Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS

Adarsh Patil
University of Edinburgh
adarsh.patil@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Nikos Nikoleris
Arm
nikos.nikoleris@arm.com

Nicolai Oswald
University of Edinburgh
nicolai.oswald@ed.ac.uk

***Abstract*—As cloud workloads increasingly adopt the fault-tolerant Function-as-a-Service (FaaS) model, demand for improved performance has increased. Alas, the performance of FaaS applications is heavily bottlenecked by the remote object store in which FaaS objects are maintained. We identify that the upcoming CXL-based cache-coherent disaggregated memory is a promising technology for maintaining FaaS objects. Our analysis indicates that CXL's low-latency, high-bandwidth access characteristics coupled with compute-side caching of objects, provides significant performance potential over an in-memory RDMA-based object store.**

We observe however that CXL lacks the requisite level of fault-tolerance necessary to operate at an inter-server scale within the datacenter. Furthermore, its cache-line granular accesses impose inefficiencies for object-granular data store accesses.

We propose Āpta, a CXL-based object-granular memory interface for maintaining FaaS objects. Āpta's key innovation is a novel fault-tolerant coherence protocol for keeping the cached objects consistent without compromising availability in the face of compute server failures. Our evaluation of Āpta using 6 full FaaS application workflows (totaling 26 functions) indicates that it outperforms a state-of-the-art fault-tolerant object caching protocol on an RDMA-based system by 21–90% and an uncached CXL-based system by 15-42%.

I. Introduction

The Function-as-a-Service (FaaS) model is quickly becoming the defacto standard for cloud developers. In FaaS, applications are composed as workflows of stateless functions, and the cloud provider then orchestrates and schedules the functions dynamically on a fleet of compute servers.

The stateless nature of functions is good for availability, scalability, and elasticity, but it inevitably forces state to be maintained externally. Indeed, data stores such as Amazon S3 [7] are used to maintain state and pass input/output data between the stateless functions in the workflow. These data stores are the backbone of FaaS platforms.

Splitting state and compute, however, has an intrinsic data movement cost. Our analysis of FaaS functions from the FunctionBench [43] and SeBS [11] benchmark suites shows that on average 96% of the execution time per FaaS function is spent in retrieving data from the S3 object store. Replacing the S3 object store with a RDMA-based in-memory object store improves the situation somewhat – with 51% of execution time spent in retrieving data – but the problem persists. Communication overheads still limits performance.

Insight: FaaS objects on CXL disaggregated memory. We observe that upcoming CXL-based hardware disaggregated memory [9], [70] is a promising avenue for maintaining FaaS objects. CXL pools memory resources onto a logically centralized, physically distributed, highly-available memory server, and allows compute servers to perform load/store remote memory accesses in hardware over the network. The memory server, as shown in Fig. 1, is equipped with specialized hardware controllers for performing frequent *data plane* operations and minimalist low-power processors to handle rarer *control plane* operations [31], [48]. Since CXL allows for loads and stores to be handled in hardware like in a traditional NUMA machine [49], [50], [74], CXL-based disaggregated memory allows for significantly lower latency and higher bandwidth compared to high-performance RDMA-based remote memory.

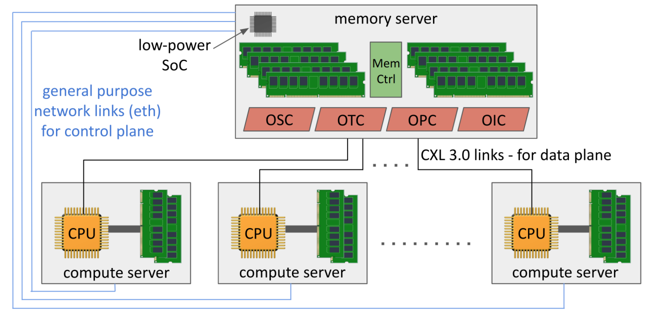

Fig. 1. Āpta system schematic (new controllers in red). The figure shows a CXL disaggregated memory system, where compute servers are connected to and cache data from a logically centralized (physically distributed and highly-available) memory server via a hardware load/store interface. Āpta augments the memory server with new controllers to support object-granular accesses and keeps the caches consistent with a fault-tolerant coherence protocol. Sec III-D1 defines the micro-architecture of Āpta's memory server controllers: object serving controller (OSC), object tracker controller (OTC), object persistence controller (OPC), object invalidation controller (OIC)

Furthermore, the recently announced CXL 3.0 specification [10] allows the compute server caches to transparently cache data from a *shared region* in disaggregated memory, which matches well with the access patterns of a FaaS object store. Because FaaS functions typically share objects between them, object accesses exhibit significant locality and are amenable to caching. Therefore, such object caching and the use of a locality-aware scheduling policy (schedule functions where objects it uses are cached) has the potential for significantly reducing data movement.

Our analysis shows that a FaaS object store over a CXL-based disaggregated memory system with support for object-

granular accesses, coupled with a locality-aware scheduling policy can improve performance of the aforementioned FaaS functions by a significant 2.3× over the state-of-the-art RDMA-based object store. This is the performance opportunity Āpta targets.

CXL provides consistency but forgoes availability. To preserve flexibility and maximize throughput, a cloud system *dynamically* schedules a function on any available compute server. This dynamicity combined with compute-side caching results in FaaS objects being replicated, and it is imperative that the replicas be kept consistent. Because compute servers can fail or be unresponsive in the datacenter, it is important that the consistency protocol remains *available* in the presence of such failures: i.e., the protocol should not block indefinitely if any of the servers fail. Alas the CXL 3.0 protocol [76] (which is a conventional protocol that enforces the Single-Writer-Multiple-Reader (SWMR) invariant [65]), while enforcing strong consistency, fundamentally blocks in the presence of server failures: if a server sharing an object fails, a write to that sharer from any other server could indefinitely block waiting for an acknowledgment from the failed sharer. Thus, this naive application of a traditional multi-processor coherence protocol (non fault-tolerant) for distributed disaggregated memory leaves CXL systems vulnerable to system crashes.

Severity of the problem: Building system resiliency is an important problem as servers frequently fail or become unavailable in a datacenter environment. Google has observed that up to 25% of service-level disruptions are caused by machine-level failures [6]. A study of errors in even the highest reliability petascale supercomputers has shown that network link and server faults causing job failures occur with a mean time between failures (MTBF) of 160 hours [14], [40]. Consequently, fault tolerance is a key tenet of FaaS platforms. This is precisely why FaaS applications have already embraced failures via idempotent functions [5], [28]: if a function fails while executing (e.g., due to a compute server failure) the FaaS function can simply recover by re-executing. Therefore, it is imperative that the underlying CXL-based object store operates correctly in the presence of such server failures.

Consistency & availability via fault-tolerant coherence. We transform a strongly consistent SWMR-enforcing coherence protocol into a highly-available protocol in the presence of compute server failures. The idea consists of two simple steps: *lazy invalidation* and *coherence-aware scheduling*. In the first step we move the invalidations out of the critical path of the write so a writer is not blocked indefinitely when a server caching the sharer fails. But because invalidations are moved off the critical path of the write, there is a window of inconsistency where caches may hold stale values. In the second step we make a simple change to the FaaS scheduler [47] allowing it to schedule functions *only* on servers where there are no pending invalidations – thereby enforcing strong consistency as well as availability. Āpta's method for transforming the non fault-tolerant coherence protocol into a highly-available one can easily be applied to upcoming versions of CXL.

TABLE I
TAXONOMY OF STATE-OF-THE-ART PROPOSALS

System	Caching support? (granularity, write-policy, inter-server sharing, coherence mechanism, sharer invalidation)	Hardware support?	Compute server fault-tolerance	Performance for object stores
S3 [7]	No	No	High	Low
Pond [50], Kona [8] ThymesisFlow [74]	Yes (cacheline, write-back, No, N/A N/A)	Yes	Low	Low
LegoOS [79]	Yes (page, write-back, No, N/A, N/A)	Yes	Low	Medium
Clio-KV [31]	Yes (object, write-through, Yes, No, N/A)	Yes	Low	Medium
MIND [48]	Yes (page, write-back, Yes, MSI, sync)	Yes	Low	Medium
OFC [64]	Yes (object, write-back, Yes, version-based - all reads require remote version match, No)	No	Low	Medium
Faa$t [77]	Yes (object, write-through, Yes, version-based - all reads require remote version match, No)	No	High	Medium
CXL 3.0 spec [10]	Yes (cacheline, write-back, Yes, MESI, sync)	Yes	Low	Medium
Āpta	Yes (object, write-through, Yes, SI, async)	Yes	High	High

Contributions.

1) We make the case for a CXL-based object store for FaaS with object-granular reads/writes (Sec II). Our analysis using stand-alone FaaS functions indicates that such a design can provide a 69× performance improvement over the Amazon S3-based FaaS system, and a 2.3× improvement over a RDMA-based system. We observe, however, that such a system must remain fault-tolerant, which the existing CXL protocol specification falls short of.
2) We introduce Āpta (Fig. 1) – a CXL-based object store that allows compute server-side caching of objects without compromising consistency or availability (Sec. III). Āpta is tailored for object-granular accesses and defines a fault-tolerant inter-server cache-coherence protocol that, together with the FaaS scheduler, enforces strong consistency and provides high-availability in the presence of server failures. We have verified safety and liveness of the protocol in a model checker.
3) We evaluate the performance of Āpta (Sec. IV) using 6 full FaaS applications (total of 26 functions) and show that it provides 21–90% execution time speedup over a state-of-the-art fault-tolerant RDMA-based object store and 15–42% speedup over a reliable CXL-based object store without caching.
4) We observe that amongst all state-of-the-art high-performance remote memories and object stores that support caching (Table I), Āpta has the highest performance, and the highest availability in the presence of compute server failures.

II. MOTIVATION & ANALYSES

In this section, we first demonstrate the compelling performance reason to migrate FaaS object store to a disaggregated memory system (abbreviated as DM). Next, we illustrate why DM, while providing improved performance, falls short of providing the level of fault-tolerance required for the FaaS paradigm. Finally, we highlight inefficiencies when existing cache line granularity DM is used to design an object store.

A. The performance potential of a DM-based object store

We compare the performance of FaaS functions from FunctionBench [43] and SeBS [11] benchmark suites[1] using three

[1]We exclude micro benchmarks and network benchmarks that are non-deterministic and sensitive to external system delays.

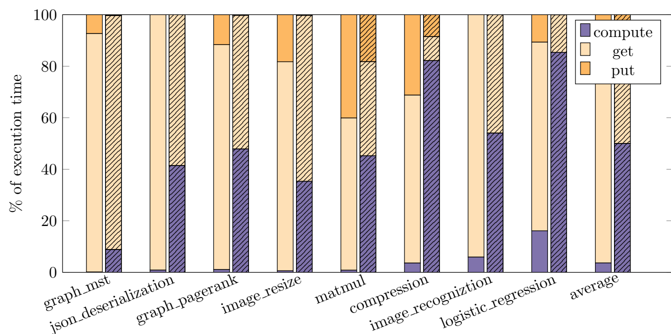

Fig. 2. Compute-to-Communication ratio in function execution with - (a) Amazon S3 (first bar) (b) in-memory RDMA store (second, striped bar)

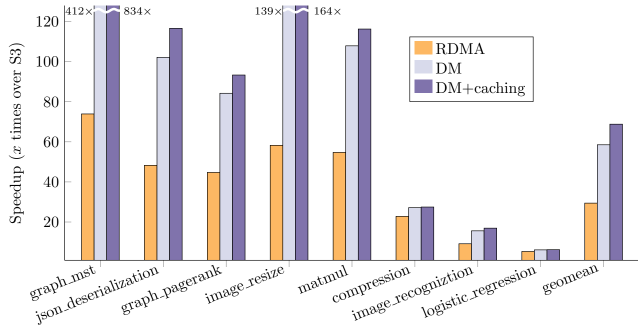

Fig. 3. Comparison of FaaS functions performance with various object stores (Baseline Amazon S3)

different object stores: Amazon S3, RDMA-based, and DM-based object stores. The functions execute two basic operations on the object store: `obj` ← `get(objID)` at the beginning and `put(objID, obj)` at the end, where `objID` is an identifier for an object `obj`. The computations in the middle of these functions are often unoptimized which hides the true bottlenecks in the system. We envision that high-performance frameworks such as Google TensorFlow [39] and Facebook PyTorch [38] will be adopted for FaaS in the future. We therefore ran the functions with Intel OneAPI [37] which applies vectorization, parallelization, cache blocking and other architecture specific optimizations.

FaaS functions experience high communication overheads with Amazon S3: When using the S3 object store, a `get` operation downloads the object from a remote S3 server into the compute server memory; post computation the `put` operation uploads an object from the compute server into a remote S3 server. We take the median of 100 executions accounting for cold function and tail latency effects [23], [86].

We observe that on an average 96% of execution time is spent in communicating data from/to the S3 object store (Fig. 2, all unstriped bars). This shows that the execution of FaaS functions in the cloud today is severely limited by the latency of accessing data from object stores. While S3 is based on disk based storage servers, it employs several optimizations like replication and sharding [7] to provide the best performance among today's production object stores.

In-memory object stores do not alleviate the communication overheads: High-performance RDMA-based in-memory object stores completely bypass the remote CPU to read (write) objects directly from (into) the memory of the remote object server [12], [30], [63]. The `get` and `put` operations were modified to use one-sided RDMA verbs, which runs over an Infiniband network (Mellanox ConnectX-3 NIC on PCIe-gen3 x16) [17], [58]. RDMA-based object stores are faster than traditional in-memory data stores that operate over general purpose ethernet networks like Redis [75], memcached [59] or Amazon ElastiCache [4]. However, even with such modern RDMA-based data store, on average 51% of execution time is still spent in communicating objects (Fig. 2, all striped bars).

Overcoming RDMA's Achilles heel: The RDMA-based approach has several fundamental characteristics that limit performance – the use of software libraries like libibverbs and libmlx4, the need to perform two DMA *data copy* operations (at source and at destination, copying data to/from RNICs Memory Region) and managing the memory regions with software-initiated per-server static connection queue pairs. Several works have analyzed these and other drawbacks of RDMA [29], [31], [49]. DirectCXL [29] quantifies that even with the same underlying physical interconnect, RDMA's irreducible overheads makes `get`/`put` operations 2.2× slower than CXL-based DM. DM is the new approach that chip manufacturers and cloud providers are investing in. DM overcomes the drawbacks of RDMA by allowing all data plane operations to be handled in hardware, thereby providing lower latency and higher bandwidth.

DM reduces communication overheads: The object is retrieved from a load/store semantic DM system. All standards for building such a DM system (GenZ [25], OpenCAPI [70]) have coalesced under the CXL umbrella due to their synergistic goals. Currently however, there exist only early prototypes: (i) OpenCAPI-based DM [74], providing RTT latencies of $950\,\mathrm{ns}$ and a bandwidth of $12.5\,\mathrm{GiB/s}$; (ii) CXL-based DM [29], providing a lower RTT latency of $500\,\mathrm{ns}$ and a higher bandwidth. We pessimistically model the worst-case latency and bandwidth of OpenCAPI for our DM system. Our modeled DM system lowers latency by 3× and improves bandwidth by 10× over the RDMA system [49], [74].

With DM, the fraction of execution time spent for communication in FaaS functions reduces to 13% of the total time, on average. This translates to a large reduction in execution times of the functions. Fig. 3 shows that the DM-based object store is able to achieve a 59× geomean speedup over Amazon S3 and a 2× speedup over the RDMA object store.

Caching - an additional benefit of DM: A CXL DM system transparently caches object cache lines in the (on-chip SRAM or DRAM) hardware caches of the compute server, thereby being served at a lower latency compared to a remote memory server access. Such caching is extremely effective for FaaS applications which exhibit good object access locality [62].

This is because full FaaS applications, defined a state machine workflow of multiple individual functions ("function chains"), demonstrate known communication patterns like producer-consumer and broadcast within them [87]. This communication implies that successor functions can potentially access objects produced by any of its predecessors. When functions read objects from compute server caches in the DM system their execution time further speedups by 2%-100% (Fig. 3 DM+caching, assumes a DRAM cache of DDR4-like latency).

Summary: Our analysis indicates that maintaining FaaS objects in DM and the caching benefits it provides largely mitigates the key performance bottleneck of the FaaS paradigm.

B. The lack of fault-tolerance in current DM systems

FaaS object stores, such as Amazon S3, are designed to provide fault-tolerant operation for a failure-prone datacenter environment. Object `get` and `put` atomically read and write entire objects, with all or nothing semantics. A `get` is also guaranteed to read the value of the most recent `put`, therefore providing a strong consistency model known as linearizability [7]. This greatly simplifies things for a FaaS developer who can simply assume that a `get` would return the object written by the most recent put in the workflow.

Enforcing strong consistency in the presence of caching. In the caching DM system, enforcing strong consistency for the FaaS execution environment can be challenging. For example, consider a simple workflow consisting of three functions: $f_1 \rightarrow f_2 \rightarrow f_3$, where f_1 and f_3 read object X, while f_2 writes to X. Further, let us assume that f_1 is assigned to server C1 while f_2 is assigned to C2. When f_1 executes on C1, it would cache the object in C1. When f_2 writes the object, it would render the value cached in C1 stale. Suppose the FaaS scheduler chooses to schedule f_3 on C1, f_3 would then read the stale value of X, violating strong consistency.

One way to enforce strong consistency in the presence of caching is to employ a cache coherence protocol. Conveniently, CXL 3.0 specifies an inter-server MESI-based coherence protocol [76], that enforces the SWMR invariant. In the above example, the write from f_2 would invalidate the cached copy of X in C1, ensuring that when f_3 is scheduled on C1, it will read the most recent value written by C2, and not the stale value.

Whither Fault tolerance? It is imperative that the aforementioned inter-server cache coherence protocol operates correctly even when compute servers fail or become unavailable. (In this work we assume that the DM server is kept highly-available using techniques such as replication [72], [79] and power redundancy.) Alas, traditional coherence protocols can block in the presence of such failures. Consider the same example where f_1 caches object X in server C1. When f_2 executing on C2, writes to X, the coherence protocol would send an invalidation to C1 which holds the object. Now, should C1 fail or become unreachable the write from f_2 would simply block, waiting for an acknowledgment, thereby rendering the system unavailable. Even if C1 does not fail but is simply slow to acknowledge (e.g., due to network congestion), the write from f_2 would be impacted, which can lead to high tail latency – a critical issue for FaaS platforms [85].

C. Inefficiencies of DM for object stores

Current DM system standards specify fixed fine-grain data access, caching and coherence mechanisms. However, object reads/writes typically have widely variable sizes, ranging from bytes to MBs [18], [62]. This causes two key inefficiencies. CXL enables compute servers to read cache lines from memory server while objects frequently span multiple cache lines. Hence, reading an object will incur multiple round trips to the DM, owing to limited MSHRs (miss status handling registers).

Second, CXL permits single cache line atomic write while a `put` must atomically write an object of multiple cache lines to the DM. This incurs additional latency for software write-ahead-logging i.e., undo/redo logs. Our analysis for the above benchmarks shows that a CXL object store will incur an average of 32% and 89% higher latency for `get` and `put` respectively, compared to an optimized object granular DM (evaluation methodology in Sec. IV).

Summary: Supporting compute server caching mandates a fault-tolerant coherence protocol that enforces strong consistency in the presence of compute server failures. CXL-based DM systems fail to provide this. Second, existing CXL cache line granular accesses are ill suited for FaaS object granular accesses.

III. Design

Āpta's goal is to design a DM-based object store for FaaS applications (Sec. III-A) that provides fault-tolerant coherence (Sec. III-B) and optimum performance (Sec. III-C). To accomplish this, Āpta designs DM hardware controllers and modifies runtime software (Sec. III-D). Sec. III-E walks through the working of the entire Āpta system when executing real-world FaaS applications.

A. Setting the stage: Designing a DM-based object store

This section describes how Āpta leverages the features of a CXL 3.0 based DM system to construct an object store.

(a_1) **Sharing objects between FaaS functions through DM** ► *Extend shared memory IPC*
CXL 3.0 allows compute servers to access a shared memory region on the memory server. The compute server OS discovers and manages this CXL memory device as per UEFI/ACPI specifications [84] and exposes the DM address space as an extended CPU-less NUMA region [49], [50], [70].

In Āpta, FaaS functions execute as independent processes on compute servers. To access a shared object, the `get` and `put` operations map a DM memory region (containing the object) into their virtual memory using shared memory inter-process communication (shmem IPC) [77]. The shmem IPC API is enhanced to allow function processes on different compute servers to mount/access a shared memory region.

To illustrate, Fig. 4 shows the two functions f_1 and f_2 executing on server C1 and C2 respectively, sharing the object

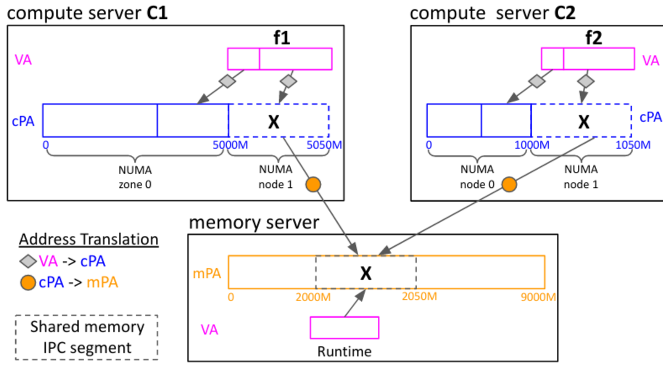

Fig. 4. FaaS object sharing through DM: organization and addressing

X of size 50MB through the DM system. On each compute server, the shmem IPC segment, where X resides, is located in an extended NUMA physical address space (cPA - blue dashed regions in Fig. 4). Just as in CXL, the access controls and page tables for end-to-end address translation from compute server process virtual address (VA) to the memory server physical address (mPA) are initialized and setup by the OS[2]. Once mapped, the object is accessed by the CPU (during the compute phase of the function) using load/store on cPA address. When these accesses miss in the LLC, the request is routed to the "home node" of the extended NUMA region (DM controller on the compute server). The DM controller uses the mapping to verify permissions and provides the memory server physical addresses (mPA) to be accessed.

(a_2) **Caching objects in compute server caches** ► *Defining a caching policy*
Āpta introduces minimal changes to compute server caches, making them almost oblivious to disaggregation (in the spirit of CXL). The `get` operation, when mapping the shared object, sets the memory region of objects larger than size of the LLC as uncacheable using PAT or MTRR [71][3]. On an LLC miss, the cache line is read from DM and allocated in compute server caches. Similarly, objects are also write-allocated in the LLC. This policy allows retaining data in the LLC for any expected future reuse.

Importantly, the `put` operation immediately writes all modified cached lines through to DM, making the caches effectively write-through. This policy allows tolerating compute server failures since a compute server LLC never holds the only copy of the object, and the memory server always holds a valid copy. The LLC silently evicts any of the DM cache lines which are in shared state i.e., the LLC does not issue a PutS coherence request to the directory. This saves interconnect network bandwidth and avoids LLCs having to evict entire objects if one of the object's cache lines is evicted.

[2]CXL uses Address Translation Services (ATS) defined in PCIe Specification for translation of cPA→mPA. The compute server OS sets the translation table base address register - ZMMU [26] or extended page table pointer [45].

[3]Other object cache allocation policies can be employed - e.g., fraction of LLC capacity per CPU, dynamically based on predictors of object hotness or reuse potential [64] etc. The exploration of allocation policies is orthogonal.

(a_3) **Exploiting the locality provided by caching** ► *Locality aware scheduling policy*
The FaaS runtime schedules each function invocation on compute servers. The scheduler makes intelligent heuristic decisions to achieve lowest execution latency for the functions execution by accounting for various factors [23], [56], [86]. In Āpta, this runtime scheduler is used to exploit object locality by scheduling invocations on compute servers where cached objects will be or are likely to be reused. This allows functions to benefit from lower latency for object access.

This is done on a best effort, maximization basis. For generality, in this work, the scheduler heuristically assumes that the function has a high likelihood of accessing any object consumed or produced by any of its predecessors and picks a compute server where most predecessors executed.

B. Fault-tolerant Coherence Protocol

The conventional MESI coherence protocol, specified in CXL 3.0, does not provide reliable operation in an environment where compute servers can fail independently. We now detail Āpta's highly-available fault-tolerant coherence protocol that is designed for a failure-prone cloud environment.

(b_1) **Keeping the cached objects on compute servers coherent** ► *Tailored coherence mechanism and protocol*
"Simplicity is prerequisite for reliability" - Edsger Dijkstra

Simplified coherence: Recall that a FaaS function reads the object from the memory server and the compute server caches it in shared state; a `put` writes-through to the memory server and subsequently invalidates all sharers of the object in other compute servers. This eliminates the need for Modified or Exclusive states and reduces the inter-server protocol to two stable states - Shared and Invalid. This simplified coherence protocol, designed for the execution model of FaaS functions, hardens Āpta against compute server faults.

This Āpta protocol is layered hierarchically over and above intra-server coherence protocol. The intra-server coherence protocol is unchanged and regardless of this protocol Āpta enforces different policies in the inter-server protocol. This hierarchic organization allows Āpta to track sharers at compute server granularity (not individual caches within them). The Āpta protocol is employed for all requests from the compute server to the DM server.

Coarse granularity tracking: The use of DM in FaaS systems is restricted to sharing objects. Thus, it suffices for Āpta to use variable-sized object granularity tracking for the coherence protocol, as opposed to cache line level tracking in traditional chip level coherence protocols. In other words, we use a single state to encapsulate the state of all cache lines within the object. This is tracked using an object unique triplet of (`objID`, base mPA, size).

(b_2) **Provide high-availability while enforcing strong consistency** ► *Lazy invalidation of sharers with coherence-aware scheduling*
Recall, to enforce strong consistency of the caches, a `put` completes only when all servers caching that object are

invalidated; therefore, put can block when servers fail. This invariant of any conventional coherence protocol called Single-writer-multiple-reader (SWMR) is enforced by synchronously invalidating all sharers in the critical path of the put.

In Āpta, the sharers are sent an invalidation message asynchronously, i.e., the put is acknowledged immediately without waiting for the sharers to be invalidated. The sharers that are sent invalidations are tracked off the critical path until they acknowledge the invalidation messages.

This lazy invalidation policy: (a) allows the write to be acknowledged at lower latency thereby improving performance and (b) more importantly, because writes need not wait for sharers to be invalidated, there is no risk of writes being blocked, thereby ensuring fault-tolerance.

Whither Consistency? Note, however, that this asynchronous protocol described above could violate SWMR (and hence linearizability). This is because at the instant the put is acknowledged there may be cached copies in other servers yet to be invalidated.

Lazy linearizability with scheduler support. Āpta enforces linearizability lazily using a combination of the coherence protocol and the FaaS runtime scheduler. More specifically, Āpta never schedules function invocations on servers with pending invalidations – the *§Scheduling Correctness Criterion*. This correctness criterion ensures there is no risk of reading any yet-to-be-invalidated stale objects present in the caches. More precisely, we are now in a position to assert Lemma 1.

Lemma 1. *The coherence protocol ensures that a* get *returns the value of most recent* put *to that object.*

Proof. Consider a get to object X. When the get is about to be scheduled, there are either pending invalidations to X or there are none. If there are no pending invalidations, there are no stale values and hence the get will return the latest value as per the original synchronous protocol. If there are one or more pending invalidations, the scheduler ensures that the function containing the get is not scheduled on those servers with pending invalidations, and hence there is no risk of get reading a stale value. □

Thus, the Āpta coherence protocol ensures that the caches on the compute servers where functions execute are strongly consistent. Meanwhile, caches in compute servers where functions are not executing can be stale without affecting consistency. Another benefit of Āpta's lazy invalidation protocol is the ability to perform coherence actions at line-rate. This is particularly important for processing packets in the data plane on DPUs or SmartNICs in the network [21], [69].

C. Addressing the inefficiencies of DM

This section describes Āpta's optimization to adapt the DM system for object idiosyncrasies.

(c_1) **Object-granular reads** ▶ *Via bulk cache line loads*
Recall, for each object load request that miss in the LLC, the DM controller on the compute server issues a single cache line read request over the interconnect making it inefficient for objects spanning multiple cache lines.

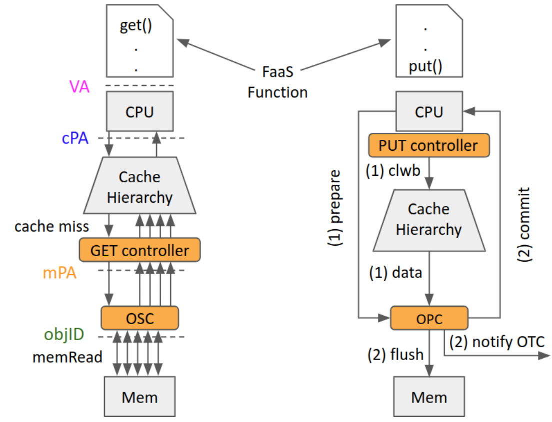

Fig. 5. Operation of get (left) and put (right) using controllers on the compute and memory servers (controllers shaded in orange).

CXL 3.0 [10] does provides fixed block request semantics (2 or 4 contiguous cache lines) with the block request size to be specified in advance. However, objects have more variable block sizes and compute server LLCs cannot specify the block size in advance as they operate oblivious of objects.

Āpta builds on CXL 3.0 to provide variable sized, bulk cache line requests. It *bulk reads* all the objects cache lines into the compute server cache in one round trip to the memory server (similar to [12]), providing the lowest possible latency and maximizing the interconnect bandwidth utilization. This process is illustrated in Fig. 5, left. The GET controller (optimized DM controller) issues the LLC's read request over the interconnect. The memory server reads all cache lines constituting the object from DRAM memory. It replies with all these cache lines and squashes/ignores any immediate requests for this object from that compute server. The GET controller receives all the prefetched cache lines and inserts them into the respective cache sets in the LLC. The LLC forwards the demand miss cache lines to lower level caches and the CPU.

(c_2) **Object-atomic writes** ▶ *Transactional atomic durability*
Recall, CXL permits atomic writes of single cache lines which forces a put to use software transactions (*libpmemobj* API) to write an object of multiple cache lines atomically to the DM system. These transactions use software logging (undo or redo) which adds significant number of additional instructions per transaction, hurting latency and throughput.

Āpta provides hardware transactions for object atomic writes to improve performance (similar to [32], [54]). The hardware transaction ensures that when an object put is executed, either the entire object is persisted[4] or, in case of failures, any partial writes are collectively discarded. If the transaction succeeds, the memory server overwrites the new version into the objects memory area. If the transaction fails, it is retried, assuming the cause of the failure is transient. If a retry threshold is exceeded, the exception is reported to

[4]Recall the memory server is usually kept highly-available and persistent

an external FaaS infrastructure system and the entire function execution is considered to have failed.

In the compute server, the PUT controller, co-located with the CPU, flags for persistence all the cache lines written by an object `put`. The controller orchestrates an atomic transaction, using a one-phase commit protocol with the memory server (Fig. 5, right). When the memory server issues a commit response, the PUT controller clears the persistence flags.

D. Realizing Āpta's architecture

We now detail the memory server components - data plane controllers, control plane software and the interaction between them required to realize Āpta. We also describe in detail the coherence protocol sketched out in the previous section.

1) Micro-architecture of data-plane controllers: The data plane on the memory server is composed of a conventional memory controller and the Āpta controller. The Āpta controller is composed of four modular sub-controllers as shown in Fig. 1. This section details the micro-architecture of these controllers, each providing a certain functionality.

Object Serving sub-controller (OSC):
≻ *Function:* Serving objects (bulk cache lines) when the GET controller requests an object's cache line.
The OSC translates the requested mPA to an object triplet. For this, OSC walks the *object mapping data structure*, populated by the FaaS runtime object manager (See III-D2). Similar to page tables, this translation latency can be reduced by using TLBs, page walk caches, cuckoo filters [82] etc. Once the physical address of the object is retrieved, the OSC issues memory access requests to the memory controller and replies to the compute server once it receives the data from it.

Object Persistence sub-controller (OPC):
≻ *Function:* Persists an entire object atomically into DM.
Recall, an object `put` initiates a one-phase commit protocol, between the PUT controller on the compute server and OPC on the memory server, to atomically write all the objects cache lines. As shown in Fig. 5 (right), first, the PUT controller on the compute server CPU sends a prepare message with the `objID` to be written. Then, it issues cacheline writeback (*clwb* [36]) for all the cache lines that are written by the `put`. The data from these cache lines, resident anywhere in the cache hierarchy of the compute server, are flushed to the memory server. OPC uses buffers (either using SRAM registers or a dedicated DRAM area) to temporarily stage cache lines written back from the compute server. OPC expects to receive a fixed number of cache line writes to complete the object write (inferred from the object triplet). Once it receives all cache lines of the object, it replies with a commit message, marking the end of transaction. OPC notifies object tracker controller (OTC) of the competition of an object write and flushes/drains the buffers to the memory controller.

Object Tracker sub-controller (OTC):
≻ *Function:* Directory for the Āpta coherence protocol.
Similar to a conventional directory, OTC maintains entries about the state and sharer vector for each object triplet. The

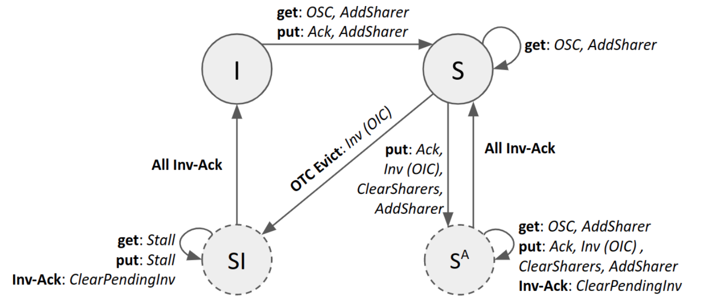

Fig. 6. OTC(Directory): Complete coherence protocol. I, S are stable states; SI, S^A are transient states

OTC directory represents the ordering point for all requests. The directory is inclusive of all the compute server LLCs i.e., it holds directory entries for a superset of all objects cached in all the compute server LLCs. A miss in this directory cache indicates that the object is in state I. The sharer list tracked is not precise since the compute server LLC silently evicts blocks in shared state.

The protocol: OTC uses a simplified coherence protocol with Shared and Invalid stable states, avoiding the Modified and the Exclusive States. This is in line with the CXL specification as it flexibly allows implementations to use fewer stable states in the protocol. (We discuss further details of the CXL protocol in Sec. V-1.) Most coherence protocols involve transient states since transition from one stable state to another is not typically atomic [65]. Āpta re-purposes a transient state to account for the asynchronous invalidations. Fig. 6 illustrates the transition diagram for the OTC directory controller (with events and actions on stable and transient states). For ease of explanation, this protocol assumes that FaaS applications are race-free, i.e., no `put` or `get` can occur during an ongoing `put`. (However, the actual Āpta protocol can handle buggy FaaS applications with races as well.) A `put` event is triggered on the completion of an object persistence transaction by the OPC; a `get` event is triggered at the beginning of an object serving request by the OSC. An "Invalid" state for an object implies it is not cached in any compute server. A "Shared" state implies the object is cached in readable state in one or more compute servers. The transient state S^A signifies that the new version of the object is cached in **S**hared state in one or more compute servers and there are pending invalidation-**A**cknowledgments from one or more compute servers for the old version of the object.

Suppose the directory receives a `put` for an object currently in shared state. Once the `put` transaction completes, the directory performs 3 actions in parallel - acknowledges the write, notifies the object invalidation controller (OIC) to send invalidation messages to all prior sharer compute servers, clears the old sharer vector and adds the compute server that requested the `put` as a sharer for the new version of the object (Recall the caching policy is write-allocate - compute server retains the object after a `put`). The directory then transitions to S^A until it receives all invalidation-acknowledgments. While

in this state the directory can still service a `get` or `put` request for the object. For a `get`, the OSC responds with the latest object version, satisfying Lemma #1. Once OIC notifies that all outstanding invalidation-acknowledgments are received, the directory transitions back to shared state for the object.

The organization: The directory is organized as a standalone, set-associative directory cache structure at object granularity. When a directory cache set is full, the directory controller evicts a cold, shared object in the set. It issues an invalidate to all compute server sharers that cache this object and transitions to transient state SI. Any requests to the object while in this state, are stalled until all invalidations are acknowledged and an entry becomes available. Sizing the directory cache appropriately and correctly identifying cold objects can ensure that operations can continue at line-rate, without stalling.

Object Invalidation sub-controller (OIC):
≻ *Function:* Invalidates stale objects cached in compute servers.
The OTC requests the OIC to invalidate an object triplet (`objID`, base mPA, size) on a set of compute servers. OIC issues invalidation messages to the compute servers for all the object's cache lines. At the compute server, the challenge however is translating the mPA address of the object to cPA address to issue invalidations to caches. This is achieved using an efficient object based reverse mapping [13], implemented in Linux for reserve mapping virtual memory (rmap chains)[5]. This reverse mapping is used by system calls like mmap, munmap, madvise etc. However, this is an expensive software call (measured to be $\sim$1.4μsec per call) invoked by GET controller using interrupts and adds significant time overhead. Recall, in Āpta, invalidations are out of the critical path and hence this does not affect performance. Finally, the GET controller sends invalidation-acknowledgments.

The OIC tracks the number of invalidation-acknowledgments that are outstanding from each compute server using a counter. It notifies the OTC when all the invalidation-acknowledgments are received.

2) Control-plane software: Āpta modifies existing FaaS runtime control-plane software [47]. This software runs on the low-power SoC of the memory server. We outline the changes required in two of these components and describe their interface to the Āpta hardware controllers.

Executor Manager (EM):
≻ *Function:* Responsible for scheduling and tracking the execution of the state machine workflow of FaaS applications. EM selects a suitable compute server to schedule a function invocation and passes the invocation parameters to the function sandbox. EM scheduler is guided by the performance and correctness criteria when scheduling function invocations.
≻ *Hardware interface:* When scheduling functions, if the set of all objects to be accessed by the function is unknown (not declared), the scheduler queries the OIC to exclude all compute servers which have pending invalidation-acknowledgments. If the set of objects to be accessed by a function are declared in the state machine workflow, the scheduler looks up the object in OTC to determine where scheduling can be beneficial (current sharers) and if any compute servers are to be excluded (invalidation-acknowledgment pending).

Object Manager (OM):
≻ *Function:* Responsible for memory allocation and deallocation of objects in the memory server.
The objects are allocated in mPA in fixed bucket sizes (rounded up to the nearest fixed bucket size). The buckets are allocated as a contiguous physical memory address range, aligned at the cache line boundary in the memory server. This memory allocation strategy is akin to the memcached slab allocator [60] The OM runtime stores an *object mapping data structure* of mPA to unique `objID`, at a fixed location in memory. This data structure is organized as a radix tree followed by a trie[6].
≻ *Hardware interface:* For serving objects, the OSC controller reads the *object mapping data structure* written by the OM runtime (from the fixed location in memory) and responds to the compute server mPA request with object-granularity bulk read semantics.

E. Putting it all together

Fig. 7 illustrates the application state machine workflow of three real world FaaS applications [46], [51] with the objects accessed by each function and annotated with an instance of scheduling decision made by the EM on a cluster of compute servers (C1 to C4) connected to access the Āpta object store.

We walk through the working of Āpta with the sentiment analysis application (Fig. 7, App 3) that evaluates customer reviews for products of a company and is triggered when the collated raw reviews file (csv) is uploaded to the object store.

When *read_csv* function on C1 receives a invocation trigger, the `get` call (*rdata = get("raw_data.csv")*) maps *rdata* to the shmem IPC region, located in the DM address range on C1. When *rdata* is accessed, the LLC miss triggers a request to the GET controller. The OSC controller responds with a set of cache lines of the object. All subsequent accesses to *rdata* in the computation hit in the caches. The `put` call atomically writes *parsed_reviews* object to memory server using hardware transaction between the PUT controller and OPC. C1 caches both *raw_data.csv* and *parsed_reviews* objects and accordingly, the OTC tracks C1 as a sharer of these objects.

Next, the *sentiment_analysis* function, scheduled on C2, similarly performs a `get` on *parsed_reviews*. On access the object is brought into the LLC, making C2 a sharer for the object. After computation, a `put` call writes a new version of *parsed_reviews* to the memory server. The Āpta

[5]Originally an object in [13] referred to a memory mapped file which maps a range of data to a range of physical addresses. This works very well for our purposes since FaaS objects are also allocated contiguously within a range.

[6]For object mapping, a combination of space efficient radix tree and lookup time efficient trie is used (inspired from page table in virtual memory and longest prefix match in routers, respectively). The radix tree traversal first points to 4KB/2MB page. Within the page, objects are organized as a trie. This organization ensures the data structure can be read in hardware controllers.

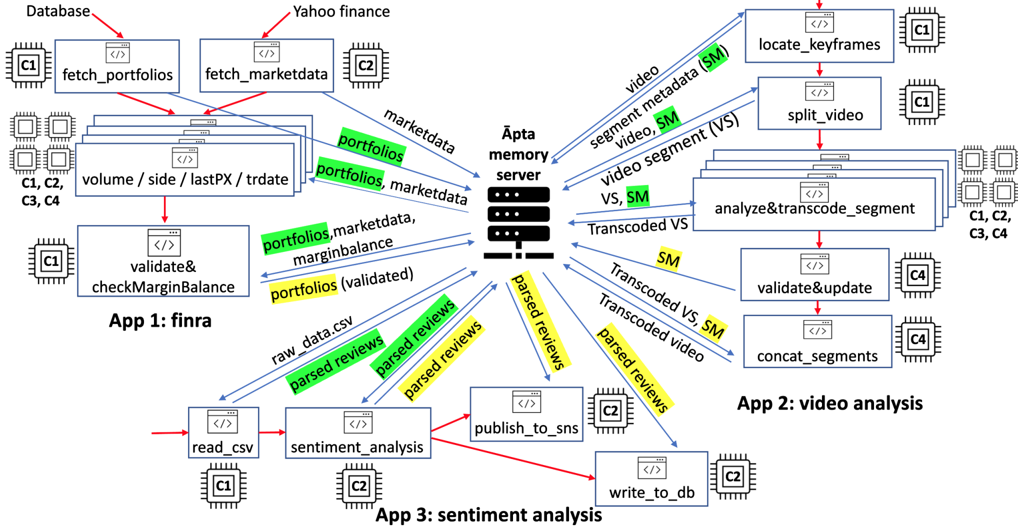

Fig. 7. FaaS applications annotated with object store interactions and scheduling decisions; highlight color changes indicate object writes requiring invalidation

protocol acknowledges the write from C2 immediately and sends invalidation to C1 asynchronously, tracking C1 as having outstanding invalidation-acknowledgments. When scheduling the next set of parallel functions, the EM checks with OIC and does not schedule the functions on C1 to satisfy *§Scheduling Correctness Criterion*. Scheduling on C2 provides opportunity to exploit locality as it previously executed a predecessor function. Accordingly, *publish_to_sns* and *write_to_db* are scheduled on C2 and both functions benefit from cache hits for accesses to *parsed_reviews*.

IV. Evaluating Āpta

A. Evaluation Goals:

(i) Compare performance of Āpta against the following *state-of-the-art* compute server fault-tolerant systems:

- RDMA-based object store with Faa$t caching [77]: An immutable object caching protocol run using two-sided RDMA verbs over Infiniband. On a `put`, write-through to object store with no sharer invalidation. On a `get`, if cache hit, incur one round trip to object store to ensure cached data is not stale (no object data transferred unless data is stale); if cache miss, read object from remote object store.
- *faster* RDMA-based object store with Faa$t caching: Uses the same interconnect as Āpta for RDMA, along with the above Faa$t software-based object caching protocol. This configuration allows us to isolate the benefits of improving just the underlying interconnect (transport layer).
- *faster* RDMA-based object store with Āpta caching: Uses the same interconnect and Āpta's object caching protocol but in software. This configuration allows us to quantify the benefit of our optimized coherence protocol.
- CXL uncached DM: The cache line granularity DM of CXL, that achieves fault-tolerance by disabling caching of any DM data (requires no coherence protocol). This configuration allows us to quantify performance benefits of caching in DM and object semantic operations proposed for Āpta.

TABLE II
Configuration of the Simulated System

3 Compute Servers	
Processor	single socket, 3.0 GHz; Int/FP Ops: 0.02 CPI
L1 I/D Cache	256KB, 8-way, private per core, 1 cycle
L2 Cache (LLC)	32MB, 16-way, shared, 10 cycles, 128 MSHRs
Local Directory	embedded in L2, fine sharing vector (cores, cachelines)
Local Memory	2 × 8GB DDR4-2400 MHz, 1 channel
1 shared Disaggregated Memory Server	
Directory (OTC)	20-cycle, coarse sharing vector (compute servers, objects)
Memory	2 × 8GB DDR4-2400, 1 channel
Interconnect	
Latency & Bandwidth	point-to-point, 500ns, 80Gbps full-duplex

(ii) Demonstrate the fault-tolerance and resilience of Āpta
(iii) Break-down performance gains for get and put operations (compute time is kept constant for all configurations)
(iv) Evaluate robustness of performance gains with respect to varied interconnect properties and compute server capabilities.

B. Evaluation Methodology:

Our evaluation of Āpta is driven by a simulator based methodology (similar to DM proposals [8], [44], [45]). We now set out the configuration parameters and workloads used in the simulation of such a system.

≻ *Workloads:* We use 6 full FaaS application workflows, totaling 26 functions, from different domains seen in FaaS - text, numeric, image, video processing. We simulate these full FaaS application workflows from start to finish to demonstrate realistic cache hit rate, invalidations and scheduling decisions. For each application, Table III shows the communication patterns in the workflow, input data size, constituent functions and a chosen instance of a schedule for an invocation. These applications cover the full range of characterized input dataset/object sizes and function communication and invocation patterns [62], [66], [87]. The applications use local DRAM main memory to store intermediate data, akin to a scratchpad. Table III shows this measured local memory usage

TABLE III

FAAS APPLICATIONS ANNOTATED WITH SCHEDULE

Application (Patterns; Input data size; Max RSS)	Functions (compute server c1-c3)
PHI data [46] (Broadcast, Pipeline; 20KB; 100MB)	identifyPHI (c1), deIdentify (c2), anonymize (c1), analytics (c1)
Sentiment Analysis [46] (Broadcast, Pipeline; 480KB; 93MB)	readcsv (c1), sentimentAnalysis (c2), publishSNS (c2), writeDB (c2)
FINRA [46] (Broadcast-Gather; 1.2MB; 23MB)	fetchMarket (c1), fetchPortfolios (c2), volume (c1), trdate (c2), lastpx (c3), side (c2), marginBalance (c1)
Video Transcode and Analysis [51] (Scatter-Gather, Pipeline; 2MB; 117MB)	locateKeyFrame (c1), splitVideo (c1), AnalyzeProcess (c1,c2), validate (c3), concat (c3)
Image Prediction [46] (Pipeline; 2.7MB; 357MB)	resize (c1), predict (c1), render (c1)
Serverless GEMM (sparse) [80] (Map-Reduce; 234KB; 943MB)	split (c1), mapper (c1, c2), split (c1), reducer (c1, c2)

excluding the input object (max resident set size). We report the geometric mean speedup as an aggregate statistic across all applications.

≻ *System configuration:* We model a DM system with four servers (3 compute servers and 1 memory server). Each compute server has a single socket CPU with local DRAM memory. The CPU has per-core L1 and a socket-shared L2 cache, kept coherent with a directory-based MOESI protocol. Within the memory server, we simulate DDR4 DRAM memory, along with the Āpta controllers. The compute servers connect to the DM server with ordered point-to-point links of a fixed latency and bandwidth (full system config in Table II).

The RDMA configurations are measured on same hardware as in Sec. II. To model futuristic, faster RDMA ($RDMA_f$ - running over the same PCIe gen5 interconnect as Āpta), we add the latency overheads incurred for using RDMA operations and software coherence protocol to Āpta's network latencies, as an approximation. Object get/put operations in a key-value store using RDMA to read/write data from/to remote memory are an average of 2.2× slower than CXL [29]. Above this, fault-tolerant coherent object get/put operations require employing key-value stores like Faa$t and Hermes [42], which use complex two-sided RDMA, adding even more latencies. We measured this as the latency difference between a write in Hermes and a one-sided RDMA write (≈14 μsec per call, fixed overhead irrespective of object size). We use these overhead latencies along with the respective coherence protocol actions to simulate $RDMA_f$+Faa$t and $RDMA_f$+Āpta.

≻ *Simulator setup:* We simulate the identical shared memory version of the full FaaS applications written in python, compiled down to C. We generate traces of these programs using the Prism framework [67], which uses Valgrind to generates traces of compute, memory, thread create/join and barrier events. The tool produces synchronization and dependency-aware, architecture-agnostic traces. These traces are manually annotated with FaaS phases of execution i.e., get/compute/put.

We replay the traces in a modified gem5 simulator [78]. We implement the proposed inter-server Āpta coherence protocol and its hardware controllers. OSC and OTC lookup incur latency of 20 cycles each, modeled on average address translation and directory lookup latencies in modern processors

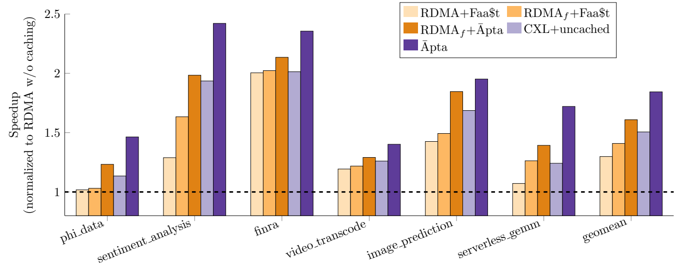

Fig. 8. Performance comparison of all configurations, normalized to RDMA without caching, for 6 full FaaS applications comprising of 26 FaaS functions

[33]. Memory ops are simulated with a detailed memory hierarchy. The replay mechanism uses FaaS phase of execution annotations to apply appropriate memory access characteristics for each phase of execution i.e., caches+local memory for compute, caches+DM for get/put. Integer and floating point ops are simulated with fixed CPI. We use an aggressive CPI and larger, lower latency L1/L2 caches to represent execution with optimized libraries (as in Sec. II-A). This simulator setup speeds up the computation phase of FaaS functions by 5× geomean compared to unoptimized (single thread) python functions run on a Intel i7-9700K machine. (Note this is conservative as Intel python extensions provide 200× speedup for scitkit-learn, 90× for pandas, 3× for tensorflow [37].)

C. Evaluation Results:

1) Performance benefit and analysis: Fig. 8 shows the performance of all configurations normalized to a baseline RDMA-based object store without caching.

Result 1: Āpta provides 42% geomean speedup over state-of-the-art RDMA+Faa$t. This performance gain comes from three sources: (a) improved network, (b) optimized Āpta coherence protocol, and (c) using hardware controllers for object access and coherence in DM. The $RDMA_f$+Faa$t configuration provides 7% performance improvements over RDMA+Faa$t, showing the performance gains from just the improved network. Next, employing Āpta's coherence protocol over $RDMA_f$ ($RDMA_f$+Āpta) provides further 12% improvement over the previous $RDMA_f$+Faa$t, showing the performance gains from our optimized coherence protocol. Finally, Āpta's use of DM hardware-controllers eliminates the irreducible software overheads of $RDMA_f$, thereby providing 18% higher performance than previous $RDMA_f$+Āpta.

Result 2: Āpta provides 24% geomean performance gain over CXL-uncached by using a fault-tolerant object caching protocol and object semantic reads/writes. Note that employing the CXL-uncached object store will perform worse than a faster RDMA-based object store with caching ($RDMA_f$+Āpta), emphasizing the need for Āpta's design in a DM system.

Result 3: We also evaluated the performance against the non fault-tolerant cached CXL DM. Āpta provides 10% geomean speedup over this CXL-cached system (not shown in graph). This shows that there is no performance cost to Āpta's fault tolerance; in fact, Āpta shows a small improvement in

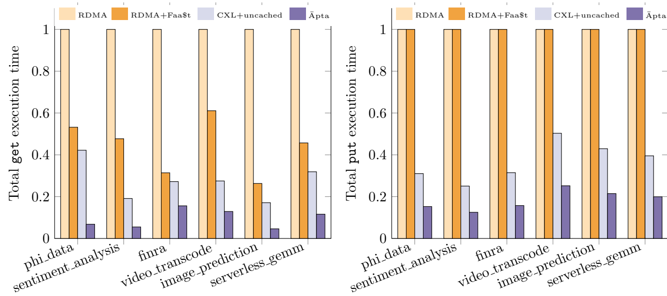

Fig. 9. Object `get` and `put` latencies, normalized to RMDA without caching

performance because it addresses CXL's inefficiencies owing to its cache line granular accesses.

2) Fault-tolerance validation: We verify the complete Āpta protocol (with additional states to handle races, if any applications misbehave), in the Murϕ model checker [15] and exhaustively verified for liveness (deadlock-freedom) and safety (linearizability). Importantly, we also model check to prove correct and non-blocking behavior in the presence of sharer compute server failures. The detailed state transition table and Murϕ model for the protocol are available online at https://github.com/adarshpatil/apta.

Result 4: Because the Āpta protocol does not wait for acknowledgments in the critical path, it has the potential for lower tail latencies. To measure this, we run the applications 50 times under variable network latencies to reflect real world rack scale networks [73] and measure the standard deviation of execution times. The network requests experience a random latency within a Gaussian distribution (40% variation around the mean as measured for an Infiniband network [41]). On average, applications exhibit 32% lower standard deviation of execution time with Āpta compared to the non fault tolerant CXL-cached system, demonstrating the resilience of Āpta.

3) Performance Break-down: For the simulated schedule, Table IV shows the number of `gets` which hit in the cache (compulsory cache miss for first `get` request on all compute servers, while subsequent `gets` may potentially hit in the cache) and the number of `puts` that require sharer invalidations (these `puts` jeopardize DM system availability and increase latency with the blocking CXL-cached protocol).

Result 5: Āpta lowers geomean get latency by 90%, compared to RDMA+Faa$t's 57% reduction and CXL-uncached 71% reduction over baseline. Fig. 9, left shows the total `get` latency, normalized to baseline for each application. Although, both caching mechanisms (RDMA+Faa$t and Āpta) see same cache hit rate, Āpta lowers geomean get latency by using an improved protocol and the DM interconnect.

Result 6: Āpta achieves the highest 81% reduction in geomean put latency, compared to 63% reduction for CXL-uncached. Fig. 9, right shows the baseline normalized total `put` latency. Since a `put` operation always writes through to the object store, RDMA and RDMA+Faa$t see the same `put` latencies. Āpta achieves the reduction by using optimized hardware

TABLE IV
`GET` AND `PUT` CHARACTERISTICS FOR FAAS APPS EXECUTION

App ↓ / Num. of →	`gets` (cache hit,miss)	`puts` (no inv, with inv)
PHI data	4 (2,2)	4 (4,0)
Sentiment Analysis	4 (2,2)	2 (1,1)
FINRA	5 (3,2)	3 (2,1)
Video Transcode	5 (2,3)	6 (5,1)
Image Prediction	3 (2,1)	3 (3,0)
Serverless GEMM	6 (3,3)	6 (6,0)

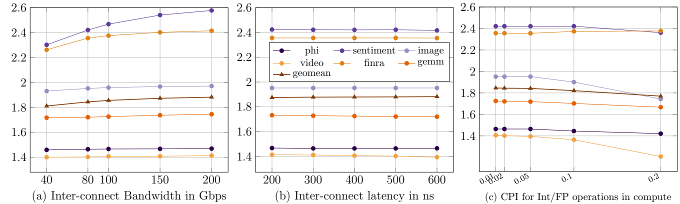

Fig. 10. Speedups of Āpta over baseline, for varied interconnect characteristics (a) bandwidth, (b) latency and (c) computation capability (INT/FP CPI)

transactions over an improved DM interconnect.

4) Sensitivity studies: Āpta performance gain is subject to the compute-to-communication ratio of the application. Therefore, we study the performance sensitivity due to variations in interconnect characteristics and computation capabilities.

Result 7: The performance of Āpta improves with increase in bandwidth of the DM network, seeing 90% geomean speedup over baseline for 200Gbps. Interconnect latency has a smaller impact on the performance of Āpta. Āpta still provides a 84% geomean speedup with high latencies of 600ns, as expected for CXL switched fabrics. Fig. 10 a & b summarizes the speedups of Āpta over baseline, for varied interconnect network latencies and bandwidth.

Result 8: Āpta's performance gain marginally reduces to 78% geomean with lower capability compute cores, as computation segment latency dominates in the total execution time. Fig. 10c shows the speedups obtained as we vary int/fp ops CPI of the core for both baseline and Āpta.

D. Evaluation Summary:

Āpta provides performance gains over all types of FaaS applications – from communication to compute heavy, applications with high object reuse and those with lower reuse, applications with serial and multiple parallel functions and over a range of object sizes.

V. Discussion

1) Specifics of CXL support for Āpta: Āpta's design introduces minimal changes to the CXL protocol and the servers. We discuss (i) the precise CXL protocol leveraged to design Āpta and (ii) changes needed to the CXL protocol specification to realize Āpta's fault-tolerance benefit. We refer to relevant sections in the CXL 3.0 specification [10] in the discussion.

Using CXL.mem protocol for pooled shared memory: CXL 3.0 specification defines the creation of a pooled memory device where multiple compute servers are configured to access a single memory region concurrently - called "shared FAM"

(fabric attached memory) (Sec. 2.4.3). Āpta designates the coherency model for the shared FAM as "hardware coherency with a directory" (Sec 2.4.4 [7]) and builds over the CXL.mem hardware coherence protocol (Sec 3.3). While CXL does not specify the detailed implementation of the directory, it does allow for tracking fewer states per cacheline i.e., 2 or 3 states instead of the original 4 state MESI protocol (Sec 3.3.3 Implementation Note). Accordingly, Āpta encodes the state of the cacheline using I and A stable states as per the CXL.mem parlance and implements the directory logic in the OTC. To send invalidations and receive invalidation-acknowledgments to/from compute server caches, OTC uses Back-Invalidation Snoop (BISnp) and Back Invalidation Response (BIRsp) messages (Sec 3.3.7, Sec 3.3.8), sent over dedicated S2M BISNP and M2S BIRSP channels (Sec 3.3.2).

Permitting asynchronous invalidation in CXL.mem protocol: The CXL 3.0 specification clearly defines the blocking behavior of BISnp requests (Sec 3.3.3). The synchronous invalidation behavior is further reinforced in the ordering rules (Sec. 3.4, Table 3-50 and Appendix C.1.2). Āpta requires the synchronous BISnp condition to be relaxed in the specification. This would allow implementations like Āpta to respond to write requests and other read requests immediately without waiting for the invalidation to complete.

2) Generality of proposed hardware: Designing controllers for CXL memory is currently under active development. CXL is being investigated to provide persistent memory [54], pooled remote memory to expand memory capacity [50], near-memory accelerators on CXL [34] and dynamic tiered memory [35]. Efficient implementation of these designs requires controllers for data persistence, address translation and data coherence. Āpta's controllers OPC, OSC and OTC basically provide the aforementioned services and can be adapted to suit these and other emerging applications.

3) FaaS scheduler deep-dive: Using Kubernetes as a case-study we illustrate how Āpta interfaces with a scheduler.

Generally, FaaS schedulers are complex frameworks that correctly and efficiently schedule functions on compute nodes. Notably, several custom built [22], [53], [56] and cloud provided [3], [27], [61] frameworks exist. The *kube-scheduler* component of Kubernetes is an example of such a scheduler. It considers several factors like individual and collective resource requirements, hardware/software policy constraints, affinity/anti-affinity specifications, inter-workload interference etc., when making scheduling decisions [47]. The scheduler has two cycles: a serial scheduling cycle and a parallel binding cycle, with each cycle consisting of multiple stages. Āpta can use existing stages in the serial scheduling cycle to achieve its objectives. Specifically, the pre-filter stage can query the OIC and remove invalidation pending compute servers from scheduling decisions (for correctness). The pre-score stage can add affinity labels to nodes with locality which can then be used in the score stage to preferentially select these nodes.

[7]CXL 3.0 permits the coherency model of the shared FAM to be either hardware coherency or software-managed coherency.

VI. Related Work

Resilient coherence protocols: A class of works [1], [19], [20] design coherence protocols that can tolerate dropped and faulty messages. They reissue requests on a timeout to recover, but crucially assume all participants are alive. Āpta is the first work to handle complete node failures.

FaaS applications: A number of works [11], [43], [57], [85], [86], [90] composed function benchmarks and software stacks employed in FaaS platforms. They also demonstrate several FaaS inefficiencies: data communication, cold start etc. Āpta addresses a chief inefficiency of FaaS – data transfer overheads and provides a fault-tolerant DM system for FaaS applications.

Reducing communication overheads in FaaS: Several works [55], [64], [77], [81], [83], [88] aim to provide software-based caches at compute servers to cache objects. These works reinforce the potential of caching to improve performance despite being connected by fast networks. Āpta provides software transparent object caching using CXL-based DM.

Faastlane [46], SAND [2] co-locate functions within an application (restricting scheduling) as threads/light-weight contexts to use local shared memory for low communication latency. Āpta allows function processes, to access shared objects from local caches if co-located, but critically also permits flexible scheduling, anywhere in the DM system.

High performance remote memory: Numerous works [12], [16], [24], [30], [68], [79] have used RDMA interconnects to provide software-based remote memory for applications. Āpta overcomes inefficiencies of RDMA by using DM, providing the highest performance remote memory. MIND [48] accelerates RDMA remote memories with in-network coherence and memory management. Analogously, Āpta designs a hardware coherence protocol for a DM object store but crucially enforces availability in the presence of compute server failures.

Disaggregated memory: This line of work use hardware-supported operations [52] to provide remote memory. Clio [31] defines explicit virtual memory API calls for processes on compute servers to allocate, read/write and synchronize accesses to the DM. COARSE [89] uses DM to accelerate distributed deep learning training. Kona [8] and DM prototypes [49], [74] create per-server private regions on the memory server to allow compute servers to transparently extend their memory capacity. While Āpta shares some common objectives, it builds on top of a CXL 3.0 DM system, specializes it for FaaS, and ensures availability in the face of server failures.

VII. Conclusion

In this paper, we have observed that upcoming CXL-based DM systems can alleviate the communication bottlenecks of cloud-based FaaS applications but lacks the necessary fault-tolerance to operate in a failure-prone datacenter. We have proposed Āpta, a CXL-based DM system for maintaining FaaS objects that provides efficient object-granular access and allows fault-tolerant caching of objects in compute servers caches, without compromising consistency. Thus, Āpta has showcased for the first time a fault-tolerant cloud use-case for CXL-based coherent disaggregated memory.

References

[1] K. Aisopos and L.-S. Peh, "A systematic methodology to develop resilient cache coherence protocols," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 47–58.

[2] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018.

[3] Amazon, "AWS Step Functions," https://aws.amazon.com/step-functions/.

[4] Amazon AWS, "A fully managed in-memory data store," https://aws.amazon.com/elasticache/.

[5] Amazon AWS, "Make a lambda function idempotent," https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/.

[6] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[7] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, "Using lightweight formal methods to validate a key-value storage node in amazon s3," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. Association for Computing Machinery, 2021.

[8] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2021.

[9] Compute Express Link, https://www.computeexpresslink.org/.

[10] Compute Express Link 3.0 Specification, https://bit.ly/cxl3-specification, August 2022.

[11] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. Association for Computing Machinery, 2021.

[12] A. Daglis, D. Ustiugov, S. Novaković, E. Bugnion, B. Falsafi, and B. Grot, "Sabres: Atomic object reads for in-memory rack-scale computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[13] Dave McCracken, IBM, "Object-based reverse mapping," https://landley.net/kdocs/ols/2004/ols2004v2-pages-71-74.pdf.

[14] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[15] D. L. Dill, "The murphi verification system," in *Proceedings of the 8th International Conference on Computer Aided Verification*, ser. CAV '96. Berlin, Heidelberg: Springer-Verlag, 1996.

[16] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014.

[17] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of cloudlab," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USENIX Association, 2019.

[18] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A review of serverless use cases and their characteristics," https://arxiv.org/pdf/2008.11110.pdf, 2021.

[19] R. Fernandez-Pascual, J. M. Garcia, M. E. Acacio, and J. Duato, "A low overhead fault tolerant coherence protocol for cmp architectures," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 157–168.

[20] R. Fernandez-Pascual, J. M. Garcia, M. E. Acacio, and J. Duato, "A fault-tolerant directory-based cache coherence protocol for cmp architectures," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 267–276.

[21] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018.

[22] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 475–488.

[23] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. Association for Computing Machinery, 2021.

[24] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan, "Scale-out ccnuma: Exploiting skew with strongly consistent caching," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. Association for Computing Machinery, 2018.

[25] Gen-Z consortium, http://genzconsortium.org.

[26] Gen-Z consortium, "ZMMU and Memory Interleave," https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-MMU-and-Memory-Interleave.pdf.

[27] Google, "Google Cloud Workflows," https://cloud.google.com/workflows.

[28] Google Cloud Functions, "Retrying event-driven functions," https://cloud.google.com/functions/docs/bestpractices/retries.

[29] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, High-Performance memory disaggregation with DirectCXL," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 287–294. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/gouk

[30] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017.

[31] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. Association for Computing Machinery, 2022.

[32] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358321

[33] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[34] W. Huangfu, K. T. Malladi, A. Chang, and Y. Xie, "Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 727–743.

[35] M. Inc., "Memverge unveils first software-defined cxl memory," https://memverge.com/memverge-unveils-first-software-defined-cxl-memory-applications-to-support-4th-gen-amd-epyc-processors/.

[36] Intel, "Intel Architecture ISA," https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[37] Intel, "Boost pandas, scikit-learn, and TensorFlow Performance," https://www.intel.com/content/www/us/en/developer/articles/technical/code-changes-boost-pandas-scikit-learn-tensorflow.html, 2021.

[38] Intel+Facebook, "Intel and Facebook Accelerate PyTorch Performance," https://community.intel.com/t5/Blogs/Tech-Innovation/Artificial-Intelligence-AI/Intel-and-Facebook-Accelerate-PyTorch-Performance-with-3rd-Gen/post/1335659.

[39] Intel+Google, "Intel collaborating with Google to optimize TensorFlow," https://www.intel.com/content/www/us/en/developer/articles/news/leverage-deep-learning-optimizations-tensorflow.html.

[40] S. Jha, V. Formicola, C. D. Martino, M. Dalton, W. T. Kramer, Z. Kalbarczyk, and R. K. Iyer, "Resiliency of hpc interconnects: A case study of interconnect failures and recovery in blue waters," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 915–930, Nov 2018.

[41] M. R. S. Katebzadeh, P. Costa, and B. Grot, "Evaluation of an infiniband switch: Choose latency or bandwidth, but not both," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[42] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, *Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol*. Association for Computing Machinery, 2020, p. 201–217.

[43] J. Kim and K. Lee, "Practical cloud workloads for serverless faas," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. Association for Computing Machinery, 2019.

[44] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. Association for Computing Machinery, 2019.

[45] V. R. Kommareddy, C. Hughes, S. D. Hammond, and A. Awad, "Deact: Architecture-aware virtual memory support for fabric attached memory systems," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[46] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *2020 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[47] Kubernetes, "Production-Grade Container Orchestration," https://kubernetes.io/.

[48] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. Association for Computing Machinery, 2021.

[49] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. Hill, M. Fontoura, and R. Bianchini, "First-generation memory disaggregation for cloud platforms," in *arxiv*, 2021.

[50] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, and I. Agarwal, "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*. Association for Computing Machinery, 2023.

[51] Q. Li and C. Kozyrakis, "Thousand island scanner (THIS): Scaling video analysis on AWS lambda," https://github.com/qianl15/this.

[52] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. Association for Computing Machinery, 2009.

[53] D. Liu, A. Levy, S. Noghabi, and S. Burckhardt, "Doing more with less:orchestrating serverless applications without an orchestrator," in *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'23. USA: USENIX Association, 2023.

[54] Y. Lu, J. Shu, Y. Chen, and T. Li, "Cache-coherent accelerators for persistent memory crash consistency," in *2022 Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2022.

[55] T. Lykhenko, R. Soares, and L. Rodrigues, "Faastcc: Efficient transactional causal consistency for serverless computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. Association for Computing Machinery, 2021, p. 159–171. [Online]. Available: https://doi.org/10.1145/3464298.3493392

[56] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "Wisefuse: Workload characterization and dag transformation for serverless workflows," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, jun 2022.

[57] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "Faasdom: A benchmark suite for serverless computing," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20. Association for Computing Machinery, 2020, p. 73–84. [Online]. Available: https://doi.org/10.1145/3401025.3401738

[58] Mellanox Technologies , "ConnectX-3 VPI Single and Dual QSFP+ Port Adapter Card User Manua," https://network.nvidia.com/pdf/user_manuals/ConnectX-3\%20VPI_Single_and_Dual_QSFP+_Port_Adapter_Card_User_Manual.pdf.

[59] Memcached, "an in-memory key-value store," https://memcached.org/.

[60] Memcached, "Memcached internals for end users," https://github.com/memcached/memcached/wiki/UserInternals#how-much-memory-will-an-item-use.

[61] Microsoft, "Temporal Platform," https://learn.microsoft.com/en-us/azure/azure-functions/durable/.

[62] Microsoft Azure, "Azure Functions Blob Access Trace 2020," https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsBlobDataset2020.md.

[63] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013.

[64] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, "Ofc: An opportunistic caching system for faas platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. Association for Computing Machinery, 2021.

[65] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[66] N. Nikitas, I. Konstantinou, V. Kalogeraki, and N. Koziris, "Cherry: A distributed task-aware shuffle service for serverless analytics," in *2021 IEEE International Conference on Big Data (Big Data)*, 2021, pp. 120–130.

[67] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead, "Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[68] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. Association for Computing Machinery, 2018.

[69] NVIDIA, "Mellanox bluefield dpu," https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[70] OpenCAPI consortium, http://opencapi.org.

[71] V. Pallipadi and S. Siddha, "Patting linux," *2008 Linux Symposium*, 2008.

[72] A. Patil, V. Nagarajan, R. Balasubramonian, and N. Oswald, "Dvé: Improving dram reliability and performance on-demand via coherent replication," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*. IEEE Press, 2021.

[73] A. Patke, H. Qiu, S. Jha, S. Venugopal, M. Gazzetti, C. Pinto, Z. Kalbarczyk, and R. Iyer, "Evaluating hardware memory disaggregation under delay and contention," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 1221–1227.

[74] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[75] Redis, "in-memory data structure store," https://redis.io/.

[76] Robert Blankenship, Intel Corporation, "Cxl 1.1 protocol extensions: Review of the cache and memory protocols in cxl," https://www.snia.org/educational-library/cxl-11-protocol-extensions-review-cache-and-memory-protocols-cxl-2020, 2020.

[77] F. Romero, G. I. Chaudhry, I. n. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2021.

[78] K. Sangaiah, M. Lui, R. Jagtap, S. Diestelhorst, S. Nilakantan, A. More, B. Taskin, and M. Hempstead, "Synchrotrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of cmp and hpc workloads," *ACM Trans. Archit. Code Optim.*, Mar. 2018.

[79] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *13th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, Oct. 2018.

[80] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. Association for Computing Machinery, 2020, p. 281–295. [Online]. Available: https://doi.org/10.1145/3419111.3421287

[81] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020.

[82] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. Association for Computing Machinery, 2020, p. 1093–1108. [Online]. Available: https://doi.org/10.1145/3373376.3378493

[83] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," in *Proceedings of VLDB Endowment*, vol. 13, no. 12. VLDB Endowment, jul 2020.

[84] UEFI Forum, "CXL: UEFI and ACPI specification enhancement," https://uefi.org/node/4093.

[85] D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing tail latency in serverless clouds with stellar," in *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021.

[86] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2021.

[87] vHive Ecosystem, "vSwarm - Serverless Benchmarking Suite," https://github.com/ease-lab/vSwarm.

[88] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020.

[89] Z. Wang, J. Sim, E. Lim, and J. Zhao, "Enabling efficient large-scale deep learning training with cache coherentdisaggregated memory systems," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[90] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. Association for Computing Machinery, 2020.