A Case Against CXL Memory Pooling

Philip Levis
Google
plevis@google.com

Kun Lin
Google
linkun@google.com

Amy Tai
Google
amytai@google.com

Abstract

Compute Express Link (CXL) is a replacement for PCIe. With much lower latency than PCIe and hardware support for cache coherence, programs can efficiently access remote memory over CXL. These capabilities have opened the possibility of CXL memory pools in datacenter and cloud networks, consisting of a large pool of memory that multiple machines share. Recent work argues memory pools could reduce memory needs and datacenter costs.

In this paper, we argue that three problems preclude CXL memory pools from being useful or promising: cost, complexity, and utility. The cost of a CXL pool will outweigh any savings from reducing RAM. CXL has substantially higher latency than main memory, enough so that using it will require substantial rewriting of network applications in complex ways. Finally, from analyzing two production traces from Google and Azure Cloud, we find that modern servers are large relative to most VMs; even simple VM packing algorithms strand little memory, undermining the main incentive behind pooling.

Despite recent research interest, as long as these three properties hold, CXL memory pools are unlikely to be a useful technology for datacenter or cloud systems.

CCS Concepts

• **Networks** → **Data center networks**; • **Information systems** → *Enterprise resource planning*.

Keywords

datacenter networking, CXL memory pooling

ACM Reference Format:
Philip Levis, Kun Lin, and Amy Tai. 2023. A Case Against CXL Memory Pooling. In *The 22nd ACM Workshop on Hot Topics in Networks (HotNets '23), November 28–29, 2023, Cambridge, MA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626111.3628195

1 Introduction

Memory is an expensive component of datacenter and cloud servers: recent papers report its fraction of a server's cost is 40% for Meta [14] and 50% for Azure [21]. Google faces similar pressures [6]. The pressure to reduce RAM needs and costs has motivated work in far memory [18], memory compression [12], and Intel Optane memory, which trades off performance for lower cost [17]. If a server has insufficient memory, it can have free cores but no available memory (stranded cores); if it has too much memory it can have free memory that cores do not use (stranded memory).

One approach to reduce RAM costs is to disaggregate memory through a shared pool. In this model, servers have their own local RAM, which is sufficient for average or expected use. If a server needs more memory or has stranded cores, it can allocate from a pool shared among several servers. A memory pool needs to solve two major problems: latency and cache coherence. Main memory in a larger server CPU has a latency of 120-140ns; if a memory pool's latency is much higher, application performance will suffer.

The Compute Express Link (CXL) protocol promises to provide low-latency, cache coherent access to remote memory. With claimed latencies in the hundreds of nanoseconds, CXL can build a large memory pool shared across several servers. Disaggregating storage from compute led to much more efficient and scalable datacenter storage [7]; disaggregating memory from compute could have a similar impact, enabling more efficient and lower cost computing.

Unfortunately, this paper argues that CXL memory pooling faces three major problems. Each of these problems, in isolation, might limit potential use cases but is surmountable. Together, however, they mean that CXL memory pools cost more, require rewriting software, and do not reduce resource stranding (e.g., unused memory).

The first problem is **cost**. The primary benefit of a CXL memory pool is reducing the aggregate RAM needs of datacenter and cloud systems. Today, servers are provisioned so they can keep all of their VMs or containers in memory even when all of them maximize their footprint simultaneously (a "sum-of-max" approach). Using a CXL pool can allow servers to instead provision for expected use, and when VMs uses their entire footprint the system can store cold data in a CXL pool. This cost calculation, however, ignores infrastructure costs. CXL requires a completely parallel network infrastructure to Ethernet, consisting of a top-of-rack (or top-of-N server) CXL appliance, with direct, alternative cabling to all of its servers.

The second problem is **software complexity**. Recent experimental results from real CXL hardware find that many of

CXL's latency claims are best-case estimates. For example, estimates in the Pond system are that CXL will add 70-90ns over same-NUMA-node DRAM. Recent experimental results, however, are that CXL adds 140ns for pointer chasing workloads [19]. This slowdown is for a directly-connected CXL memory device, not a shared pool, which adds switching, re-timers, and queueing. While loads and stores to a CXL device will be much slower than DRAM, hardware-accelerated copies of 8kB blocks are close to DRAM speed [20]. Therefore, achieving good performance involves rewriting software to explicitly manage CXL memory, copying blocks into local DRAM; this explicit, conditional, and pervasive memory management increases software complexity. Furthermore, maintaining multiple copies reduces CXL's memory savings.

The third problem is **limited utility**. The primary argument for CXL memory is that memory that would otherwise be stranded, i.e. memory that cannot be allocated to a VM because there are no more compute resources to support a VM, can now be pooled and used by other servers. However, after analyzing common server and VM shapes in a 2019 Google cluster trace [22] and 2020 Azure Cloud trace [9] using a methodology we developed to evaluate the conditions when memory pooling can improve stranding, we conclude pooling is rarely helpful. Modern servers are large (hundreds of cores and terabytes of RAM), and VMs are small enough, that VMs can be placed on a single server *with little stranding*. For the traces we examined, the ratio of VM to server sizes must increase by 32x (Google) and 8x (Azure) before pooling yields even modest efficiency gains. To the best of our knowledge, this is the first methodology for estimating the potential of memory pooling for resource packing.

In summary, as long as the cost, software complexity, and lack of utility properties hold, sharing a large DRAM bank between servers with CXL is a losing proposition. If one of these issues goes away – CXL is cheap, CXL is nearly as fast as main memory, or VM shapes become difficult to pack into servers – then CXL memory pools might prove to be useful.

2 CXL Memory Pools

This section explains Compute Express Link (CXL) and how CXL memory pools work. Readers familiar with CXL can skip this section. Because many of the details of CXL are extraneous to this paper, we gloss over them; an interested reader can consult the specifications [3–5].

2.1 CXL

In this paper, we focus on CXL 1.1, the first productized version. Samsung [15], Intel [10], and Astera Labs [1] produce CXL 1.1 devices, but none of them are generally available; they are only for pilot use and commercial evaluation. Version 3.0 was published in August, 2022 [5].

CXL 1.1 uses the same physical layer (connections and signaling) as PCIe. PCIe connections consist of one or more parallel "lanes". CXL 1.1 uses PCIe Gen5 signaling, which provides 3.9GB/s per lane. CXL devices can have 1-32 lanes.

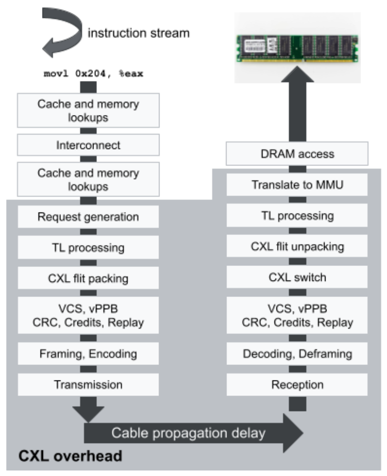

Figure 1: Processing path of a memory load (into a CXL Request) in a dual-socket server to a memory pool connected through. Queueing is possible at almost every transition. The response goes through the same path; lookups become updates.

A Samsung 128GB CXL memory device, for example, uses 8 lanes to support a maximum throughput of 35GB/s. [16] CXL 3.0 uses PCIe Gen6 to double per-lane throughput. PCIe Gen7 is expected to double throughput again (to 15GB/s), but this approaches the practical limit for a differential pair (224Gbps) due to gate switching latencies.

CXL differs from PCIe in two major ways: lower latency and cache coherence. CXL strips out many of the protocol overheads of PCIe to reduce latency. While PCIe Gen5 devices have best-case round-trip-time latencies of over 500ns, CXL devices can be as low as 150ns. This is the minimum *signaling* latency: it does not include the time for a device to generate a response (e.g., read from DRAM), any protocol processing, or queueing delays. While CXL 3.0 increases the throughput of CXL, it will not have lower latency [11] as packetization delay is not significant.[1]

CXL's second improvement is hardware cache coherence. This is useful for devices such as NICs or GPUs, but it is less important for memory pools, which typically do not allow servers to share memory.

2.2 Inside a CXL Pool

CXL memory pools can take many forms; in this section, we focus on the cloud use case of multiple servers connecting to a single device through separate physical links (called a "multi-headed device"), as proposed by Pond. [13] We assume the best-case use of a CXL pool, in which effectively all memory accesses are to memory that is exclusive to a single server, such that there are no cache coherence overheads.

Figure 1 decomposes the sources of latency when reading from a CXL memory pool. First, there are the standard memory latency costs: a core must detect that the memory is not in

[1]For a 16-lane device at 62GB/s, a 256 byte CXL flit takes 4ns.

any cache or local DRAM. In a dual-socket system (common in cloud servers today), the CXL device might be connected to either socket's PCIe/CXL lanes, so there is potentially the latency overhead of the CPU interconnect from one socket to the other. The processor's memory management unit (MMU) must transform a memory request into a CXL request. This enters the CXL root port, which dispatches it to the virtual switch (VCS) and virtual PCI-to-PCI bridge (vPPB) of the device; this dispatch is necessary because a port's many lanes can be allocated to many devices (e.g., 16 lanes can be allocated to 4 different 4-lane M.2 SSDs). The read request is packetized, encoded, and modulated onto the CXL link, adding packetization and propagation delays.

On reception, the CXL read request has to be reassembled from the parallel lanes, decoded and passed to the CXL memory controller. After protocol processing. the controller translates the request into DDR memory read requests. DDR reads are striped across multiple DDR sockets to maximize bandwidth. Once the data is assembled, the CXL device responds with a data response, which goes through a similar switching, processing, and encoding as the request did.

At every step of this process, there can be queueing. E.g., CXL read requests can queue at the client, memory read requests can queue at the device, DDR read requests can queue in the DDR memory controller, etc.

2.3 Pond, an example CXL Pool

Pond is a recent proposal for using a CXL memory pool to reduce RAM spending in cloud/VM systems [13]. Through extensive analysis of Azure workloads, the paper finds that a sizeable fraction of Azure memory is stranded: some VMs request a low RAM-to-CPU ratio, such that some servers have unused RAM but every core is in use. Pond argues that by moving a fraction of every server's memory to a CXL pool and statistically multiplexing the memory, a pool can reduce the total memory needed: memory that is unused today can instead be used by another server. The tradeoff is performance: since some VM memory is in the CXL pool, it is slower. Through careful prediction of which applications are latency sensitive and which pages are untouched or rarely touched, Pond can reduce overall DRAM requirements by 7–9% with only 2% of VMs seeing performance degrade by more than 5%.

2.4 The Case Against Memory Pools

We argue that, despite recent interest, CXL memory pools are not an effective way to provision networked servers. They seem like an exciting possibility and fruitful area of research, but this rosy picture is built on three mistaken and simplified assumptions: cost, complexity, and utility. The next three sections examine each issue in detail.

3 Cost

The first obstacle for CXL memory pools is their cost. On one hand, RAM is a large fraction of server cost, so a shared pool to meet peak needs while reducing per-server memory would reduce costs. We argue that such an analysis makes two assumptions that do not hold in practice. First, it assumes that memory is fungible and it is possible to cut a server's RAM by a small fraction (e.g., 7-9% in the Pond paper [13]). Second, it ignores the cost of an additional cabling and networking infrastructure.

3.1 Memory Provisioning

Cloud and datacenter servers are limited to discrete steps in DRAM capacity; small reductions in memory do not necessarily translate to cost savings. Modern server CPUs have 8 (Intel) or 12 (AMD) DDR channels. Because many applications are memory bandwidth bound, servers always populate every channel. DIMMs, however, only come in certain discrete sizes (e.g., 32GB, 48GB, 64GB)[2], and every channel must have the same sized DIMM. For example, a modern AMD Genoa CPU has 12 channels and 192 cores (384 vCPUs). A Genoa server can be configured with 750GB (64GB DIMMS), 1.15TB (96GB DIMMS), or 1.5TB (128GB DIMMS), but no intervening values.

Pond finds that allocating 25% of VM memory (on average) to a shared pool leads to only small slowdowns (1% of VMs slow down by more than 5%). This is achievable, e.g., by replacing 128GB DIMMs with 96GB DIMMS. While achievable, allocating 25% of memory to a pool does not reduce the amount of memory. The servers still need the same amount of memory, just some of it is in a CXL-connected memory appliance.

More importantly, Pond also finds that a CXL memory pool can reduce total RAM by 7-9% without significantly harming performance. Some VM memory is unused, and by clustering many servers worth of VMs together, Pond can aggregate these savings. However, as one cannot shrink server RAM by 7% or 9%, servers must cut their RAM by 25%, and the memory pool takes the 7–9% out of this 25%. This, in turn, requires targeting a very specific amount of memory in the CXL pool device, which is difficult given the need to populate every socket to maximize throughput and the large jumps in DIMM size. There are some specific configurations where this can work out, but using them constrains the rest of the system to specific amounts of memory, numbers of cores, and degree of CXL pooling.

3.2 CXL Infrastructure

CXL memory pools are not free. We find that their costs exceed any savings from reducing RAM. When considering cost tradeoffs, we consider consumer (MSRP) prices. Cloud providers and hyperscalers receive steep discounts, but as we are considering relative costs and tradeoffs between components, we make a simplifying assumption that hyperscaler discounts are similar across high-volume parts.

[2]Today, sizes that are not a power of two, such as 48GB, are rare; we assume vendors would produce large numbers for a cloud provider if asked.

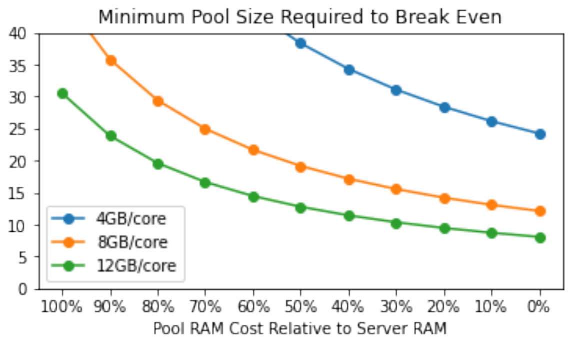

Figure 2: Minimum pool size for RAM cost savings to equal switch cost as pool RAM cost decreases relative to server RAM. Even if pool RAM is free, for a standard 4GB/core memory shape, the pool must be 24 nodes to break even with just the switch cost.

Because there are no CXL memory pool devices today, we do not know how much one costs. However, given the speeds and processing involved, we propose that an Ethernet switch is a good approximation. A CXL memory pool device is effectively a high-speed switch, processing CXL packets, managing cacheline state, reading and writing memory, and sending responses back to servers. A standard CXL memory device (e.g., a Astera Leo [1] or Intel device [10]) uses 16 lanes. At PCIe Gen5 speeds this is 480Gbps. A 16-server pool therefore processes data at 7.6Tbps.

A modern, low-end, 32-port 200Gbps Ethernet switch such as the Mellanox MSN3700-VS2F0 costs \$38,500. [2] DDR5 RAM today is $\approx$ 3\$/GB. For the CXL pool device to break even with its RAM savings, it must save 12.6TB of RAM. Assuming Pond's optimistic 9% reduction, to break even with just the switch, the servers must have $\frac{12.6TB}{0.09}$ = 140TB of RAM in aggregate (using Pond would reduce this to 127TB). For a 32-node pool, 127TB, means 4TB per server. A dual-socket AMD Genoa server, the standard next-generation system for cloud providers, has 384 vCPUs. At 4TB/server, there is $>$ 10GB of RAM per Genoa vCPU, more than high-RAM VMs provide. You have to buy considerably more RAM for Pond's RAM savings to pay for themselves: you are better off just buying less RAM.

What if pool RAM is cheaper than server RAM? E.g., it could be slower, more cost-efficient DIMMs, or DDR4. Figure 2 shows how pool RAM cost affects the minimum pool size to break even. These results assume the Genoa setup described above, reducing server RAM by 25% of RAM, and being able to reduce aggregate RAM by 9%. Even if pool RAM is completely free, for a standard 4GB/core memory shape, the pool must be 24 nodes to break even. For memory-optimized VMs (8GB/core), if the pool memory is half the cost of server memory (50%), a pool size of 20 could break even with the switch cost.

This accounting is only the capital expenditure of the pool device: it doesn't include the cost of the cabling, assembly, and maintenance to connect the servers to the pool, the cost of the interface cards that connect to the cables, the space costs of giving up rack slots to pools, or the energy costs of the pool devices. It also does not consider any operational expenditures for maintaining or managing this parallel network infrastructure. We conclude that the costs of introducing CXL devices into a datacenter network eclipse any cost savings of reducing RAM.

4 Software Complexity

The second major problem with a CXL memory pool is that it will significantly add to software complexity. Experimental results from real hardware show that, for random accesses, CXL devices are significantly slower than the best case numbers suggested in standards documents. While CXL has high latency, its high throughput means that transferring larger blocks of memory (a few kB) can be competitive with DRAM. This requires explicitly *copying* the remote memory into local memory; the CXL pool stops being memory accessed directly and instead becomes a far memory cache.

Today, CXL devices are not commercially available, and NDAs preclude publishing results without prior approval. The only CXL experimental results we are aware of are from a series of versions of a paper by authors from UIUC and Intel [19] (the Pond paper [13] assumes values reported in standards). Because we do not have an agreement with CXL device vendors, we base our conclusions on these published experimental results.[3]

4.1 CXL Performance

CXL memory devices are high throughput. They are typically 16-lane CXL devices; at PCIe Gen5 speeds, this is 480Gbps. A single DDR5-4800 DIMM (standard on new servers today) is 300Gbps. Server CPUs have many DIMMs, but 480Gbps is fast and it can support reasonable copy performance of larger objects. For example, a copy from CXL memory to local DDR by a single core can use 80% of the bandwidth of two DIMMs.

However, CXL memory devices are also high latency. The UIUC measurement results of a directly connected (no switching) CXL device show CXL loads have best case latencies of 2x of local memory, substantially slower than a remote NUMA node (1.5x) [19]. For a modern server CPU (e.g., Sapphire Rapids, as used in the paper), this means a memory access jumps from 140ns for local memory to 280ns for CXL memory. At 2.0-3.0 GHz in a multiple-issue superscalar processor, this latency stalls the CPU for over 500 instructions. Switched system with re-timers will have higher latency.

[3]The revision of the UIUC/Intel paper accepted to MICRO [19] reports results from multiple CXL devices, which vary greatly in performance. We focus on latencies from the highest performance device measured, CXL-A, which is an ASIC.

4.2 Instructions or Transfers?

While CXL will perform poorly as a transparent far memory, its throughput means it can read or write larger blocks with good performance. For example, early Intel/UIUC results show that synchronously copying 8kB from DRAM to CXL memory can have 80% the throughput of DRAM-to-DRAM copies when using the DSA memory copy accelerator. [20]

This involves explicitly *copying* CXL memory into local memory. In this model, CXL memory is a far memory cache, which processors can access faster than remote memory or storage, but which programs must explicitly copy from. This gets at a fundamental question with using CXL memory: how does a program access it?

Instruction-level loads and stores operate on a cache line granularity. Reasonable CXL performance, however, requires 8kB transfers. Unless an application can take an enormous performance hit when accessing CXL memory, it cannot do so transparently. Instead, it must do so explicitly, or the device must act as a page-level cache (e.g., a far memory RAMdisk partition).

Explicit copies require invasive changes to applications. For example, suppose a program calculates the maximum value over an array (tens of kB), and this array is in CXL memory. The loop is fast, consisting of only 4 instructions. At 2 instructions per clock, it can process 2 bytes every clock. At 3GHz, 280ns is 840 ticks, and the loops processes 1680 bytes in 280ns. A cache line is 64 bytes, so the processor must have over 26 prefetches in flight in order to keep the pipeline busy. Processors do not prefetch so deeply, so this loop will spend most of its time stalled on CXL reads.

In contrast, a program that explicitly copies from CXL memory into local memory will perform much faster, because it pays the 280ns latency once then operates on local, in-cache memory. However, the problem is that this requires an explicit memory copy to local memory. It requires rewriting programs to conditionally copy if data is in CXL; CXL memory is not transparent and requires pervasive changes to software. Furthermore, it requires making *copies* of data, which increases application memory use.

5 Limited Utility

In this section we develop a methodology for estimating the efficiency gains that can be recouped with memory pooling. Our primary efficiency metric is *utilization*, defined as $\frac{\text{used capacity}}{\text{total capacity}}$. The main argument for memory pooling is that it can improve utilization by reducing the amount of stranded memory, in other words reducing 'total capacity'.

Methodology. To approximate utilization improvement from memory pooling, we need to estimate a cluster's utilization. Utilization is a multi-dimensional bin-packing problem [8, 9, 23, 24], and to optimize efficiency, a datacenter cluster scheduler should pack VMs onto physical machines as tightly as possible. Of course, there are performance and isolation considerations when packing workloads as tightly as possible, so realistically, operators leave some fraction of headroom on each machine. Despite this, the optimal bin packing utilization of a machine is still a good proxy for the actual utilization an operator can achieve in a real deployment; later in this section, we validate that the optimal packing approximates an event-driven packer within reasonable error bounds. Crucially, using the optimal utilization does not pin efficiency gains to a specific cluster scheduler implementation and enables faster analysis than re-running a full cluster trace. Therefore, we define a cluster's utilization as its utilization under the optimal packing of a VM workload on the cluster.

To determine the efficiency gain of pooling, we calculate the utilization improvement from adding a pool to a cluster. We model a cluster as a set of machines. We model pooling as a large machine that is N-times the size of a machine. N reflects the number of machines that share a CXL pool. Note that modelling pooling as a large machine *overestimates* the utilization gain from pooling, because the large machine elides allocation boundaries. In a true pooling setup compute resources must still be allocated to the machine boundary, and memory resources must respect a machine and pool boundary. Therefore any improvement due to pooling in this section is a generous *upper bound* on what pooling can realistically achieve.

Datacenter traces. For VM workloads and machine sizes that are representative of real deployments, we analyze two cluster traces, the 2019 Google cluster trace and 2020 Azure Trace for Packing [9]. From each trace we derive a distribution of VM demand and realistic machine sizes [22]. In the Google trace, we use VMs and machines from Cell A and only include VMs with production priority. In the Azure trace, we only include VMs with high priority. Note that the Google trace is a trace of *internal* workloads, and the Azure trace is for cloud, or customer, VMs; we analyze both traces to see if the two different settings affect the impact of pooling. We model both machines and VMs as two-dimensional vectors of CPU and memory.

Optimal packing. We use a vector bin-packing library to calculate the optimal packing of each set of VMs. The library takes a set of VMs and a machine size, and returns the *minimum* number of machines it takes to pack the entire set. Therefore, from a utilization perspective, 'used capacity' is always fixed, because we must land all VMs, but 'total capacity' is variable, because it depends how optimally the VMs can be packed on the given machine sizes.

Our optimal packing packs a snapshot of the cluster, which is a simpler problem than the packing problem in a real cluster scheduler, because the snapshot is not bound to previous placement decisions and does not take VM departures and arrivals into consideration. We packed many snapshots and studied the result distribution; all snapshots had trends similar to Figures 3 and 4.

Validation. We validate that a cluster's utilization under optimal packing is close to the utilization under a reasonable, live cluster scheduler. We implement a greedy bin packer that replays a cluster trace and compare the cluster's utilization

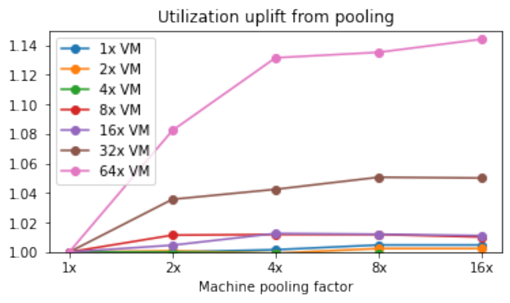

Figure 3: Google cluster trace: Pooling resources across up to 16 machines yields does not yield utilization improvements until VMs are at least 32x larger.

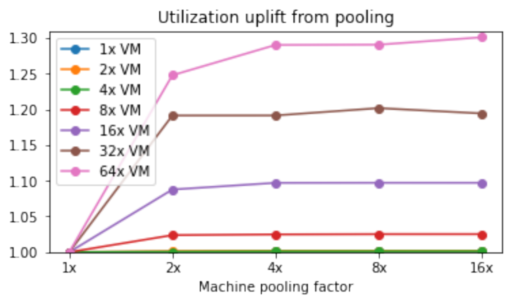

Figure 4: Azure trace: Pooling resources across up to 16 machines does not yield utilization improvements until VMs are at least 8x larger. These (cloud) VMs are much larger than the VMs in the Google trace, but still not large enough for pooling to matter.

after running the trace to the utilization of the optimal packing, at the snapshot of the cluster after the trace has been played. We do this comparison for subsets of traces across all 8 cells available in the 2019 Google trace and find that optimal packing utilization is 0-17% better than the utilization of the greedy bin packer, with a median difference of 5%. This greedy bin packer is likely worse at packing workloads than production-grade cluster schedulers, which means that the optimal packing can even more closely approximate the utilization of production-grade schedulers. These error bounds suggest that the optimal packing is a reliable proxy for cluster utilization in the following results.

Results. Figures 3 and 4 show the results, where utilization is normalized to a 1x machine pooling factor and 1x VM sizes. Taking the unmodified VM demands from the trace resulted in no utilization gain from pooling of any size (flat blue line) in both datasets.

To see how sensitive this result is and how much packing flexibility there is, we inflate the sizes of VMs. For example, for 8x we increase the core count and memory size of every VM by a factor of 8. For the Google trace, we find that pooling has at most 1% utilization improvement, even when VMs are inflated up to 16x. Weighed against the additional costs and complexity of pooling outlined earlier in the paper, this small improvement renders pooling out of the question. In Figure 3, pooling begins to have a benefit with a 32x inflation factor, and even then, it is modest, less than 5%. VMs must be 64x larger in order for there to be significant resource stranding at a single server, which is what reduces stranding with pooling.

On the other hand, the Azure VMs (Figure 4) begin to see utilization improve at around 8x VM sizes, suggesting that the cloud VMs are larger than the internal Google VMs to begin with, and that the pooling calculus may differ for internal workloads versus public cloud workloads.

In general, if VMs can reach a certain size with respect to physical machine size, pooling can help with the resulting resource stranding. However, based on the analyzed traces, most VMs are small and servers are large: while bin-packing is an NP-hard problem, if the bins are large and almost all of the objects are small, there is little leftover space in any bin. As long as VM sizes remain small, pooling is untenable.

6 Discussion

Compute Express Link provides numerous improvements to PCIe, notably lower latency and cache coherence. For complex, latency-sensitive peripherals such as NICs and GPUs, CXL will allow much faster and more fine-grained coordination between the CPU and these processors.

This paper examines another potential use of CXL, disaggregation memory across servers through a shared CXL memory pool. While there has been a lot of excitement and interest in such an approach, there has been almost no experimental data to verify its feasibility. Furthermore, analyses of its potential benefits ignore many of the practical deployment issues and costs.

Disaggregation in datacenter and cloud systems was first proposed for hard disk drive storage, where seek times of milliseconds outweigh any additional system or network latency. Memory, however, is at the opposite of this spectrum. Architectures move memory *closer* to compute because locality is key to performance. GPUs, TPUs, and IPUs all have their own memory. For example, a 6 foot cable adds 12ns of propagation delay in each direction, and at memory speeds every such little increase matters.

In this paper, we described three reasons why CXL memory pools will not be useful in cloud and datacenter systems: cost, software complexity, and a lack of utility. Each reason is based on the best information we could find and is grounded in computing today. Future advances or marketplace shifts may invalidate our assumptions and change the calculus to make CXL pools attractive; that they could play a role as far memory RAMdisks, which an OS copies into local memory. We look forward to and encourage research on such a future, but at the same time do not want to mistake hopeful possibilities for technical reality.

References

[1] Astera Labs. Leo cxl memory connectivity platform. https://www.asteralabs.com/products/cxl-memory-platform/leo-cxl-memory-connectivity-platform/, 2023.

[2] CDW Corporation. Mellanox Spectrum-2 MSN3700 switch 32 ports. https://www.cdw.com/product/mellanox-spectrum-2-msn3700-switch-32-ports-managed-rack-mountable/6415759, 2023.
[3] Compute Express Link Consortium, Inc. Compute Express Link (CXL) Specification, Revision 1.1, 2019.
[4] Compute Express Link Consortium, Inc. Compute Express Link (CXL) Specification, Revision 2.0, 2020.
[5] Compute Express Link Consortium, Inc. Compute Express Link (CXL) Specification, Revision 3.0, 2022.
[6] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic, B. Morris, C. Mukherjee, J. Ren, G. Thelen, P. Turner, C. Villavieja, P. Ranganathan, and A. Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.
[7] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, page 249–264, USA, 2016. USENIX Association.
[8] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. ACM SIGCOMM Computer Communication Review, 44(4):455–466, 2014.
[9] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, et al. Protean: Vm allocation service at scale. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, pages 845–861, 2020.
[10] I. Intel. Intel FPGA Compute Express Link (CXL) IP. https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html, 2023.
[11] Ishwar Agarwal. CXL overview and evolution. In Proceedings of HotChips 34, 2022.
[12] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery.
[13] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.
[14] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.
[15] S. Newsroom. Samsung Electronics Introduces Industry's First 512GB CXL Memory Module. https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module, 2022.
[16] S. Newsroom. Samsung Develops Industry's First CXL DRAM Supporting CXL 2.0. https://news.samsung.com/global/samsung-develops-industrys-first-cxl-dram-supporting-cxl-2-0, 2023.
[17] I. B. Peng, M. B. Gokhale, and E. W. Green. System Evaluation of the Intel Optane Byte-Addressable NVM. In Proceedings of the International Symposium on Memory Systems, MEMSYS '19, page 304–315, New York, NY, USA, 2019. Association for Computing Machinery.
[18] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. Aifm: High-performance, application-integrated far memory. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, OSDI'20, USA, 2020. USENIX Association.
[19] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, I. Jeong, R. Wang, and N. S. Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, October 2023.
[20] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, I. Jeong, R. Wang, and N. S. Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, v1, March 2023.
[21] The Next Platform. 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.
[22] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: the Next Generation. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20), Heraklion, Greece, 2020. ACM.
[23] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In 2014 IEEE International Conference on Cluster Computing (CLUSTER), pages 48–56. IEEE, 2014.
[24] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In Proceedings of the Tenth European Conference on Computer Systems, pages 1–17, 2015.