's memory savings come from pooling instead of oversubscription, Pond must minimize pool fragmentation and wastage in its system software layer (§4.2).

To achieve (G2), Pond preallocates local and pool memory at VM start. Pond decides this allocation in its allocation, performance monitoring, and mitigation pipeline (§4.3). This pipeline uses novel prediction models to achieve the PDM (§4.4). Finally, Pond overcomes VM-opaqueness (G3) and host-overheads (G4) using lightweight hardware counter telemetry.

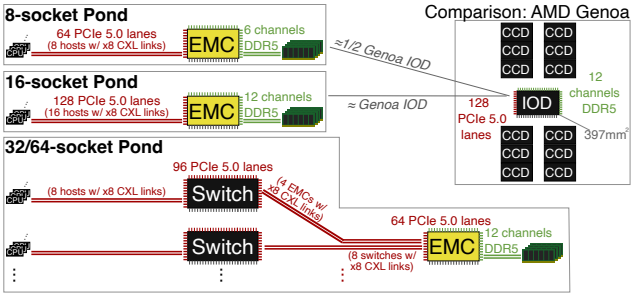


Figure 6: External memory controller (EMC) (§4.1). *The EMC is multi-headed which allows connecting multiple CXL hosts and DDR5 DIMMs. A 16-socket Pond requires 128 PCIe 5.0 lanes and 12 DDR5 channels, which is comparable to the IO-die (IOD) on AMD Genoa [10, 11]. Larger Pond configurations combine a switch with the multi-headed EMC.*

4.1 Hardware Layer

Hosts within a Pond pool have separate cache coherency domains and run separate hypervisors. Pond uses an ownership model where pool memory is explicitly moved among hosts. A new external memory controller (EMC) ASIC implements the pool using multiple DDR5 channels accessed through a collection of CXL ports running at PCIe 5 speeds.

EMC memory management. The EMC offers multiple CXL ports and appears to each host as a single logical memory device [6, 15]. In CXL 3.0 [28, 70], this configuration is standardized as multi-headed device (MHD) [28, §2.5]. The EMC exposes its entire capacity on each port (*e.g.*, to hosts) via a Host-managed Device Memory (HDM) decoder. Hosts program each EMC's address range but treat them initially as offline. Pond dynamically assigns memory at the granularity of 1GB memory slices. Each slice is assigned to at most one host at a given time and hosts are explicitly notified about changes (§4.2). Tracking 1024 slices (1TB) and 64 hosts (6 bits) requires 768B of EMC state. The EMC implements dynamic slice assignment by checking permission of each memory access, *i.e.*, whether requestor and owner of the cacheline's slice match. Disallowed accesses result in fatal memory errors.

EMC ASIC design. The EMC offers multiple ×8-CXL ports, which communicate with DDR5 memory controllers (MC) via an on-chip network (NOC). The MCs must offer the same reliability, availability, and serviceability capabilities [2, 3] as server-grade memory controllers including memory error correction, management, and isolation. A key design parameter of Pond's EMC is the pool size, which defines the number of CPU sockets able to use pool memory. We first observe that the EMC's IO, (De)Serializer, and MC requirements resemble AMD Genoa's $397mm^2$ IO-die (IOD) [10, 11]. Figure 6 shows that EMC requirements for a 16-socket Pond parallel the IOD's requirements, with a small 8-socket Pond paralleling half an IOD. Thus, up to 16-sockets can directly connect to an EMC. Pool sizes of 32-64 would combine CXL switches with Pond's multi-headed EMC. The optimal design point balances the potential pool savings for larger pool sizes (§6) with the added cost of larger EMCs and switches.

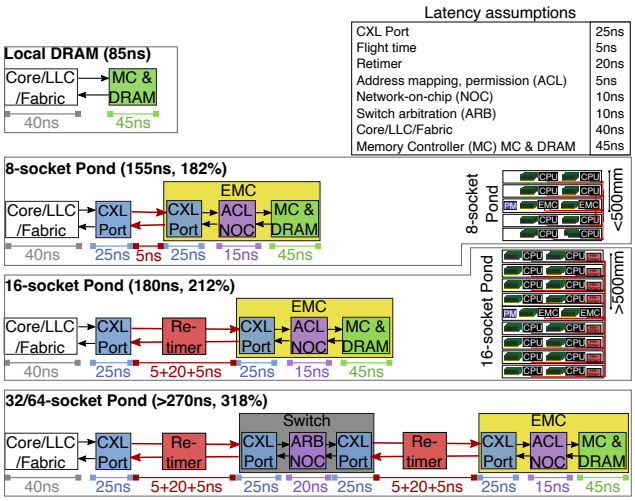


Figure 7: Pool size and latency tradeoffs (§4.1). *Small Pond pools of 8-16 sockets add only 75-90ns relative to NUMA-local DRAM. Latency increases for larger pools that require retimers and a switch.*

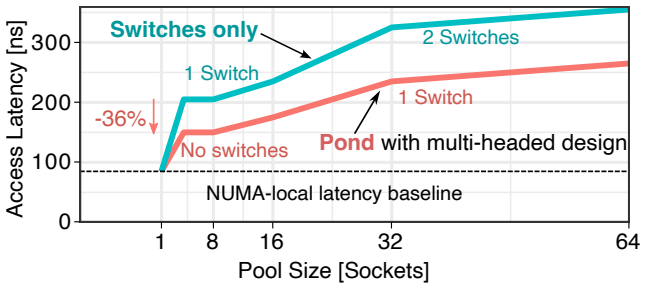


Figure 8: Pool access latency comparison (§4.1). *Pond reduces latencies by 1/3 compared to switch-only designs.*

EMC Latency. While latency is affected by propagation delays, it is dominated by CXL port latency, and any use of CXL retimers and CXL switches. Port latencies are discussed in §2 and [69]. Retimers are devices used to maintain CXL/PCIe signal integrity over longer distances and add about 10ns of latency in each direction [17, 18]. In datacenter conditions, signal integrity simulations [39] indicate that CXL could require retimers above 500mm. Switches add at least 70ns of latency due to ports/arbitration/NOC with estimates above 100ns [61].

Figure 7 breaks down Pond's latency for different pool sizes. Figure 8 compares Pond's latency to a design that relies only on switches instead of a multi-headed EMC. We find that Pond reduces latencies by 1/3 with 8-and 16-socket pools adding only 70-90ns relative to NUMA-local DRAM. In practice, we expect Pond to be deployed primarily with small 8/16-socket pools, given the latency and cost overheads, and diminishing returns of larger pools (§3). Modern CPUs can connect to multiple EMCs which allows scaling to meet bandwidth and capacity goals for different clusters.

4.2 System Software Layer

Pond's system software involves multiple components.

Pool memory ownership. Pool management involves assigning Pond's memory slices to hosts and reclaiming them for the pool (Figure 9). It involves 1) implementing the control paths for pool-level memory assignment and 2) preventing pool memory fragmentation.

Hosts discover local and pool capacity through CXL device discovery and map them to their address space. Once mapped, the pool address range is marked hot-pluggable and "not enabled." Slice assignment is controlled at runtime via a Pool Manager (PM) that is colocated on the same blade as the EMCs (Figure 7). In Pond's current design, the PM is connected to EMCs and CPU sockets via a low-power management bus (*e.g.*, [20]). To allocate pool memory, the Pool Manager triggers two types of interrupts at the EMC and host driver. `Add_capacity(host, slice)` interrupts the host driver which reads the address range to be hot-plugged. The driver then communicates with the OS memory manager to bring the memory online. The EMC adds the host id to its permission table at the slice offset. `Release_capacity(host, slice)` works similarly by offlining the slice on the host and resetting the slice's permission table entry on the EMC. An alternative to this design would be inband-communication using the Dynamic Capacity Device (DCD) feature in CXL 3.0 [28, §9.13]. This change would maintain the same functionality for Pond.

Pond must avoid fragmenting its online pool memory as the contiguous 1GB address range must be free before it can be offlined for reassignment to another host. Pool memory is allocated to VMs in 1GB-aligned increments (§4.3). While this prevents fragmentation due to VM starts and completions, our experience has shown that host agents and drivers can allocate pool memory and cause fragmentation. Pond thus uses a special-purpose memory partition that is only available to the hypervisor. Host agents and drivers allocate memory in host-local memory partition, which effectively contains fragmentation.

With these optimizations, offlining 1GB slices empirically takes 10-100 milliseconds/GB. Onlining memory is near instantaneous with microseconds/GB. These observations are reflected in Pond's asynchronous release strategy (§4.3).

Failure management. Hosts only interleave across local memory. This minimizes the EMCs' blast radius and facilitate memory hot-plugging. EMC failures affect only VMs with memory on that EMC, while VMs with memory on other EMCs continue normally. CPU/host failures are isolated and associated pool memory is reallocated to other hosts. Pool Manager failures prevent reallocating pool memory but do not affect the datapath.

Exposing pool memory to VMs. VMs that use both NUMA-local and pool memory see pool memory as a zNUMA node. The hypervisor creates a zNUMA node by adding a memory block (`node_memblk`) without an entry in the `node_cpuid` in the `SLIT/SRAT` tables [75]. We later show the guest-OS preferentially allocates memory from the local NUMA node before going to zNUMA (§6). Thus, if zNUMA is sized to the amount of untouched memory, it is never going to be used. Figure 10 shows a Linux VM which includes the correct latency in the NUMA distance matrix (`numa_slit`). This facilitates guest-OS NUMA-aware memory management [79, 81] for the rare case that the zNUMA is used (§4.4).

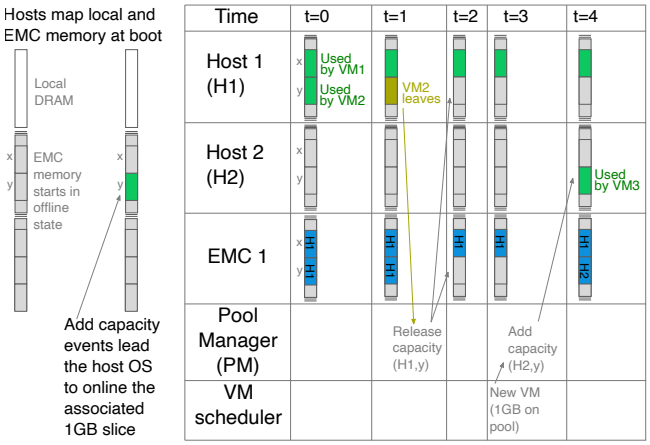


Figure 9: Pool management example (§4.2). *Pond assigns pool memory to at most one host at a time. This example shows Pond's asynchronous memory release strategy which engages when a VM departs (t=1 and t=2). During VM scheduling, memory is added to the corresponding host before the VM starts (t=3 and t=4).*

```
→  ~ sudo numactl --hardware
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11
node 0 size: 24124 MB
node 0 free: 23546 MB
node 1 cpus:
node 1 size: 8038 MB
node 1 free: 7999 MB
```

zNUMA

```
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

Figure 10: zNUMA (§4.2). *zNUMA seen from a Linux VM.*

Reconfiguration of memory allocation. To remain compatible with (**G2**), local and pool memory mapping generally remain static during a VM's lifetime. There are two exceptions that are implemented today. When live-migrating a VM or when remapping a page with a memory fault, the hypervisor temporarily disables virtualization acceleration and the VM falls back to a slower I/O path [67]. Both events are quick and transient and typically only happen once during a VM's lifetime. We implement a third variant which allows Pond a one-time correction to a suboptimal memory allocation. Specifically, if the host has local memory available, Pond disables the accelerator, copies all of the VM's memory to local memory and enables the accelerator again. This takes about 50ms for every GB of pool memory that Pond allocated to the VM.

Telemetry for opaque VMs. Pond requires two types of telemetry for VMs. First, we use the core-performance-measurement-unit (PMU) to gather hardware counters related to memory performance. Specifically, we use the top-down-method for analysis (TMA) [25, 80]. TMA characterizes how the core pipeline slots are used. For example, we use the "memory-bound" metric, which is defined as pipeline stalls due to memory loads and stores. Figure 12 lists these metrics. While TMA was developed for Intel, its relevant parts are available on AMD and ARM as well. We modify Azure's production hypervisor to associate these metrics with individual VMs (§5) and record VM counter samples in a distributed database. All our core-PMU-metrics use simple counters and induce negligible overhead (unlike event-based sampling [29, 65]).

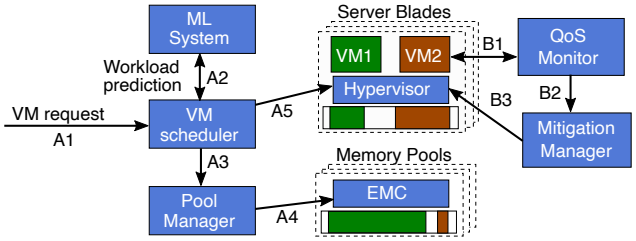


Figure 11: Pond control plane workflow (§4.3). A) *The VM scheduler uses ML-based predictions that identify latency-sensitive VMs and their likely amount of untouched memory to decide on VM placement (see Figure 13).* **B)** *The monitoring pipeline reconfigures VMs if quality-of-service (QoS) not met.*

Second, we use hypervisor telemetry to track a VM's untouched pages. We use an existing counter that tracks guest-committed memory, which overestimates used memory. This counter is available for 98% of Azure VMs. We also scan access bits in the hypervisor page table (§5). Since we only seek untouched pages, frequently access bits reset is not required. This minimizes overhead.

4.3 Distributed Control Plane Layer

Figure 11 shows the two tasks performed by Pond's control plane: (A) predictions to allocate memory during VM scheduling and (B) QoS monitoring and resolution.

Predictions and VM scheduling (A). Pond uses ML-based prediction models (§4.4) to decide how much pool memory to allocate for a VM. 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 the configuration bus to the EMCs and host (A4). Memory onlining is fast enough to not block a VM's start time (§4.2). The scheduler informs the hypervisor to start the VM on a zNUMA node matching the onlined memory amount.

Memory offlining is slow and cannot happen on the critical path of VM starts (§4.2). Pond resolves this by always keeping a buffer of unallocated pool memory. This buffer is replenished when VMs terminate and hosts asynchronously release associated slices.

QoS monitoring (B). Pond continuously inspects the performance of all running VMs via its QoS monitor. The monitor queries hypervisor and hardware performance counters (B1) and uses an ML model of latency sensitivity (§4.4) to decide whether the VM's performance impact exceeds the PDM. In this case, the monitor asks its mitigation manager (B2) to trigger a memory reconfiguration (§4.2) through the hypervisor (B3). After this reconfiguration, the VM uses only local memory.

4.4 Prediction Models

Pond's VM scheduling (A) and QoS monitoring (B) algorithms rely on two prediction models (in Figure 13).

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

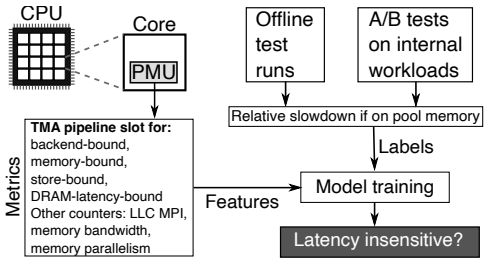


Figure 12: Pond latency insensitive model (§4.2). *This model uses metrics from the core's performance-measurement-unit (PMU). It is trained with labels gathered from offline runs and internal workloads.*

metadata as the request VM, *e.g.*, the customer-id, VM type, and location. This is based on the observation that VMs from the same customer tend to exhibit similar behavior [41].

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 untouched memory (UM) over its lifetime. Interestingly, UM predictions with only generic VM metadata such as customer history, VM type, guest OS, and location are accurate (§6). VMs without untouched memory (UM = 0) are allocated entirely with local DRAM. VMs with a UM > 0 are allocated with a rounded-down GB-aligned percentage of pool memory and a corresponding zNUMA node; the remaining memory is allocated on local DRAM.

If we underpredict UM, the VM will not touch the slower pool memory as the guest OS prioritizes allocating local DRAM. If we overpredict UM, we rely on the QoS monitor for mitigation. Importantly, Pond always keeps 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, Pond monitors if it overpredicted the amount of untouched memory during scheduling. For pool-backed VMs and zNUMA VMs with less untouched 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. Pond's two ML prediction models consume telemetry that is available for opaque VMs from Pond's system software layer (§4.2). Figure 12 shows features, labels, and the training procedure for the latency insensitivity model. The model uses supervised learning (§5) with core-PMU metrics as features and the slowdown of pool memory relative to NUMA-local memory as labels. Pond gets samples of slowdowns from offline test runs and A/B tests of internal workloads which make their performance numbers available. These feature-label-pairs are used to retrain the model daily. As the core-PMU is lightweight (§5), Pond continuously measures core-PMU metrics at VM runtime. This enable the QoS monitor to react quickly and enables retaining a history of VMs that have been latency sensitive.

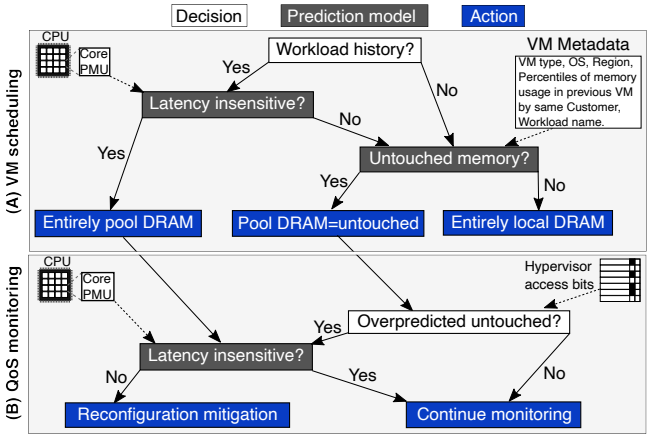


Figure 13: Pond prediction models (§4.4). *Pond's two prediction models (dark grey) rely on telemetry (blue boxes) that is available for all VM types, including third-party opaque VMs.*

Figure 14 shows the inputs and training procedure for the untouched-memory model. The model uses supervised learning (details in §5) with VM metadata as features and the minimum untouched memory over each VM's lifetime as labels. Its most important feature is a range of percentiles (*e.g.*, 80^{th}–99^{th}) of the recorded untouched memory by a customer's VMs in the last week.

Parameterization of prediction models. Pond's 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 rate of 0.1% FP may force the model to 5% of LI.

Similarly, Pond's untouched memory model is parameterized to stay below a *target rate of overpredictions* (OP), *i.e.*, workloads that touch more memory than predicted and thus would use memory pages on the zNUMA node. This parameter enforces a tradeoff as the percentage of untouched memory (UM) is a function of OP. For example, a rate of 0.1% OP may force the model to 3% of UM.

With two models and their respective parameters, Pond needs to decide how to balance FP and OP between the two models. This balance is done by solving an optimization problem based on the given performance degradation margin (PDM) and the target percentage of VMs that meet this margin (TP). Specifically, Pond seeks to maximize the average amount of memory that is allocated on the CXL pool, which is defined by LI and UM, while keeping the percentage of false positives (FP) and untouched overpredictions (OP) below the TP.

$$\begin{aligned} &\text{maximize} \quad (\mathtt{LI}_{\mathtt{PDM}}) + (\mathtt{UM}) \\ &\text{subject to } (\mathtt{FP}_{\mathtt{PDM}}) + (\mathtt{OP}) \leq (100 - \mathtt{TP}) \end{aligned} \tag{1}$$

Note that TP essentially defines how often the QoS monitor has to engage and initiate memory reconfigurations.

Besides PDM and TP, Pond has no other parameters as it automatically solves the optimization problem from Eq.(1). The models rely on their respective framework's default hyperparameters (§5).

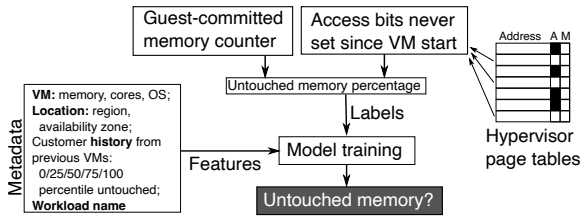
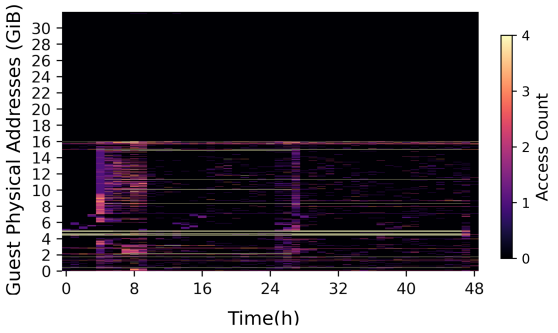


Figure 14: Training of the untouched memory model (§4.4). *This model uses VM metadata as features and labels of untouched memory gathered from hypervisor telemetry.*



Workloads	Traffic to zNUMA
Video	0.25%
Database	0.06%
KV store	0.11%
Analytics	0.38%

Figure 15: Effectiveness of zNUMA (§6.2). *Latency sensitive workloads get a local vNUMA node large enough to cover the workload's footprint. zNUMA nodes holds the VM's remaining memory on Pond CXL pool. Access bit scans, e.g., for Video (right), show that this configuration indeed minimizes traffic to the zNUMA node.*

5 IMPLEMENTATION

We implement and evaluate Pond on production servers that emulate pool latency.

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 Pond's models. We extend each virtual core's metadata with a copy of its core-PMU state and transfer this state when it gets scheduled on different physical cores. Pond samples core-PMU counters once per second, which takes 1ms. We enable access bit scanning in hypervisor page tables. We scan and reset access bits every 30 minutes, which takes 10s.

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 [63] to classify whether a workload exceeds the PDM. The model uses a set of 200 hardware counters as supported by current Intel processors. The untouched memory model uses a gradient boosted regression model (GBM) from LightGBM [49] and makes a quantile regression prediction with a configurable target percentile. After exporting to ONNX [62], the prototype adds the prediction (the size of zNUMA) on the VM request path using a custom inference serving system similar to [42, 43]. Azure's VM scheduler incorporates zNUMA requests and pool memory as an additional dimension into its bin packing, similar to other cluster schedulers [37, 48].

6 EVALUATION

Our evaluation addresses the performance of zNUMA VMs (§6.2, §6.3), the accuracy of Pond's prediction models (§6.4), and Pond's end-to-end DRAM savings (§6.5).

6.1 Experimental Setup

We evaluate the performance of our prototype using 158 cloud workloads. Specifically, our workloads span in-memory databases and KV-stores (Redis [22], VoltDB [27], and TPC-H on MySQL [26]), data and graph processing (Spark [19] and GAPBS [33]), HPC (SPLASH2x [82]), CPU and shared-memory benchmarks (SPEC CPU [24] and PARSEC [35]), and a range of Azure's internal workloads (Proprietary). Figure 4 overviews these workloads. We quantify DRAM savings with simulations.

Prototype setup. We run experiments on production servers at Azure and similarly-configured lab servers. The production servers use either two Intel Skylake 8157M sockets with each 384GB of DDR4, or two AMD EPYC 7452 sockets with each 512GB of DDR4. On Intel, we measure 78ns NUMA-local latency and 80GB/s bandwidth and 142ns remote latency and 30GB/s bandwidth (3/4 of a CXL ×8 link). On AMD, we measure 115ns NUMA-local latency and 255ns remote latency. Our BIOS disables hyper-threading, turbo-boost, and C-states.

We use performance results of VMs entirely backed by NUMA-local DRAM as our *baseline*. We present zNUMA performance as normalized slowdowns, *i.e.*, the ratio to the baseline. Performance metrics are workload specific, *e.g.*, job runtime, throughput and 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 16). With at least three repetitions of each run and 158 workloads, our evaluation spans more than 3,500 experiments and 10,000 machine hours. Most experiments used lab servers; we spot check outliers on production servers.

Simulations. 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.*,

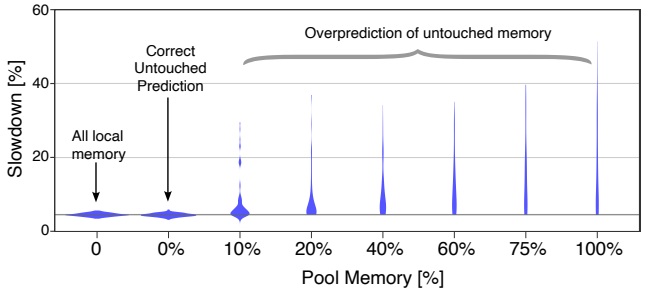


Figure 16: Slowdown under different pool allocations (§6.3). *Performance for no pool memory and a correctly-sized zNUMA is comparable. Small slowdowns arise from run-to-run variation. Slowdown become noticeable as soon as the workload spills into the zNUMA and steadily increases until the whole workload is allocated on pool memory (100% spilled).*

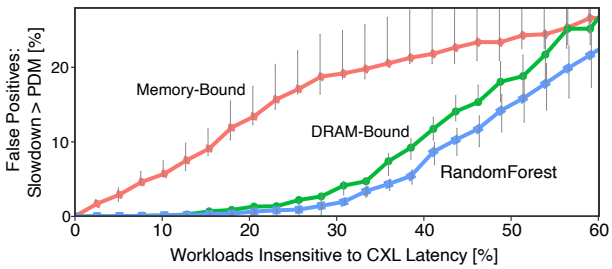


Figure 17: Latency insensitivity model (§6.4). *As we increase how many workloads are marked as insensitive (LI), the rate of false positives (FP) increases. Pond's RandomForest slightly outperforms a heuristic based only on the DRAM-bound TMA performance counter.*

due to insufficient pool memory, the simulator moves the VMs to another server.

Model evaluation. We evaluate our model with production resource logs. About 80% of VMs have sufficient history to make a sensitivity prediction. Our deployment does not report each workload's perceived performance (opaque VMs). We thus evaluate latency sensitivity model based on our 158 workloads.

6.2 zNUMA VMs on Production Nodes

We perform a small-scale experiment on Azure production nodes to validate 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. To see the effectiveness of zNUMA, we assume a correct prediction of untouched memory, *i.e.*, the local footprint fits into the VM's local vNUMA node. Figure 15 shows access bit scans over 48 hours from the video workload and a table that shows the traffic to the zNUMA node for the four workloads.

Finding #1: We find that zNUMA nodes are effective at containing the memory access to the local vNUMA node. 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. Accesses within the local vNUMA node are spread out.

Implications. With a negligible fraction of memory accesses on zNUMA, we expect negligible performance impact given a correct prediction of untouched memory.

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 on all-local memory, a correctly sized zNUMA (0% spilled), differently-sized zNUMA nodes sized between 10-100% of the workload's footprint. Figure 16 shows a violin plot of associated slowdowns. This setup covers both normal behavior (all-local and 0% spill) and misprediction behavior for latency sensitive workloads. Thus, this is effectively a sensitivity study.

Finding #2: With a correct prediction of untouched memory, the distribution of workload slowdowns is similar to all-local memory.

Implications. This performance result is expected since the zNUMA node is rarely accessed (§6.2). Our evaluation can thus assume no performance impact under correct predictions of untouched memory (§6.5).

Finding #3: For overpredictions of untouched memory (and correspondingly undersized local vNUMA node), the workload spills into zNUMA. Many workloads see an immediate impact on slowdown. Slowdowns further increase if more workload memory spills into zNUMA. Some workloads are slowed down by up to 30-35% with 20-75% of workload memory spilled and up to 50% if entirely allocated on pool memory. We use access bit scans to verify that these workloads indeed actively access their entire working set.

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

6.4 Performance of Prediction Models

We evaluate Pond's prediction models (§4.4) and its combined prediction model based on Eq.(1).

6.4.1 Predicting Latency Sensitivity. Pond seeks 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% and on both 182% and 222% latency increases, but report details only for 5% and 182%. Other PDM values lead to qualitatively similar results. The 222% model is 16% less effective given the same false positive rate target. We compare thresholds on memory and DRAM boundedness [25, 80] to our RandomForest (§5).

Figure 17 shows the model's false positive rate as a function of the percentage of workloads labeled as latency insensitive, similar to a precision-recall curve [36]. Error bars show 99% confidence

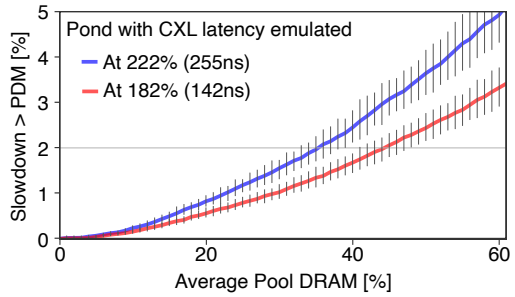


Figure 18: Combined model (§6.4). *Pond's overall tradeoff between average allocation of pool memory and mispredictions after solving Eq.(1).*

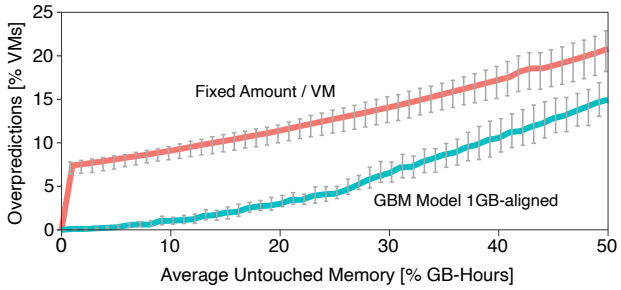


Figure 19: Untouched memory model (§6.4). *As we increase untouched memory (UM), our GBM has a significantly lower rate of overpredictions (OP) that a strawman model.*

from a 100-fold validation based on randomly splitting 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. 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. Heuristic as well as predictors will make statistical errors.

Finding #5: We find that "DRAM bound" significantly outperforms "Memory bound" (Figure 17). 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 Untouched Memory. Pond predicts the amount of untouched memory over a VM's future lifetime (§4.4). We evaluate this model using metadata and resource usage logs from 100 clusters over 75 days. The model is trained nightly and evaluated on the subsequent day. Figure 19 compares our GBM model to the heuristic that assumes a fixed fraction of memory as untouched across all VMs. The figure shows the overprediction rate as a function of the average amount of untouched memory. Figure 20 shows a production version of the untouched memory model during the first 110 days of 2022.

Finding #6: We find that the GBM model is 5× more accurate than the static policy, *e.g.*, when labeling 20% of memory as untouched, GBM overpredicts only 2.5% of VMs while the static policy overpredicts 12%.

Implication. Our prediction model identifies 25% of untouched memory while only overpredicting 4% of VMs.

Finding #7: The production version of our model performs similarly to the simulated model. Distributional shifts lead to some variability over time.

Implication. We find that accurately predicting untouched memory is practical and a realistic assumption.

6.4.3 Combined Prediction Models. We characterize Pond's combined models (Eq.(1)) using "scheduling mispredictions", *i.e.*, the fraction of VMs that will exceed the PDM. This incorporates the overpredictions of untouched memory, how much the model overpredicted, and the probability of this overprediction leading to a workload exceeding the PDM. Further, Pond uses its QoS monitor to mitigate up to 1% of mispredictions. Figure 18 shows scheduling mispredictions as a function of the average amount of cluster DRAM that is allocated on its pools for 182% and 222% memory latency increases, respectively.

Finding #8: Pond's combined model outperforms its individual models by finding their optimal combination.

Implication. With a 2% scheduling misprediction target, Pond can schedule 44% and 35% of DRAM on pools with 182% and 222% memory latency increases, respectively.

6.5 End-to-end Reduction in Stranding

We characterize Pond's end-to-end performance while constraining its rate of scheduling mispredictions. Figure 21 shows the reduction in aggregate cluster memory as a function of pool size for Pond under 182% and 222% memory latency increase, respectively, and a strawman static allocation policy. We evaluate multiple scenarios; the figure shows PDM =5% and TP =98%. In this scenario, the strawman statically allocates each VM with 15% of pool DRAM. About 10% of VMs would touch the pool DRAM (Figure 19). Of those touching pool DRAM, we'd expect that about $\frac{1}{4}$ would see a slowdown exceeding a PDM =5% (Figure 16). So, the strawman would have about 2.5% of scheduling mispredictions.

Finding #9: At a pool size of 16 sockets, Pond reduces overall DRAM requirements by 9% and 7% under 182% and 222% latency increases, respectively. Static reduces DRAM by 3%. When varying PDM between 1 and 10% and TP between 90 and 99.9% we find the relative savings of the three systems to be qualitatively similar.

Implication. Pond can safely reduce cost. A QoS monitor that mitigates more than 1% of mispredictions, can achieve more aggressive performance targets (PDM).

Finding #10: Throughout the simulations, Pond's pool memory offlining speeds remain below 1GB/s and 10GB/s for 99.99% and

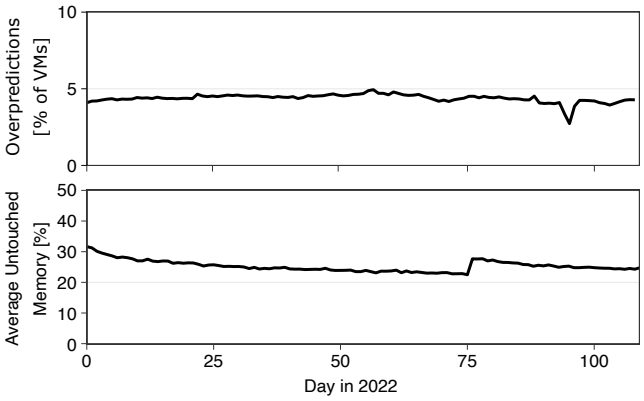


Figure 20: Untouched memory ML model performance in production (§6.4). *Our production model targets 4% overpredictions (OP). It average untouched memory percentage is similar to the simulated model (Figure 19).*

99.999% of VM starts, respectively.

Implication. Pond is practical and achieve design goals.

7 DISCUSSION

Robustness of ML. Similar to other oversubscribed resources (CPU [77] and disks [13]), customers may overuse resources to get local memory. When multiplexing resource for millions of customers, any individual customer's behavior will have a small impact. Providers can also provide small discounts when resources are not fully utilized.

Alternatives to static memory preallocation. Pond is designed for compatibility static memory as potential workarounds are not yet practical. The PCIe Address Translation Service (ATS/PRI) [1] enables compatibility with page faults. Unfortunately, ATS/PRI-devices are not yet broadly available [55]. Virtual IOMMUs [31, 34, 74] allow fine-grained pinning but require guest OS changes and introduce overhead.

8 RELATED WORK

Hardware-level disaggregation: Hardware-based disaggregation designs [47, 64] are not easily deployable as they do not rely on commodity hardware. For instance, ThymesisFlow [64] and Clio [47] propose FPGA-based rack-scale memory disaggregation designs on top of OpenCAPI [21] and RDMA. Their hardware layer shares goals with Pond. Their software goals differ fundamentally, *e.g.*, ThymesisFlow advocates application changes for performance, while Pond focuses on platform-level ML-driven pool memory management that is transparent to users.

Hypervisor/OS level disaggregation: Hypervisor/OS level approaches [46, 51, 68] 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/application level disaggregation: Runtime-based disaggregation designs [38, 66, 76] propose customized APIs for remote

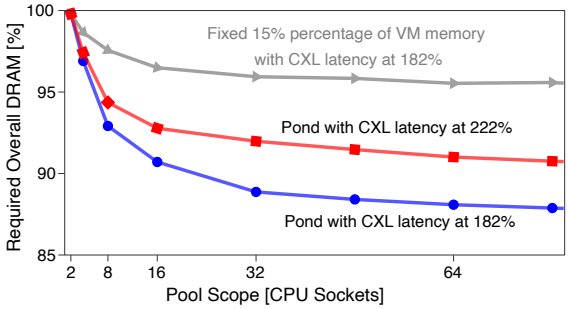


Figure 21: Memory savings under performance constraints (§6.5). *Simulated end-to-end evaluation of memory savings achieved by Pond under PDM =5% and scheduling mispredictions TP =98%.*

memory access. 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 [50, 51]. For example, Google achieves 6µs latency via proactive hot/cold page detection and compression [9, 51]. Nimble [79] optimizes Linux's page tracking mechanism to tier pages for increased migration bandwidth. Pond takes a different ML-based approach looking at memory pooling design at the platform-level and is orthogonal to these works.

ML for systems: ML is increasingly applied to tackle systems problems, such as cloud efficiency [41, 77], memory/storage optimizations [58, 84], microservices [83], caching/prefetching policies [71, 72]. We uniquely apply ML methods for untouched memory prediction to support pooled memory provisioning to VMs without jeopardizing QoS.

Coherent memory and NUMA optimizations: Traditional cache coherent NUMA architectures [53] use specialized interconnects to implement a shared address space. There are also system-level optimizations for NUMA, such as NUMA-aware data placement [54] and proactive page migration [81]. NUMA scheduling policies [40] balance compute and memory across NUMA nodes. Pond's ownership overcomes the need for coherence across the memory pool. zNUMA's zero-core nature requires rethinking of existing optimizations which are largely optimized for symmetric NUMA systems.

9 CONCLUSION

DRAM costs are an increasing cost factor for cloud providers. This paper is motivated by the observation of stranded and untouched memory across 100 production cloud clusters. We proposed Pond, the first full-stack memory pool that satisfies the requirements of cloud providers. Pond consists of contributions at the hardware, system software, and distributed system layers to manage pooled/CXL memory. Our results showed that Pond can reduce the amount of needed DRAM by 7% with a pool size of 16 sockets and assuming CXL increases latency by 222%. This translates into an overall reduction of 3.5% in cloud server cost.

10 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their tremendous feedback and comments which help improve the quality of the paper. The work was mainly done when Huaicheng was affiliated with CMU Parallel Data Lab (PDL) and we thank the member companies of the PDL Consortium for their support.

REFERENCES

[1] 2009. PCI Express Address Translation Services. https://composter.com.ua/documents/ats_r1.1_26Jan09.pdf.
[2] 2015. 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.
[3] 2017. AMD EPYC brings new RAS capability. https://www.amd.com/system/files/2017-06/AMD-EPYC-Brings-New-RAS-Capability.pdf.
[4] 2019. Single-Root Input/Output Virtualization. httpp://www.pcisig.com/specifications/iov/single_root.
[5] 2020. Compute Express Link Specification. Available at https://www.computeexpresslink.org.
[6] 2020. CXL 2.0 Specification. https://www.computeexpresslink.org/download-the-specification.
[7] 2020. 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/.
[8] 2020. Intel Virtualization Technology for Directed I/O. https://software.intel.com/content/dam/develop/external/us/en/documents/vt-directed-io-spec.pdf.
[9] 2020. Linux Memory Management Documentation - zswap. https://www.kernel.org/doc/html/latest/vm/zswap.html.
[10] 2021. AMD EPYC Genoa and SP5 Platform Leaked. https://wccftech.com/amd-epyc-genoa-zen-4-server-cpus-and-sp5-lga-6096-server-platform-details-leaked/ accessed 8/23/22.
[11] 2021. 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.
[12] 2021. AWS: Enhanced Networking Support. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.
[13] 2021. AWS: Optimize Disk Performance for Instance Store Volumes. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/disk-performance.html.
[14] 2021. Azure Accelerated Networking: Supported VM Instances. https://docs.microsoft.com/en-us/azure/virtual-network/accelerated-networking-overview.
[15] 2021. Compute Express Link 2.0 White Paper. https://b373eaf2-67af-4a29-b28c-3aae9e644f30.filesusr.com/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf.
[16] 2021. CXL Consortium Member Spotlight: Arm. https://www.computeexpresslink.org/post/cxl-consortium-member-spotlight-arm.
[17] 2021. 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.
[18] 2021. 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/.
[19] 2021. HiBench: The Bigdata Micro Benchmark Suite. https://github.com/Intel-bigdata/HiBench.
[20] 2021. MIPI I3C Bus Sensor Specification. https://www.mipi.org/specifications/i3c-sensor-specification.
[21] 2021. OpenCAPI Consortium. https://opencapi.org/.
[22] 2021. Redis. https://redis.io.
[23] 2021. Sapphire Rapids Uncovered: 56 Cores, 64GB HBM2E, Multi-Chip Design. https://www.tomshardware.com/news/intel-sapphire-rapids-xeon-scalable-specifications-and-features.
[24] 2021. SPEC CPU 2017. https://www.spec.org/cpu2017.
[25] 2021. 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.
[26] 2021. TPC-H Benchmark. http://www.tpc.org/tpch.
[27] 2021. VoltDB. https://www.voltdb.com.
[28] 2022. CXL 3.0 Specification. https://www.computeexpresslink.org/download-the-specification.
[29] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative Evaluation of Intel PEBS Overhead for Online System-noise Analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*.
[30] 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. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[31] Nadav Amit, Muli Ben-Yehuda, IBM Research, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU Emulation. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*.
[32] Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and Bruce Shepherd. 2013. Tight Bounds for Online Vector Bin Packing. In *Proceedings of the 45th ACM Symposium on Theory of Computing (STOC)*.
[33] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. *arXiv:1508.03619* (2015).
[34] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[35] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[36] Michael Buckland and Fredric Gey. 1994. The Relationship Between Recall and Precision. *Journal of the American Society for Information Science* 45, 1 (1994).
[37] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016).
[38] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[39] Elene Chobanyan, Casey Morrison, and Pegah Alavi. 2020. End-to-End System-Level Simulations with Retimers for PCIe Gen5 & CXL. DesignCon, slides available at https://www.asteralabs.com/wp-content/themes/astera-labs/images/retimer-cxl.pdf.
[40] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/.
[41] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. 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)*.
[42] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. 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)*.
[43] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[44] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, , and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*.
[45] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[46] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[47] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. 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)*.
[48] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[49] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A Highly Efficient Gradient Boosting Decision Tree. *Advances in Neural Information Processing Systems (NIPS)* (2017).
[50] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*.
[51] 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. 2019. 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)*.
[52] Christopher Lameter. 2019. 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.

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

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

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

[56] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. 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)*.

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

[58] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. 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)*.

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

[60] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. *arXiv:2206.02878* (2022).

[61] Timothy Prickett Morgan. 2021. PCI-Express 5.0: The Unintended but formidable datacenter interconnect. DesignCon, slides available at https://www.nextplatform.com/2021/02/03/pci-express-5-0-the-unintended-but-formidable-datacenter-interconnect/.

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

[63] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research (JMLR)* 12, 85 (2011).

[64] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and Peter Hofstee. 2020. 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)*.

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

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

[67] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. 2018. VM Live Migration at Scale. *ACM SIGPLAN Notices* 53, 3 (2018).

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

[69] Debendra Das Sharma. 2022. Compute Express Link: An Open Industry-standard Interconnect Enabling Heterogenous Data-centric Computing. In *Proceedings of the 29th IEEE Hot Interconnects symposium (HotI29)*.

[70] Debendra Das Sharma. 2022. CXL 3.0: New Features for Increased Scale and Optimized Resource Utilization. In *Flash Memory Summit*.

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

[72] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. 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)*.

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

[74] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. 2020. 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)*.

[75] UEFI. 2021. Advanced Configuration and Power Interface Specification.

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

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

[78] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[79] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. 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)*.

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

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

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

[83] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. 2021. 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)*.

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

Received 2022-07-07; accepted 2022-09-22