Rethinking Applications' Address Space with CXL Shared Memory Pools

Tong Xing
The University of Edinburgh
Edinburgh, United Kingdom
tong.xing@ed.ac.uk

Antonio Barbalace
The University of Edinburgh
Edinburgh, United Kingdom
antonio.barbalace@ed.ac.uk

Abstract

Compute Express Link (CXL) is an emerging cache coherence interconnects standard that enables new forms of memory pooling and sharing in data centers. Today's data centers are built with many network-interconnected machines. While memory sharing could be implemented by software-managed coherence – like distributed shared memory (DSM), data centers prefer distributed computing as an alternative. Will the upcoming CXL 3.0 with hardware-managed coherence change the game?

To start answering this question, in this paper we compare the strengths and weaknesses of hardware-managed coherence and software-managed coherence in the context of CXL 3.0, showing that neither is generally optimal. We propose an Adaptive Coherence Management strategy at the Operating System (OS) level that dynamically selects the most suitable coherence management mechanism at runtime, based on an application memory access pattern, and CXL memory access latency. The proposed Adaptive Coherence Management rethinks today's application's address spaces, introducing variable coherence management mechanisms for different parts of an address space, at runtime. We envision such parts of the address space being larger than a page but likely smaller than a Virtual Memory Area – this paper discusses our initial design.

CCS Concepts

• **Hardware** → **Emerging architectures**; • **Software and its engineering** → **Memory management**.

Keywords

Compute eXpress Link, CXL, Cache Coherence, Distributed shared memory, Shared memory, Address Space, Operating System

ACM Reference Format:
Tong Xing and Antonio Barbalace. 2025. Rethinking Applications' Address Space with CXL Shared Memory Pools. In *4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25), March 30, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3723851.3723858

1 Introduction

We are living in a time of unprecedented computational demands driven by big-data analytics, artificial intelligence (AI), and high-performance computing (HPC). To meet these demands, modern data centers have moved from (a few) large monolithic servers to large clusters of commodity off-the-shelf (COTS) machines, orchestrated by frameworks like Apache Spark [2], Kubernetes [26], SLURM [43], or specialized HPC runtimes [20]. These frameworks typically treat each server as a standalone independent entity because servers are exclusively interconnected via network technologies, like Ethernet.

A new generation of high-speed peripheral interconnects, Compute Express Link (CXL) [13] among others, promises to revolutionize this landscape by enabling memory to be accessible at the IO bus level, and shared among servers. In other words, memory can now be connected on the IO bus (i.e., on PCIe) and accessed by multiple hosts, rather than solely attached to a CPU memory bus and accessed by a single host. Historically, HPC systems have used custom coherent interconnects [1, 41, 42], but the arrival of CXL expands the concept of shared memory beyond specialized clusters, bridging HPC technologies with commodity data center infrastructure. Although early versions of CXL (1.0, 1.1, and 2.0) are already available – supporting features like memory expansion and basic switching [44], newer CXL 3.0 [12] with hardware cache coherence is still emerging, and prototypes of multi-host cache coherent shared memory pools connected via CXL switches are not available yet because under development.

Today, exploiting these shared memory pools requires a significant amount of changes to applications (e.g., using dedicated `set()/get()` interfaces [18]) or removes the OS from the control path (e.g., by leveraging Linux Direct Access – DAX). Indeed, the shared memory window abstraction, like Famfs [15], is a step towards solving such problems. It can be used to let multiple processes on different machines to access the same memory area with the memory consistency provided by each specific CPU and the CXL equipment used – i.e., CXL 3.0 strong consistency. Nevertheless, CXL 3.0's large address spaces with hardware-managed coherence may make things even more complicated because hardware-managed coherence is not a panacea [14, 21]. In fact, even assuming all CPUs have the same memory consistency, naively mapping a CXL memory area to multiple machines may result in such machines seeing very different access latencies, due to CXL network topology – affecting overall application performance. At the same time, the utilizable memory could reduce (a) increasing the number of machines sharing such memory – due to cache coherency tracking, and (b) increasing the area size [14].

An alternative to hardware-managed coherence is software-managed coherence (required by CXL 2.0). While software-managed coherence in data centers is not much used today because of its overheads [9], its implementation over CXL (vs over network technologies) hasn't been fully explored yet, and initial evidence [48] shows potential benefits. In fact, page copy is one of the key mechanisms in software-managed coherence, and by just using optimized string copy instructions (like REP MOVSB on x86 [11]), the execution time of page copy is not much different than the time to access a cacheline. Thus, we believe that there should be a tradeoff between hardware- and software-managed coherence, which we analyze in this paper. We identify that the tradeoff varies not only based on the distance between a CPU and a CXL memory module, but also by an application access pattern, which changes during program execution. Hence, a CXL memory area may benefit from hardware-managed coherence for a part of the program but from software-managed coherence for another part.

We propose to change the way an application's address space works today, where each memory area is on hardware cache coherence memory – when pages are not swapped out. With our proposal, each memory area or a sub-area is either on hardware- or software-managed coherence, and that changes over time based on the application memory access pattern. Prior work by Kelm et al. [25] demonstrated that such dynamic switching is possible.

In this paper, we introduce our design: *Adaptive Coherence Management*, which implements our vision of a dynamic application address space, where parts of the address space continuously change their coherency management strategy based on varying applications' memory access profiles. This enables multi-process applications to run seamlessly among multiple machines while sharing memory over CXL consistently and performantly.

2 Background

2.1 Memory on the Peripheral Interconnect

CXL is a high-speed asymmetric cache coherence interconnect standard that leverages the PCIe physical layer to enable attaching memory on the peripheral bus with access latencies similar to CPU-attached memory, with potentially hardware-managed coherency. Early versions, CXL 1.0 and 1.1, enable direct point-to-point links between the CPU and a (memory) device. With CXL 2.0/2.1, coherence is largely managed by the host processor, making shared memory among multi-hosts possible, but likely costly [14].

Hardware-managed Coherence (Direct Access). CXL 3.0 introduces Back Invalidation (BI) – a hardware-level mechanism for managing cache coherence. A memory device can send Back-Invalidate Snoop (BISnp) messages to any host that holds a copy of a shared cacheline, prompting them to invalidate or update it – eliminating the cost of device–host round trips prominent in CXL 2.0's host-driven coherence. Each CXL 3.0 memory device can maintain a directory or snoop filter that tracks the cacheline ownership and status – like the MESI cache coherency protocol. With hardware cache coherency software running on different machines *directly access* CXL memory with load/store semantics consistently. However, applying hardware-managed coherence at very large scales (e.g., terabytes of shared memory pool) remains challenging: directory tables or snoop filters may become huge, and the overhead of tracking frequent updates grows accordingly. Thus, some designs opt for coarser-grained coherence (e.g., page-level) or rely on workloads' varying memory access patterns [14, 21] – such as write-intensive, that may benefit the most from hardware-managed coherence.

When software accesses non-local memory kept consistent via hardware, like NUMA or CXL 3.0, we identify that as **Direct Access**.

2.2 Distributed Memory

Today, large-scale data analytics and HPC applications rely on distributed memory. Traditional runtime libraries such as MPI [34] treat each node as individual, each having distinct memory and exchanging messages to keep data synchronized among nodes. Similarly, OpenSHMEM [35] adopts a Partitioned Global Address Space (PGAS) model with APIs for remote memory access, atomic operations, collective communication, and synchronization. In modern cloud platforms, distributed frameworks like Apache Spark [2], Kubernetes [26], and SLURM [43] coordinate workloads across the machines in a cluster. Again, this is using software-level communication and synchronization to handle the data sharing. While these approaches are well-established, in the majority of cases the overhead of communication become a performance limit – especially as data sets and node counts grow.

Software-managed Coherence (Page Replication). Numerous research efforts have explored the way to unify physically separated memories under a consistent shared memory abstraction. Efforts include user-space libraries, like Grappa [31] or Argo [24], as well as supervisory software solutions. Those comprise multiple kernels OS designs [4, 6, 8, 23, 28, 32], which explored how to use software-managed coherence in the form of Distributed Shared Memory (DSM) [3, 5, 7, 39], or hardware-level address remapping over interconnects like PCIe [8]. Likewise, Hypervisor-based solutions (e.g., ScaleMP [40], TidalScale HyperKernel [45], FragVisor [9, 33]) aggregate multiple physical machines into a single large-memory "mega-node." Although all such solutions create the illusion of a unified consistent memory, software-managed coherence – which requires the *replication* of data and keeping the *replicated* data copies consistent among multiple machines at the granularity of pages, can become a scalability bottleneck. Particularly, when writes trigger invalidation across many machines [31].

Because software-managed coherence keeps replicas of each page among different machines, we also call it **Page Replication**.

3 Motivations

In a foreseeable future, data centers will continue to scale up and machines will become more tightly interconnected – thanks to new technologies like CXL, providing shared memory across the cluster(s). Thus, <u>how to handle terabytes of memory shared amongst different machines?</u> Designers face a critical decision: maintain coherence through software-managed coherence (e.g., DSM), or hardware-managed coherence (e.g., CXL 3.0). Figure 1 depicts the two options. Each option brings advantages and drawbacks, and the multi-tiered memory structure in future data centers further complicates the choice.

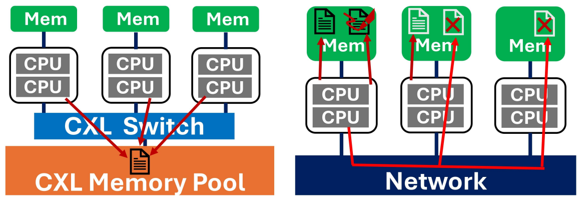


Figure 1: Multiple machines interconnected via CXL sharing accesses to a single memory pool at byte granularity and memory semantic – potentially HW cache coherent (left figure). Multiple machines network interconnected using software distributed shared memory to concurrently access the same data (right figure).

3.1 Pros and Cons of Existing Approaches

Software-managed Coherence. This may be implemented via network (e.g., Ethernet, RDMA) or peripheral/memory bus (i.e., load/store, DMA). Pages are moved on demand among machines and/or memory pools, each machine accesses only local copies.

- **Pros:** Local data copies reside in local memory, offering low-latency direct accesses.
- **Cons:** Under heavy sharing, writes trigger invalidations across all replicas on different servers, causing high overheads [24]. As data volumes grow, repeated invalidations may become prohibitively expensive [3, 9].

Hardware-managed coherence. This is exclusively implemented in hardware; we assume sequential memory consistency or an approximation of it like in x86 [36].

- **Pros:** Hosts share data at (remote) memory access latency for both read and write operations. Writes may have the additional latency of Back-invalidate messages, like in CXL 3.0. This is very probably faster than software-managed coherence – at least up to a certain number of CXL switches.
- **Cons:** At terabyte scales, maintaining per-cache-line directory or snoop filters becomes impractical – both in storage requirements and lookup times [14, 16, 21, 25]. Moreover, it is unclear how hardware cache coherence is going to scale increasing the number of readers and writers [6].

3.2 Challanges

Challenges with Future Cluster Topologies. Although direct-attached CXL devices might have latencies in the realm of ~200ns-400ns – comparable to remote NUMA (~150–200ns) [29], future cluster topologies will feature multiple tiers of memory over CXL switches, with latencies from ~600ns up to the microseconds scale [29]. This increases the per-memory access cost, but also increases the cost of hardware cache coherence.

Takeaway. Such variability of latencies underscores why no single coherence mechanism – be it software-managed or hardware-managed coherence, can handle all workloads effectively. Further, even hybrid solutions that combine hardware and software techniques may struggle if they are static, unable to respond dynamically to changing access patterns or memory access latencies. Therefore, future data centers require a dynamic memory subsystem that can adaptively switch among different coherence mechanisms on a per-region (or per-partition) basis – e.g., using hardware-managed coherence when writes are dominant and software-managed coherence when local caching offers the most benefit.

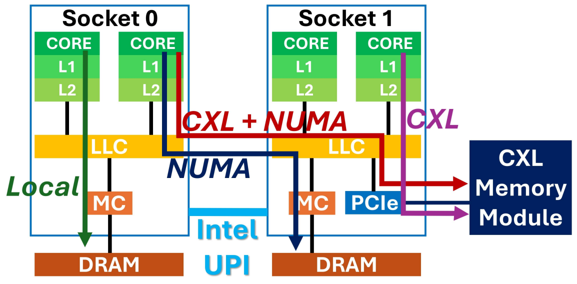


Figure 2: Experimental setup and memory access schema.

Challenges with Using Shared Memory. Current solutions such as MemVerge's Project Gismo – a user-space solution [18], require (1) application rewriting to adopt the API calls, such as the `Gismo_release()` and `Gismo_get()`, as well as managing the data mappings; and (2) kicking out the OS from the control path, because memory areas are mapped as DAX. While the former impedes applications' portability to CXL shared memory, and may originate vendor lock-in situations, the latter limits what the OS can actually do to support applications. In fact, newer approaches such as Famfs [15], which address a similar and complementary problem, are implemented in the OS, repositioning the OS on the control path.

Takeaway. An application transparent approach doesn't require application rewriting. Thus, avoiding (re-)development costs, as well as vendor lock-in. However, tackling transparency in user space is cumbersome; therefore, OS-level approaches should be considered. Tackling transparency at the OS level has the additional advantage that it enables runtime application profiling, and consequent data mapping and migration, without program rewriting.

Specifically, hardware-based memory access monitoring (e.g., performance counters or PEBs [38]) could be leveraged to dynamically profile application behavior at runtime. Based on these observations, the OS might seamlessly partition an application's address space, matching hot or frequently written data with the most suitable coherence management mechanism. Moreover, because the OS can directly access all memory tiers, it can factor in varying latency characteristics to optimize data placement – a decision that would otherwise require explicit user-space coordination.

4 Overheads Analysis

We conducted different experiments to explore the tradeoffs between hardware-managed coherence (direct access) and software-managed coherence (page replication). Experiments were run on a single Supermicro PC-BX25270 with dual Intel Xeon Gold 5418Y @ 2.00GHz and 768GB of memory (over 2 NUMA/socket), equipped with a CXL 1.1 device. In our setup, the CXL memory module(CMM), Samsung CXL MXP E3.s 128GB [10] is connected to NUMA/socket 1; this is schematized in Figure 2. The machine runs Linux kernel

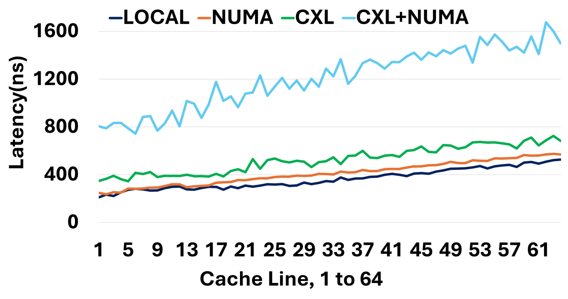


Figure 3: 1- 64 cacheline direct access overhead (ns), compare between Local, NUMA, CXL and CXL+NUMA.

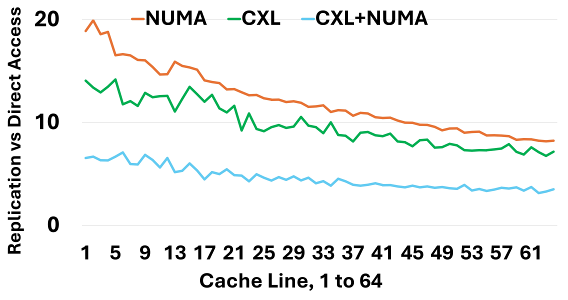


Figure 4: Number of remote cacheline(1-64) fetches equivalent to the cost of page replication(both OS stack and the data movement overheads) over NUMA, CXL and CXL+NUMA.

version 6.13, which we modified for the experiments. We explored 4 different scenarios:

- **Local** accesses directly-attached memory on the same NUMA node as the running thread;
- **NUMA** accesses directly-attached memory on a remote NUMA node (1 hop);
- **CXL** accesses CXL-attached memory on the same NUMA node as the running thread; and
- **CXL+NUMA** accesses CXL-attached memory on a remote NUMA node (i.e., 2 hops: remote NUMA, and CXL).

Because we don't have any CXL switch, we believe CXL+NUMA is a reasonable proxy for the scenario where a CXL memory module is interconnected via a CXL switch. While CXL+NUMA may be representative for read or write latency and bandwidth costs for a single host, we found the setup may not sensibly reproduce the behavior of multiple hosts read and write or all write at the same time – i.e., scenarios in which the cache coherency protocols kick in. This is because the CXL hardware cache coherence is handled at the memory extender, while in our case at the last level cache. As a future work, we are planning to further analyze such scenarios either on real hardware or on a simulator [17, 48]. Because of that, we didn't consider the cost of a software consistency protocol that will also be addressed in future work.

Direct Access. CXL 3.0 tracks memory accesses at cacheline granularity. Hence, we conducted experiments to measure the time to fetch data (direct access) from 1 to 64 cachelines, for different memory tiers – where 64 cachelines make up a page (4kB). Before each experiment, we polluted the caches and flushed the L3 to ensure the memory was actually accessed. All measurements are taken in user-space. Results are shown in Figure 3, the latency varies across different memory tiers.

In our setup, accessing data over CXL+NUMA is approximately 4× more expensive than accessing local DRAM, aligning with Liu et al. [29], which identified cache hierarchy traversals and prefetcher behavior as the primary overhead contributors. While CXL 3.0 devices are not yet on the market, we anticipate that at larger scales, hardware-managed coherence via BI could introduce potential messaging overhead, as observed in similar contexts from previous work [25]. Furthermore, although the way in which CXL 3.0 integrates into contemporary microarchitectures remains uncertain (e.g., the stall because of latency mismatch in the CXL+NUMA case [29]), the multi-tier latency variations may become main factors influencing both system design and flexibility in next-generation cloud data centers.

From	NUMA	CXL	CXL+NUMA
`migrate_pages()`	4826ns	4966ns	5272ns
`memcpy()`	887ns	942ns	1158ns

Table 1: Time to copy one page (4kB) from non-local (either NUMA, CXL, or CXL+NUMA) to local memory.

Page Replication. To assess the cost of Page Replication, we modified the Linux kernel so that when an application accesses a non-local page, that page will be migrated over to local DRAM. We modified Linux's page fault handler (`handle_page_fault()`), and precisely its user-space handler, to force execute `migrate_pages()` to copy, unmap, and remap a page from NUMA, CXL, or CXL+NUMA to local memory. Before that, all non-local pages are R/W protected, and we pollute the CPU caches to ensure `migrate_pages()` will actually do (non-local) memory accesses when copying.

We measure the time for page migration (`migrate_pages()`) in user-space to include also the cost of transition from/to user-space – faulting, and the time for the actual data copy (`copy_mc_highpage()`, which calls a very optimized `memcpy()`) in kernel-space. We use the `rdtsc` assembly instruction to measure these two costs, and we report the results in Table 1 for different tiers. Our results show that OS management overheads dominate the cost – about 91% is spent on the OS unmapping and remapping the page, while the actual memory copy is in the range 815-1328 ns – varying based on different memory tiers. The average page replication cost on different tiers we have tested is about 8-13 µs. It is worth noting that the overall page replication time is almost independent from the source or destination because it is dominated by OS management routines.

4.1 Direct Access vs Page Replication

We expect that as remote latency grows – e.g., connecting multiple CXL switches, the additional penalty for *direct access* remote memory increases, while the *page replication* overhead remains relatively constant. Hence, the benefit of replicating pages grows as connection latency increases. Indeed, when multiple hosts access the same data in R/W or in write, things may look different as software-managed coherence is notorious for its scalability challenges [3, 9] (for instance, page invalidation must be broadcast to multiple nodes upon a write). Hardware-managed coherence presents similar scaling problems [6], but it may be faster than using the network. Nevertheless, this highlights the need for dynamically selecting the appropriate coherence mechanism at runtime to maximize the potential benefit of cross-machine shared memory programming. Overall, page replication – though software-heavy, can still be beneficial when remote latency is large and the data is read frequently.

Figure 4 shows the cost comparison between direct access and page replication (`migrate_pages()`) – which always copies an entire page, varying the amount of cachelines accessed. For a NUMA setup, if an application accesses only one cacheline on a remote node, the overhead to replicate the entire page equals roughly 20 NUMA accesses. If the application accesses an entire page (64 cachelines), that ratio drops to about 9 accesses. For access over CXL or CXL+NUMA, because the access latency is getting higher, the direct access to page replication ratio drops to about 7-14 for accessing a single cacheline, or 4-7 for full-page access.

In other words, higher remote latency makes page replication more attractive. Moreover, highly polluted caches will enforce the CPU to fetch data again and again from non-local memory for direct access, while page replication will access local memory, making the latter a more favorable solution. Finally, as the actual page copy is very short (see Table 1), we believe `migrate_pages()` can be redesigned to be faster.

4.2 Application Memory Profiles

We profiled memory accesses of various NAS Parallel Benchmarks (NPB) [30] with four threads using Intel Pin [46]. We chose NPB because benchmarks are compute and memory bound. Figure 5 plots the fraction of read accesses (x-axis) against the fraction of total shared pages (y-axis). We excluded all pages that are accessed but not shared among threads. This is because such pages are not needed for coherence management; therefore, they could be kept in local memory vs being kept in (CXL) shared memory.

Several workloads, BT, CG, IS, LU, MG, and SP exhibit at least 20% of their shared pages that have over 90% of accesses that are reads. On average, these applications see at most one write for every ten reads. For such read-intensive workloads, this suggests that replicating pages locally could outperform remote direct access, especially in high-latency settings (e.g., CXL+NUMA). However, EP and FT show write ratios of 40–70% in most of their shared pages, which indicate the frequent write during the workload processing – making page replication less appealing since every write introduces overheads to maintain consistency. Therefore, in these scenarios, hardware-managed coherence will likely outperform software-managed coherence. Collectively, these observations confirm that no single approach – direct access or page replication – dominates across all usage scenarios. Instead, the optimal strategy depends at least on hardware memory latency (e.g., local, vs NUMA, vs CXL tiers), and application access patterns (read- vs. write-intensive).

5 Rethinking Applications' Address Space

Applications' (virtual) address space are backed up by actual memory – if not swapped to persistent storage, disk, remote memory; at the granularity of pages. At the same time, an address space is divided into (virtual) memory areas. Each area has different protection bits and hosts file-backed data or is an "anonymous" mapping (like the `.heap`). On Symmetric Multiprocessing (SMP) machines – i.e., multicores and multiprocessors, it is assumed that the entire address space follows the same consistency model, which is provided by hardware cache coherency[1]. Therefore, the coherency mechanism is fixed – i.e., hardware-managed, for the entire address space.

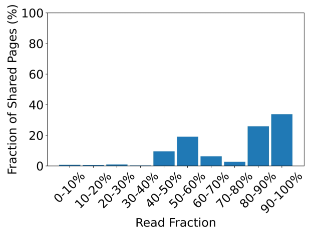
We believe that with CXL, also software-managed coherency will play an important role, and based on the application memory access profile, each virtual memory area – or a set of its constituent pages, may need to dynamically switch between hardware- and software-managed coherence at runtime, potentially transparently from the application. Making applications' address spaces more dynamic. With the goal of enabling applications to efficiently share memory among multiple machines, which is crucial for many real-world workloads, including Python-based AI workloads and large-scale data analytics, we propose the *Adaptive Coherence Management*, which realizes this vision.

5.1 Design Principles

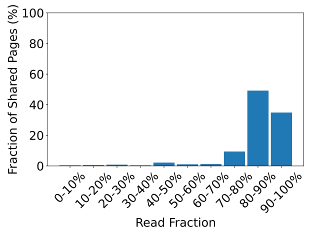
Our proposal is based on the following design principles that stem from the need for application transparency for portability and low overhead for high performance:

- **Single Memory Consistency Model**
  We aim at providing the illusion of a single consistency model independently of the coherence mechanism that is managing an area of memory. This abstracts away lower-level protocols from the application, allowing dynamic adaptation to changing conditions.
- **Per Address Space-area Handling**
  The application's (virtual) address space is logically divided into areas – like VMAs in Linux, each (1) managed by the most suitable coherence mechanism, and (2) backed by a potentially different memory tier at any time. Areas can also be split or merged based on their backing memory and coherence mechanism other than what already done in existent OSes. The application is completely oblivious of such address space changes, but an application programming interface is provided to put the application back into control.
- **Lightweight Runtime Profiling and Adaptation**
  The proposed solution continuously monitors – with low overhead, application access patterns and system metrics (e.g., latency, bandwidth, usage frequency). Based on those, it dynamically adjusts the coherence management mechanism for each area to optimize performance, resource usage, or energy efficiency.

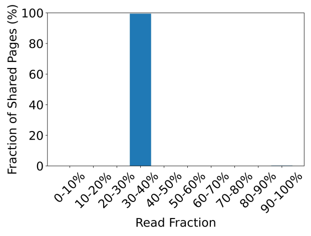
[1]Indeed, programs that bypass the OS kernel may include device memory in their address space that is mapped as uncached, for which there is no consistency guarantee.



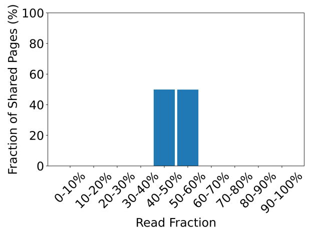
(a) BT



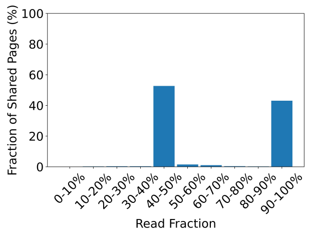
(b) CG



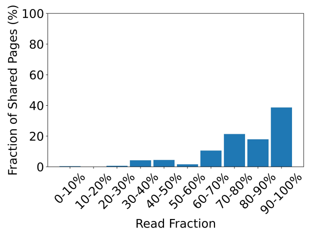
(c) EP



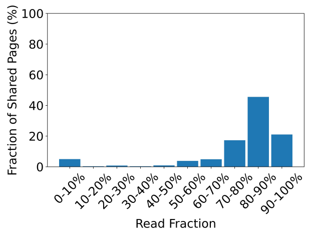
(d) FT



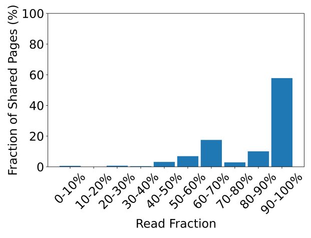
(e) IS



(f) LU



(g) MG



(h) SP

Figure 5: NPB profiling, X axis indicate the read fraction, and Y axis indicate the fraction of shared pages.

5.2 Design

The envisioned *Adaptive Managed Coherence* combines hardware- and software-managed coherence. In this design, each shared memory page (or huge page) can be *switched* between hardware- and software-managed coherence, depending on the application memory access pattern, captured with real-time profiling. Users interact with such shared memory via familiar UNIX-like interfaces (e.g., POSIX shared memory, Famfs), and can enforce static policies when desired, or opt into dynamic policies for automatic memory area management.

When a page is managed via hardware coherence, it is configured as a traditional page. If a page or memory range is designated as software-managed, the system adopts a page replication protocol similar to Popcorn Linux DSM handling [4] or an ownership protocol (OWN) like Intel MYO [22] or Linux HMM [19]. A flag indicates whether a page should be handled by software-managed coherency or not; in Linux, this flag can be kept in `struct page`. Upon a page fault, the system checks whether the page is valid or needs to be synchronized (e.g., fetched, or invalidated); such synchronization may require communication with other machines, for example using message-passing or inter-machine interrupt.

The proposed *Adaptive Managed Coherence* gathers *fine-grained access statistics* to determine whether page replication or direct access yields better performance. To make statistic collections lightweight we envision using counters over CXL [47], offloading significant profiling tasks from the CPU. Note that recent range-based memory access profilers, like DAMON [37]), are less effective in a shared-data scenario because they don't differentiate reads from writes, a critical factor in choosing coherence protocols. Instead, mechanisms like PEBs [38] can be used to detect read/write operations. An ML-based model could be used to learn application behavior over time [27] and decide when to use page replication or direct access to optimize performance. Finally, in this paper we consider traditional page migration between NUMA zones or different memory tiers as a mechanism that works together with direct access only. This is because traditional page migration cannot handle multiple copies.

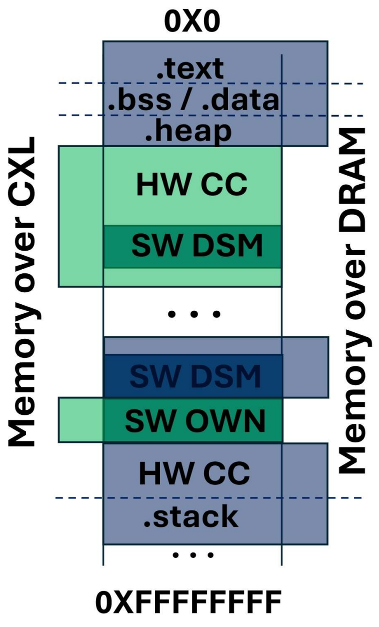


Figure 6: A snapshot of the proposed process address space.

5.3 Run-time Behavior

Hardware- and software-managed coherence are not mutually exclusive; for an address space, those are combined in both time and space.

- **Temporal Combining:** When a page experiences frequent writes during a particular phase, direct access may have the best performance. Later, when the workload transitions to read-dominant behavior, ownership-based or replicated pages can provide better performance instead.
- **Spatial Combining:** Different address space areas (linked to different memory devices, possibly on different tiers) can be either hardware- or software-managed coherence. Which one to use depends on the memory latency and bandwidth, but also application's memory access pattern – e.g., exclusive or shared memory, read-only or R/W memory.

Such address space areas may be mapped to Linux's Virtual Memory Areas (VMA). Illustrated in Figure 6, different coherence mechanisms apply to different address space areas and they change over time. Indeed, VMAs granularity rather than individual pages may reduce overheads, since monitoring and profiling don't have to maintain information per page but for a group of pages (see DAMON range-based profiling [37]).

However, a VMA may be too large in certain cases – that depends on application behavior. Thus, we suggest that such VMA should be split or handled by a sub-area.

6 Conclusion

In this paper, we have shown that software-managed coherency, provides benefits even in an era of CXL 3.0, which introduces hardware cache coherence for multiple machines accessing the same CXL shared memory pools.

By analyzing the strengths and weaknesses of hardware- and software-managed coherence with CXL, we find that neither of the two is generally optimal. Therefore, we propose rethinking applications' address space – which today is mainly backed by hardware coherent memory, introducing *Adaptive Coherence Management* that dynamically selects the most suitable hardware- or software-managed coherence mechanism based on runtime profiling of an application – e.g., hot and cold pages, and the latency of the accessed CXL memory.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work is supported in part by the UKRI EPSRC under grant EP/V028154/1.

References

[1] Altix. 2025. https://www.euroben.nl/reports/web13/altix.php.
[2] Apache. 2025. Apache Spark. https://spark.apache.org/.
[3] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 645–659. doi:10.1145/3037697.3037738
[4] Antonio Barbalace, Binoy Ravindran, and David Katz. 2014. Popcorn: a replicated-kernel OS based on Linux. In *Proceedings of the Linux Symposium, Ottawa, Canada.*
[5] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.
[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. doi:10.1145/1629575.1629579
[7] Sharath K. Bhat, Ajithchandra Saya, Hemedra K. Rawat, Antonio Barbalace, and Binoy Ravindran. 2015. Harnessing Energy Efficiency of heterogeneous-ISA Platforms. In *Proceedings of the Workshop on Power-Aware Computing and Systems* (Monterey, California) *(HotPower '15)*. ACM, New York, NY, USA, 6–10. doi:10.1145/2818613.2818747
[8] Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2020. Flick: Fast and Lightweight ISA-Crossing Call for Heterogeneous-ISA Environments. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 187–198. doi:10.1109/ISCA45697.2020.00026
[9] Ho-Ren Chuang, Karim Manaouil, Tong Xing, Antonio Barbalace, Pierre Olivier, Balvansh Heerekar, and Binoy Ravindran. 2023. Aggregate VM: Why Reduce or Evict VM's Resources When You Can Borrow Them From Other Nodes?. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 469–487. doi:10.1145/3552326.3587452
[10] Samsung CXL Memory Module Box (CMM-B). 2025. https://semiconductor.samsung.com/next-gen-memory/cmm-d/.
[11] comp.arch conversation. 2017. REP MOVSB performance (was: Hardware simulation in Forth). https://groups.google.com/g/comp.arch/c/ULvFgEM_ZSY?pli=1.
[12] Compute Express Link Consortium, Inc. 2022. *Compute Express Link (CXL) Specification* (3.0 ed.). https://www.computeexpresslink.org/download-the-specification Available: Compute Express Link Consortium, https://www.computeexpresslink.org/download-the-specification.
[13] CXL Consortium. 2022. CXL Specification. https://www.computeexpresslink.org/download-the-specification.
[14] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* 56, 11, Article 290 (July 2024), 37 pages. doi:10.1145/3669900
[15] FAMfs. 2025. https://lpc.events/event/18/contributions/1827/.
[16] HPC Software Scaling for ML using CXL 3.0 GFAM. 2025. https://ipdrm16.github.io/Estep_IPDRM.pdf.
[17] Gem5. 2025. https://www.gem5.org/.
[18] Memverge Gismo. 2025. https://memverge.com/wp-content/uploads/MemVerge-Gismo-FMS2023.pdf.
[19] Jérôme Glisse. 2018. HETEROGENEOUS MEMORY MANAGEMENT. In *Linux Plumbers Conference 2018*. https://lpc.events/event/2/contributions/70/attachments/14/6/hmm-lpc18.pdf
[20] NVIDIA® HPC-X®. 2025. https://docs.nvidia.com/networking/display/hpcxv2162.
[21] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory Sharing with CXL: Hardware and Software Design Approaches. *arXiv preprint arXiv:2404.03245* (2024).
[22] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2nd Edition* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[23] David Katz, Antonio Barbalace, Saif Ansary, Akshay Ravichandran, and Binoy Ravindran. 2015. Thread migration in a replicated-kernel os. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 278–287.
[24] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/2749246.2749250
[25] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. 2010. Cohesion: a hybrid memory model for accelerators. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 429–440. doi:10.1145/1815961.1816019
[26] Kubernetes. 2025. https://kubernetes.io/.
[27] 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 *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835
[28] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. 2015. K2: A Mobile Operating System for Heterogeneous Coherence Domains. *ACM Trans. Comput. Syst.* 33, 2, Article 4 (June 2015), 27 pages. doi:10.1145/2699676

[29] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S Berger, and Huaicheng Li. 2024. Dissecting cxl memory performance at scale: Analysis, modeling, and optimization. *arXiv preprint arXiv:2409.14317* (2024).

[30] NASA Advanced Supercomputing Division. 2025. NAS Parallel Benchmarks. https://tinyurl.com/y47k95cc.

[31] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 291–305. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[32] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 221–234.

[33] Pierre Olivier, Binoy Ravindran, and Antonio Barbalace. 2019. The Multihype: Virtualizing Heterogeneous-ISA Architectures. In *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*.

[34] OpenMPI. 2025. https://www.open-mpi.org/.

[35] OpenShmem. 2025. http://www.openshmem.org/site/.

[36] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. doi:10.1007/978-3-642-03359-9_27

[37] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3366626.3368125

[38] PEBS. 2025. https://www.intel.com/content/www/us/en/developer/articles/technical/timed-process-event-based-sampling-tpebs.html.

[39] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia. 2013. A Page Coherency Protocol for Popcorn Replicated-kernel Operating System. (2013).

[40] ScaleMP. 2025. https://wiki.metacentrum.cz/wiki/ScaleMP_(en).

[41] SGI.UV.1K.book. 2025. https://irix7.com/techpubs/007-5663-001.pdf.

[42] SGI.UV.3K.book. 2025. https://irix7.com/techpubs/007-6402-001.pdf.

[43] SLURM. 2025. https://slurm.schedmd.com/documentation.html.

[44] Xconn CXL switch. 2025. https://www.xconn-tech.com/product.

[45] Tidescale. 2025. https://cloud.ibm.com/catalog/content/node-red-operator-certified::2-7650c9f1-28ba-404e-a2bb-f898466ee6e1-global.

[46] Pin A Dynamic Binary Instrumentation Tool. 2025. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[47] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 79–94. doi:10.1145/3694715.3695968

[48] Tong Xing, Cong Xiong, Tianrui Wei, April Sanchez, Binoy Ravindran, Jonathan Balkind, and Antonio Barbalace. 2025. Stramash: A Fused-kernel Operating System For Cache-Coherent, Heterogeneous-ISA Platforms. (2025). doi:10.1145/3676641.3716275