Synchronization for CXL Based Memory

Joshua D. Suetterlein, Joseph B. Manzano, Andres Marquez
Pacific Northwest National Laboratory
Richland, Washington, USA
[joshua.suetterlein,joseph.manzano,andres.marquez]@pnnl.gov

ABSTRACT

Compute Express Link (CXL) is an important emerging standard for disaggregated memory. While this standard provisions coherency across numerous hosts and devices, implementing hardware support for type three devices is challenging. In this work, we look at the overhead of software synchronization and using software-based coherency. Moreover, we discuss the limits of software-based coherency in fully expressing modern synchronization techniques for a CXL-based disaggregate memory system. We demonstrate our approach using a CXL hardware prototype and running a version of the famous Peterson Lock (enhanced to run with more than two threads). We analyze its performance and share how more advanced synchronization techniques might interact with software-based coherence CXL hardware and program execution models.

1 INTRODUCTION

With the advent of Fabric Attached Memory (FAM), the prospect of a more cost-efficient memory provisioning that addresses capacity and system bandwidth requirements is an essential motivator for hyperscalers. While this aspect will help drive the maturation of this technology on its own, interesting questions arise for computing communities that could leverage FAM for their own specific purposes.

The hallmark of High-Performance Computing (HPC), and more concretely Machine Learning (ML), is a parallel/distributed execution model (EM) that relies on lightweight task creation/termination and synchronization (including exclusion and collectives), as well as fast data distribution and communication. It stands to reason that FAM, within some degree of locality, should hold the promise of facilitating efficient implementations of parts of an HPC EM; even more so, if the memory is enhanced with smarts provided by Near Memory Computing (NMC) acceleration.

FAM, in its incarnation as Compute Express Link (CXL) [1], has been gaining momentum, subsuming the likes of CCIX [2] and Gen-Z [3], and represents an ideal platform to test that hypothesis. In its 3.x version, CXL now supports multi-host, multi-device switched memory pooling and sharing with cache-coherency support. It multiplexes on Peripheral Component Interconnect Express (PCIe), providing a mechanism to extend passive and active computing devices over serial links. Compared to previous attempts to implement HPC EM over shared memory, one salient feature of CXL is the end-to-end implementation of the CXL standard into the processing devices, **without the need for software shims or drivers**[1] representing a shared memory system architecture: hereby providing actual Load/Store semantics. Another distinguishing characteristic of previous (virtual) shared memory implementations is the disaggregated location of the memory pool. Combined, they offer an interesting dichotomy to explore. We must find a suitable design point that juxtaposes the performance requirements of existing cache-memory hierarchies within a host with the need for parallel/distributed EM across hosts on disaggregated memory.

This paper is the first in the authors' series to answer the hypothesis by analyzing the constituents that entail the HPC EM. In this one, we tackle the thorny question of "synchronization," as its different components and requirements are prominent aspects that will enable the effective collaboration between the current memory hierarchies and the CXL-based HPC EM. In particular, we explore the feasibility of fine-grain synchronization without hardware support. The contributions of the paper are:

(1) an implementation of a Peterson/Tournament Lock for CXL using software coherence and no atomic operations;
(2) a study of the performance of such lock on a CXL hardware prototype running in one and two nodes configurations; and
(3) a discussion comparing and contrasting our strategy with modern synchronization primitives leveraging hardware support with an eye toward feasibility.

2 BACKGROUND

The following provides the salient features of the disaggregated memory system we consider in the remainder of our work. In particular, we motivate our choice of pooled CXL-based disaggregated memory using a PGAS memory model to evaluate software-based synchronization and how other synchronization efforts might be ported to the current CXL testbed.

2.1 Architecture

Disaggregated memory can broadly be categorized into two variants: split and pooled. The split architecture maintains the tight coupling between compute and memory but allows nodes to request additional memory from others in a peer-to-peer fashion. The memory division within can be flexible, and an additional daemon is required to service memory requests. Conversely, the pooled approach employs FAM to be shared by multiple nodes. Memory can be attached directly to a node or through a network of dedicated switches. The pooled approach provides a clear distinction between compute nodes and memory, permitting a more flexible approach to upgrading an existing system based on changing needs. The CXL standard makes the pooled approach attractive, with vendor support promising coherent memory and CXL-based switches for advanced memory networks.

Figure 1 illustrates a pooled architecture using CXL and demonstrates the two levels of fine-grain data sharing that CXL aims to achieve. It is important to note that the challenge of implementing fine-grain synchronization extends beyond scaling out. The first

[1]Current solutions tend to favor software coherence instead of a hardware-based one due to its complexity and overhead

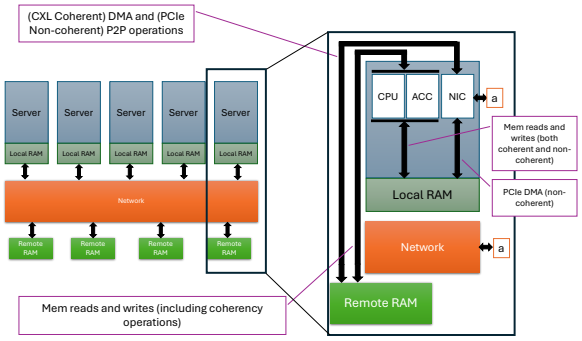

Figure 1: A Pooled CXL Environment Overview

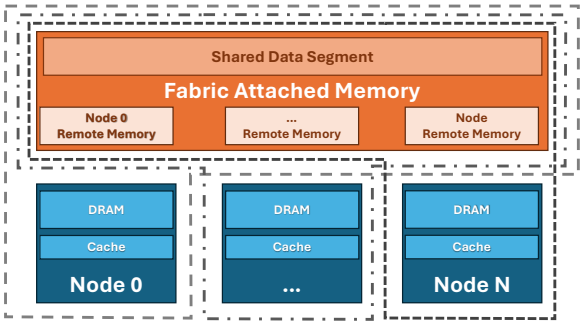

Figure 2: Hierarchical address space of a PGAS disaggregated memory system. Each node's process includes a vertical and horizontal view of memory.

level of sharing involves coherent memory access within a node, which allows CPUs, various compute accelerators (e.g., GPUs), and FAM to share data. The second level enables coherent data sharing across nodes through a dedicated memory network.

2.2 Address Space

The authors of [4] provides a thorough overview of the various efforts before CXL, highlighting three different address space types appropriate for data sharing: key-value store, unified address space, and PGAS. The key-value store model is restrictive and forgoes a traditional memory layout (e.g., Clover [5]). In return, strong consistency guarantees can be made at the additional programming development cost. The unified address space makes all memory, including the nodes, accelerator devices, and FAM, visible to every node in the system (e.g., vNUMA [6]), Argo [7], and vDSM [8]). The address space can be flat or hierarchical; however, address translation is challenging regardless, even with hardware support. PGAS is the most attractive approach, as it more easily extends to the heterogeneous node architecture (i.e., accelerators and FAM). Examples for this address space include DVM [9], and soNUMA [10]. In the PGAS model, each process has its own local and global address space. In the case of CXL-based disaggregated memory, FAM makes up the global shared address space.

Using a PGAS model's global address space requires additional considerations from a program execution model (PEX) point of view. PGAS models have been well studied within disaggregated memory and out [11, 12]; however, CXL-based systems present new opportunities to revisit their design thanks to low(er) latency memory access, explicit coherence guarantees, and potential hardware support.

Figure 2 presents our view of the hierarchical CXL-based address space. The local address space consists of memory from a single node (and its compute accelerators) and any remote memory requested on-demand during execution. This vertical address space includes a node's DRAM and cache. The shared (horizontal) address space consists of dedicated FAM. We envision a segment of this memory similar to a C/C++ data segment. The user/compiler can identify the required shared memory at compile time, and the runtime can initialize it before execution (similar to work presented in [13]). Nodes can directly access data in the shared data segment or use data structures such as queues, deques, and maps to share memory from their "local" block of FAM.

Synchronization within a programming model typically spans its memory consistency model and explicit atomic/synchronization operations. The granularity of execution depends on the overhead of synchronization, and an efficient parallel execution must amortize this overhead to observe the benefit of parallelism. From a software perspective, synchronization primitives are ideally implemented in hardware and found in all modern processors and some high-performance network devices. Synchronization is expensive to implement in hardware once the synchronization domain exceeds a threshold of active state maintained by a group of participants as a function of intra-group distance. This fact places the trade-off of software versus hardware-based coherency (and supporting in-memory atomic operations) on the critical path to achieving CXL's goal of fine-grained data sharing.

3 METHODOLOGY

In order to understand the impact of software-based synchronization that could ease hardware implementation requirements, we explore locks in FAM. From a software perspective, locks are a simple way to provide concurrent access to shared data. They are the building blocks for more advanced primitives such as semaphores, futures, and concurrent data structures. While modern locks are typically implemented using hardware-based intrinsics such as test-and-set, compare-and-swap, and fetch-and-add (see Section 19), we implement a tournament lock based on the classic Peterson algorithm [14] as it can be implemented with minimal software-coherence support.

The pseudo-code for implementing a Peterson lock is presented in Algorithm 1. The algorithm supports two threads and only requires atomic reads and writes. We manually insert producer and consumer cache flushes to ensure the latest data is available. Moreover, we pad each variable to the cache line boundary to eliminate any misses due to false sharing (as done in other classical lock implementations discussed in Section 19).

The tournament lock is built using a binary tree of Peterson locks. This setup gives a complexity $O(log(N))$ where N is the number of threads used to perform a locking operation. To allocate, the number of participating threads must be known and be uniquely

Algorithm 1: Peterson Lock Pseudocode for two threads

```
   Input  : The id of the running process
1  begin
2      Record L:
3          flag[2] ← [0, 0];
4          turn ← 0;
5      Function acquireLock(L, id):
6          backoff ← 1;
7          other ← (1 − id);
8          L.flag[id] ← 1;
9          producerFlush();
10         turn ← other;
11         producerFlush();
12         consumerFlush();
13         while L.flag[other] ≠ 0 & turn == other do
14             sleep(backoff);
15             backoff ← backoff ∗ 2;
16             consumerFlush();
17     Function releaseLock(L, id):
18         L.flag[id] ← 0;
19         producerFlush();
```

identified. The tournament lock will allocate $N - 1$ Peterson Locks. The locks are allocated in the shared data segment and initialized before execution.

4 EXPERIMENTAL SETUP

We used a hardware testbed composed of two nodes with an in-hardware CXL 1.x to CXL 2.x Protocol Conversion interface with a memory-accelerated component. Figure 3 showcases the cluster configuration. A daemon running on the second node is used to allocate memory on FAM. Each node comprises of two sockets composed of 28 Intel Xeon Gold 5420 physical cores (each with two logical threads) running at 4.1 GHz. Each physical core has a private L1 (2.6 MiB), a private L2 (112 MiB), and a shared L3 per socket of 105 MiB. The local memory for each node is 1 TB. Each node connects to the FAM device via PCIe5 and to each other through Ethernet. The memory accelerator has 32 GiB of extra memory to share between the two nodes. Finally, due to a hardware limitation of our testbed, the threads are pinned to a single NUMA domain (i.e., socket). The threads in node 1 are pinned to the first socket, and the threads in node 2 are pinned to the second socket. This setup showcases how intranodal locality affects the contention and runtime in this setup.

Regarding the software toolchain, we use GCC v11.4, Open-MPI v4.1.1, and Python 3.9.18. Our implementations use C++14's standard multi-threading. Because of the daemon behavior, we pre-allocate the shared memory segment in a separate process and keep the region alive (i.e., persistent) for the rest of the test application to access. This setup emulates the more advanced software toolchain described in Section 2.2.

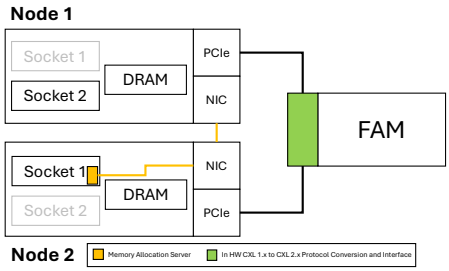

Figure 3: Block Diagram of our CXL Hardware Prototype

To test the software coherency of our CXL setup, we implemented a tournament of Peterson locks as described in Section 3. Our driver application allocates the lock's storage and the protected memory region in the FAM memory. Our critical region contains minimal operations (a single read-modify-write cycle plus a few flush operations) to ensure we are testing mutual exclusivity while maximizing the possible contention.

We performed our evaluation using each node separately and both nodes simultaneously. We used all the threads inside a single socket of each node. Each experiment performed the same number of lock and unlock operations while distributing them almost equally across all the threads. We ran each experiment 50 times to ensure statistical significance and record the time in nanoseconds. In order to contextualize our results, we normalized the performance to the cost of a read-modify-write operation to FAM.

5 RESULTS

In Figure 4, we plot the normalized average time of our locked critical region with the spread of the 50 collected cases across the number of threads used. Due to our testbed's single socket per node restriction, the single node 1 and node 2 cases end with 56 threads, while the case using both extends to 112 threads. As we increase the number of threads, the runtime increases thanks to contention (since the same amount of work is being performed). As shown, the rate of runtime increase is more pronounced in the low number of threads range, while it gets slower as we approach the more significant number of threads. This is because the tree structure helps to ameliorate some of the increased contention since only two threads can contend on an internal lock at a time. As we get to the highest number of threads, the overhead reaches around 1000x compared to a single thread's performance. This behavior indicates that the critical regions *should* be resized to be sufficiently large enough to amortize this overhead.

Figure 5 shows what happens when moving across nodes and their effect on the lock runtime. Figure 5 highlights the performance in the one and two-thread cases. In this figure, we show our baseline of the minimum time observed to perform a single read-modify cycle that reads, writes, and flushes to the FAM memory. This baseline is the normalization factor for all other experiments to showcase the lock overhead behavior. We present the maximum time observed for each single thread case as the overhead of the

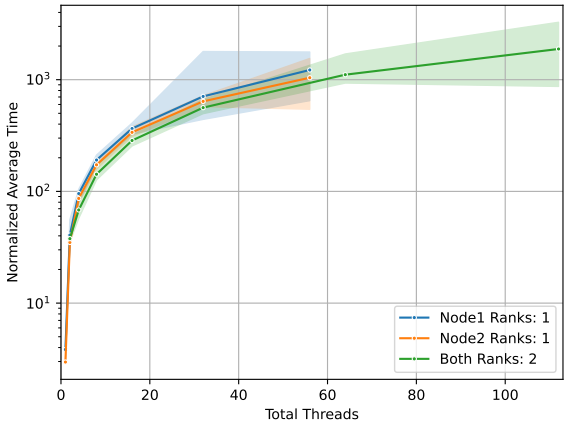

Figure 4: Scaling behavior for lock residing in the FAM device. The average time reported is normalized to a read-modify-write operation to FAM.

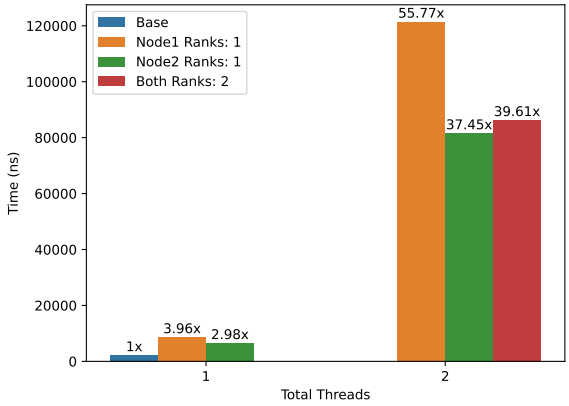

Figure 5: Speed up behavior between single thread setups in one and two nodes.

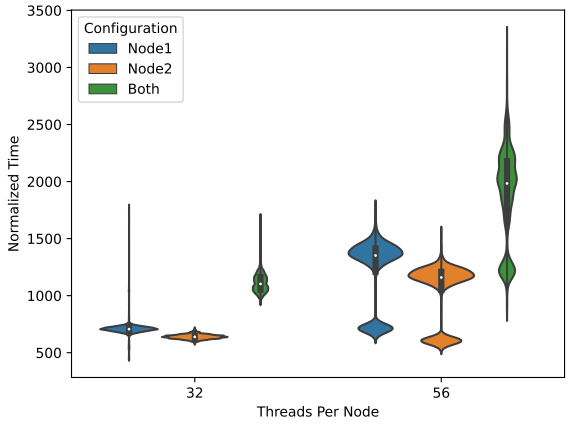

Figure 6: Violin plots for the distribution of observed locking times on the upper range of the thread numbers (full socket)

locking operation. Moreover, we present the maximum time observed for a tournament lock of two threads as it defaults to a single Peterson lock.

When running on node 1 with a single thread, we see an increase of around 4x. On the other hand, while running on node 2 with the same setup, we see a 3x increase. The difference can be attributed to the intra-locality effects of running on a socket with a longer latency to the PCIe. For two threads, node 1 and node 2 increase to 55.77x and 37.45x, respectively. The significant increase in time is due to contention since the number of small locked critical sections performed remains constant across experiments. The cost of the contention is greater in node 1 due to the increased latency to FAM. When a single thread per node competes for the same lock, we see a performance of 39.61x. This indicates that while we pay a penalty for node 1's increased latency to FAM, we contend less thanks to the offset access latencies.

On the other end of the spectrum, we explore the performance of our locks when utilizing a full socket per node. Figure 6 shows this distribution of normalized time per locked critical region as a violin plot. The single-node, 32-thread experiments showcase the extra latency needed by node 1 relative to node 2 with a larger mean and longer tails. When using 32 threads on both nodes simultaneously, we see a binomial distribution corresponding to the two different latencies. The other set of results showcases the entire socket (56 threads per node) and shows a binomial distribution for all experiments plus the same behavior as the previous set. When using both nodes totaling 112 threads, we observe a larger range with more outliers, highlighting the effects of contention.

Algorithm 2: Ticket Lock with proportional backoff

```
1  begin
2      Record L:
3          nextTicket ← 0;
4          nowServing ← 0;
5      Function acquireLock(L):
6          myTicket ← fetchAndIncrement (&L.nextTicket);
7          while True do
8              sleep(myTicket − L.nowServing);
9              if L.nowServing == myTicket then
10                 return;
11     Function releaseLock(L):
12         L.nowServing ← L.nowServing + 1;
```

6 DISCUSSION

6.1 Current Limitations of Software Consistency

One of the major performance detriments in our experiments was contention. Hence, we discuss how several lock methods designed

Table 1: Lock characteristics, when dealing with contention, high means that a lock implementation is susceptible to it, and low means that they have features to deal with it; while when reading the vertical locality column, low means that node's memory locality was not considered in its design and high means that the lock was designed to exploit it. Memory footprint is calculated in variables used.

Lock Name	Mem Footprint	Num Atomics	Contention	Vertical Locality
Ticket	2	1	High	Low
Anderson	numProcs + 1	2	Medium	High
MCS	(Up to numProcs) * 2	2	Low	High
Peterson	3	0	Low	Low
Tournament of Peterson	(numProcs - 1)*3	0	Low	Low

Algorithm 3: Anderson's Queue Lock

```
1  begin
2      Record L:
3          slots[0 .. numprocs - 1] ← [1, 0, 0...];
4          nextSlot ← 0
5      Function acquireLock(L, myPlace):
6          myPlace ← fetchAndIncrement (&L.nextSlot);
7          if (myPlace % numProcs) then
8              atomicAdd (&L.nextSlot, -numProcs);
9          myPlace ← myPlace % numProcs;
10         while L.slots[myPlace] == 0 do
11             ;
12         L.slots[myPlace] ← 0;
13     Function releaseLock(L, myPlace):
14         L.slots[(myPlace + 1) % numProcs] ← 1;
```

Algorithm 4: MCS Lock

```
1  begin
2      Record I, L:
3          next ← nil;
4          locked ← false
5      Function acquireLock(L, I):
6          I.next ← nil;
7          predecessor ← fetchAndStore(L, I);
8          if predecessor ≠ nil then
9              I.locked ← true;
10             predecessor.next ← I;
11             while I.locked == true do
12                 ;
13     Function releaseLock(L, I):
14         if I.next == nil then
15             if compareAndSwap(L, I, nil) then
16                 return ;
17             while I.next == nil do
18                 ;
19         I.next.locked = false
```

to alleviate contention in multithreaded shared memory environments may be ported to our testbed. The research presented in [15] explained the advantages and disadvantages of various types of spin locks. The design iterations between these types usually involve a trade-off between memory and contention avoidance, plus exploiting the vertical memory hierarchies of current processors. As part of our discussion, we selected three locks, highlighting their behavior and applicability to FAM. A summary table for all these locks, plus our Peterson implementations, is provided in Table 1.

In this setup, the lock has a constant memory footprint of two (lines 3 and 4). A proportional backoff is used to help to reduce the number of requests. The assumption is that since we know the participant's place in line, we can sleep for a time proportional to the difference between the current place and the serving number (Line 8). However, this implementation requires an atomic operation (Line 6), and the *nowServing* variable must be coherent to function correctly. Moreover, just using the two variables is insufficient to combat contention, and the resulting coherency actions from spinning might create even more traffic, especially since all threads will spin on *nowServing* (hence, there will be more contention).

The next evolution is Anderson's queue lock (Algorithm 3). In this setup, the lock's required memory is expanded to an array of slots as large as the number of participants (Line 3). This array ensures that each participant will have its variable to check instead of a centralized flag. The array elements are usually tailored (aligned and padded) to fully accommodate the underlying memory hierarchy and prevent spurious coherency actions. As such, this lock trades off memory space for contention avoidance. Finally, it increases the number of atomics to two (Lines 6 and 8).

The final lock selected is the MCS lock (Algorithm 4). This lock provides the best trade-off between contention reduction, atomic operation usage, and memory footprint. It is based on the idea that each participant will register to a list with only two variables per entity (Lines 3 and 4). Each queued entity will spin on its local copy until its successor signals it (Lines 12 and 19). In the worst-case scenario, the MCS's memory footprint can grow as large as Anderson's; however, it should be safe to assume these cases are rare. By having the spin variables local to the threads, the lock can take advantage of entirely spinning in its memory hierarchy (i.e.,

cache) until it is signaled by the successor (through this memory location). Finally, as Anderson, two atomic operations are required to be fully implemented.

All three locks have showcased good performance behavior in multithreaded shared memory systems [15]. However, applying them to our FAM environment will produce subpar results because they rely on (vertical) hardware coherence. In particular, variables used in spinning are cached.

The Anderson lock exhibits a better contention profile than the ticket lock at the expense of memory. Each flag is only contended on by two threads (similar to our Peterson lock). Implemented in FAM, the Anderson lock would require additional flushes, making its behavior similar to our tournament lock without the $O(log(n))$ complexity.

In the case of the MCS lock, the successor flags require global visibility. In a FAM-based implementation, the locking entities might reside in disjoint memory address spaces. Compared to single node-based implementations where the flags can be dynamically allocated quickly and globally (using malloc or alloca), the FAM-based implementation would require allocating globally visible device memory, which is expensive in our current testbed. Further, each spin needs to be *globally flushed*, completely negating the locality advantages of this lock. Lastly, by preallocating memory, our lock degenerates into an Anderson lock using a linked list instead of an array.

Based on our analysis, we conclude the importance of hardware coherence for modern locks. In particular, atomic operations are useful in reducing the memory footprint as shown by the ticket and MCS locks. In addition, Anderson's and MCS locks show how contention management can be transformed into cache coherency operations. Together, these results highlight the importance of the hardware support required to achieve fine-grain data sharing.

6.2 Programming Model Implications

Based on our findings, creative solutions are required to achieve CXL's goal of fine-grained data sharing. Given an overhead 3000 times our read-modify-write baseline, an overly conservative estimate would require a critical section on the order of 10000 times a read-modify-write operation to achieve an order of magnitude amortization. More detailed studies should show how much contention will decrease with the size of the critical section to provide a more concrete estimate. The impact of the overhead of this size pushes back on the EM as they seek to overlap the costs of tasking and data movement with useful work.

While CXL as a standard does not advocate for a single EM, its goal of achieving fine-grained data sharing supported by vendors gives hope for a model with programmability and performance akin to shared memory models like OpenMP [16] [2]. The benefit of a model like OpenMP is its balance of programmability and performance, particularly demonstrated by its parallel loops. However, since OpenMP works in a single shared address space, it has a monolithic view of memory where NUMA effects are not considered. On the other hand, dataflow-inspired, fine-grained models capable of better utilizing compute resources across NUMA domains like HPX [17], Legion [18], and ARTS [19] sacrifice some degree of simplicity and programmability by scaling across nodes and devices. Much of the added complexity and overhead comes from the discrete process spaces utilizing distributed nodes. In both cases, fine-grain synchronization is required to reduce the overhead from their runtime implementations. This overhead, combined with the cost of data movement, drives the task granularity.

CXL has vast potential to impact the EM's design space. By providing low latency coherent access to shared memory across nodes and devices, it presents a large (albeit not monolithic) memory useful for user and runtime data. From the user's perspective, FAM readily provides a means for data sharing with high read-to-write ratios or page-level interleaved access. With better support (i.e., atomic operations and hardware-base coherency), the granularity of access can be reduced. For runtimes, placing work queues into low(er) latency shared data would allow distributed work sharing across nodes without paying high network messaging costs. Such a load-balancing scheme would allow lower overhead, leading to finer-grain execution across nodes and devices.

In our work, we highlight locks as a critical step to enabling fine-grain synchronization and concurrent data structures, which are used in the runtime implementations of the mentioned EMs. Suppose hardware synchronization can not be supported or only in part (e.g., a minimal set of atomic operations). In that case, innovative solutions are required to implement these central runtime components to realize CXL's potential for fine-grained EMs.

7 RELATED WORK

The meta-study presented in [4] showcases an in-depth analysis of the impact of disaggregated memory across all software and hardware aspects in the context of data centers and their unique workflows. The fifty studies analyzed found that data sharing and its associated coherency and consistency implications are ignored or briefly touched as not applicable to their workflows. This point is critical since they recommend that if a third party wishes to use a disaggregated memory system in a multithreaded context, it should be analyzed and studied in terms of overheads of sharing, coherence, and movement compared with large deployment solutions (virtualization, message passing, etc.). We are starting this trend in this paper by analyzing the costs of a synchronization staple in a CXL-based system.

In the work presented in [20], the authors focus on HPC workflows and their applicability to disaggregated memory systems that enhance bandwidth and capacity when used as multi-tier memory systems. They also dispel common misconceptions, such as *memory bandwidth decreases when using disaggregated memory systems* and *applications always perform worse with disaggregated memories*. However, they point out that the interferences (i.e., contention) at several levels of FAM hierarchy might be one of the significant performance challenges with HPC workflows in this setup. They also highlight the need to exploit the vertical memory hierarchy composed of caches and prefetchers to successfully take advantage of this memory. Both of these points are well represented in our experiments, which showcases that software coherence is insufficient to exploit the vertical memory hierarchy in our setup fully, and

[2] We omit BSP-style message passing from our discussion since it is inherently coarse-grained by construction and due to the large overhead of moving data across the network.

the contention is just exacerbated by not being able to use vertical exploiting algorithms.

Work presented in [21] shows a more disparaging view of data sharing in data center workflows. The reasoning is that although data sharing leads to a decrease in total bytes moved, the increase in complexity and overhead required for coherency and consistency are too high to effectively implement them at scale. Although this observation is valid, they acknowledge that some workflows might require these capabilities. Their suggestion is to either minimize its use or avoid it. In many of our HPC workflows, this would require an extensive software restructuring or a heavy emphasis on system software and hardware co-design efforts.

In the current discussion, the selection of a programming model affects how synchronization and consistency/coherency are presented to the user and how the system software approaches it. In [22], the authors explore exploiting a disaggregated memory setup with its extreme heterogeneity and plug-and-play architecture. Their conclusion used a flavor of dataflow computation to represent workflows across domains. Although coherency and consistency are not mentioned explicitly, dataflow designs are well known to exchange the need for them with memory duplication. Moreover, a subset of HPC workflow memory footprints can snowball with concurrency and resolution. This behavior might not be suitable for these types of programming models.

Finally, [23] provides an exciting view of providing data sharing using the OpenSHEM programming model [11] and a hardware prototype using multi-headed CXL controllers. They provide in-device atomics for fine and coarse synchronization. One of the main aspects of the discussion is how to deal with the granularity problem since being too fine or coarse-grained can decrease performance. Their observation is that their hardware system should work well in cases with high read and low write demands.

8 CONCLUSIONS

We have presented an exploration synchronization in a CXL-based disaggregated memory system by implementing locks without hardware atomics or hardware synchronization. We highlight the performance degradation of small critical sections thanks to contention and highlight how state-of-the art locks leverage the vertical memory hierarchy to achieve a better memory footprint and contention profile. Further, we discuss the opportunities of fine-grain CXL-based synchronization for advanced runtimes. Lastly, we highlight the need for innovation to support low latency synchronization to implement fine-grain, scale-out runtime systems.

9 ACKNOWLEDGMENTS

This work is supported by the US DOE Office of Science project “Advanced Memory to Support Artificial Intelligence for Science" at PNNL. PNNL is operated by Battelle Memorial Institute under Contract DEAC06-76RL01830.

REFERENCES

[1] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. An introduction to the compute express link (cxl) interconnect, 2024.

[2] CCIX Consortium. Introduction to CCIX. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf, 2019.

[3] Seokbin Hong, Won-Ok Kwon, and Myeong-Hoon Oh. Hardware implementation and analysis of gen-z protocol for memory-centric architecture. *IEEE Access*, 8:127244–127253, 2020.

[4] Mohammad Ewais and Paul Chow. Disaggregated memory in the datacenter: A survey. *IEEE Access*, 11:20688–20712, 2023.

[5] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: an exploration of passive disaggregated key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[6] Matthew Chapman and Gernot Heiser. vnuma: a virtual shared-memory multiprocessor. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 2, USA, 2009. USENIX Association.

[7] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 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*, HPDC '15, page 3–14, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Zhuocheng Ding. vdsm: Distributed shared memory in virtualized environments. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 1112–1115, 2018.

[9] Zhiqiang Ma, Zhonghua Sheng, and Lin Gu. Dvm: A big virtual machine for cloud computing. *IEEE Transactions on Computers*, 63(9):2245–2258, 2014.

[10] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. *SIGPLAN Not.*, 49(4):3–18, feb 2014.

[11] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Jeff Daily, Abhinav Vishnu, Edoardo Aprà, Jeff Hammond, Doug Baxter, Mikhail Shiryaev, Theresa Windus, Callum Pe, Hubertus van Dam, Andy May, David E. Bernholdt, Bert de Jong, Dmitriy Solovyev, and Kristopher Keipert. Global arrays. [Computer Software] https://doi.org/10.11578/dc.20200519.25, feb 2006.

[13] Brian R. Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C. Hale. Trackfm: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 401–419, New York, NY, USA, 2024. Association for Computing Machinery.

[14] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[15] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, feb 1991.

[16] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Specification 5.2*. 2021.

[17] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. Hpx – an open source c++ standard library for parallelism and concurrency, 2023.

[18] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[19] Kiran Ranganath, Jesun Firoz, Joshua Suetterlein, Joseph Manzano, Andres Marquez, Mark Raugas, and Daniel Wong. Lc-memento: A memory model for accelerated architectures. In *Languages and Compilers for Parallel Computing: 34th International Workshop, LCPC 2021, Newark, DE, USA, October 13–14, 2021, Revised Selected Papers*, page 67–82, Berlin, Heidelberg, 2021. Springer-Verlag.

[20] Jacob Wahlgren, Gabin Schieffer, Maya Gokhale, and Ivy Peng. A quantitative approach for adopting disaggregated memory in hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[21] Hasan Al Maruf and Mosharaf Chowdhury. Memory disaggregation: Advances and open challenges. *SIGOPS Oper. Syst. Rev.*, 57(1):29–37, jun 2023.

[22] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. Programming fully disaggregated systems. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 188–195, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. Memory sharing with cxl: Hardware and software design approaches, 2024.