Global Distributed Client-side Cache for DAOS

Clarete R. Crasta
clarete.riana@hpe.com
Hewlett Packard Enterprise

John L. Byrne
john.l.byrne@hpe.com
Hewlett Packard Enterprise

Abhishek Dwaraki
abhishek.dwaraki@hpe.com
Hewlett Packard Enterprise

David Emberson
emberson@hpe.com
Hewlett Packard Enterprise

Harumi Kuno
harumi.kuno@hpe.com
Hewlett Packard Enterprise

Sekwon Lee
sekwon.lee@hpe.com
Hewlett Packard Enterprise

Ramya Ahobala Rao
ramya.ahobala@hpe.com
Hewlett Packard Enterprise

Shreyas Vinayaka Basri K S
shreyas-vinayaka.basri-k-s@hpe.com
Hewlett Packard Enterprise

Amitha C
amitha.c@hpe.com
Hewlett Packard Enterprise

Chinmay Ghosh
chinmay.ghosh@hpe.com
Hewlett Packard Enterprise

Rishi Kesh Kumar Rajak
rishikesh.rajak@hpe.com
Hewlett Packard Enterprise

Sriram Ravishankar
sriram.ravishankar@hpe.com
Hewlett Packard Enterprise

Porno Shome
shome@hpe.com
Hewlett Packard Enterprise

Lance Evans
lance.evans@hpe.com
Hewlett Packard Enterprise

Abstract

HPC/AI workloads process large amounts of data and perform complex operations on the data at exascale rates, for time-critical insights/results. Distributed workloads are often bottlenecked by communication when storage systems are used to co-ordinate and share results. Storage solutions supporting effective, scalable parallel access from compute clusters are critical to HPC architectures. Caching data on storage servers and/or clients are known techniques used by storage systems to ameliorate the communication costs. Current server-side caching methodologies are constrained by amount of memory and network bandwidth on the fixed and finite server nodes. Furthermore, most client-side caches are node-local, meaning the cached data is accessible solely by the node on which the data is stored.

Distributed Asynchronous Object Storage (DAOS)[8, 13, 14] is a high-performance exascale storage stack recently acquired by HPE. Global client-side caching for DAOS is an attractive proposition due to higher aggregate client-side resources (e.g., DRAM and network bandwidth) that can scale independent of the number of server nodes. In addition to providing faster data access, a client-side cache should also be efficient as it consumes expensive resources and requires an efficient caching framework with its associated policies. In this paper, we cover the details of realizing efficient shared client-side caching for DAOS.

CCS Concepts

• **Computing methodologies** → **Global client-side caching**; **File systems, Object Stores**; • **Information systems** → *DAOS*.

Keywords

Global, distributed, client-side caching, queryable, hpc, RDMA, DAOS

ACM Reference Format:
Clarete R. Crasta, John L. Byrne, Abhishek Dwaraki, David Emberson, Harumi Kuno, Sekwon Lee, Ramya Ahobala Rao, Shreyas Vinayaka Basri K S, Amitha C, Chinmay Ghosh, Rishi Kesh Kumar Rajak, Sriram Ravishankar, Porno Shome, and Lance Evans. 2025. Global Distributed Client-side Cache for DAOS. In *Proceedings of Cray Users Group Conference (CUG '25).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Problem Statement

Emerging HPC and AI workloads demand the ability to efficiently handle massive volumes of data at the petabyte scale. File system access costs impact the performance of these workloads significantly, especially distributed ones that rely on centralized storage to share results between processes running on different nodes. To address these costs, file systems employ caching mechanisms in faster storage. Some solutions provide server-side caching, while others extend capabilities to include client-side caching [2, 16, 30]. However, server-side caches are limited by the DRAM and network bandwidth available to the file-server nodes. Similarly, most client-side caches are node-local, meaning they store data only in the local SSDs or DRAM of the node where the file system client is running.

CUG '25, NYC, NY

ACM ISBN 978-1-4503-XXXX-X/2025/11
https://doi.org/XXXXXXX.XXXXXXX

Caches are typically designed to optimize specific use cases: application caches improve the performance of individual applications, file caches enhance file system access, and object caches accelerate object stores like Key-Value Stores. This siloed approach results in fragmented resource allocation across applications, file systems, and object stores, particularly in clusters running heterogeneous workloads with diverse storage and data access requirements. Additionally, most caches rely on a faster tier than the backing store but lack compatibility with multi-tiered memory architectures.

Advances in memory technologies such as HBM (High Bandwidth Memory)[9], SCM (Storage Class Memory)[15], NVMe (Non-Volatile Memory Express)[11], and interconnect technologies like CXL (Compute Express Link) [6] have introduced multiple tiers of memory and storage, each with distinct performance and cost trade-offs. GPU memory and GPU-based data access have further added to the complexity, creating yet another silo. As heterogeneous compute environments with accelerators and GPUs become the standard, traditional CPU-centric caching mechanisms are increasingly inadequate. Developing solutions tailored to specific compute, memory tiers, file systems and interconnect configurations often require frequent modifications as technologies evolve.

2 Approach

Efficient and advanced storage solutions are critical for enabling HPC and emerging AI workloads by providing the necessary infrastructure to store, manage, and access large datasets effectively. High-performance storage and scalability are key to ensuring that these target applications perform optimally and scale seamlessly. As the number of compute nodes in an HPC system grows, the aggregate resources (e.g., DRAM and network bandwidth) available through client-side caching also increase. This makes sharing client-side cache contents across nodes highly appealing, especially when combined with an efficient RDMA-based access framework, robust data placement algorithms, and effective caching policies. We are actively developing a shared client-side cache to front distributed storage systems. Our initial prototype focuses on DAOS [8, 13, 14], a next-generation HPC storage solution recently acquired by HPE.

Enhancing DAOS with a client-side global cache enables the cache to scale independently of the number of DAOS servers. This allows DAOS clients to leverage the memory and network bandwidth of compute nodes, reducing data access from central storage (DAOS servers) and minimizing communication and coordination overheads in distributed applications.

Additionally, our shared client-side global cache tackles challenges that are unique to modern HPC and AI workloads, including:

- Supporting parallel applications that rely on RDMA for efficient data movement.
- Managing caching and data transfers between CPU and GPU memory.
- Sharing cached data across parallel computing frameworks.

To address these challenges, we draw insights from other efforts in the field [19, 27, 30, 33], ensuring that our solution is robust, scalable, and capable of meeting the demands of heterogeneous and distributed computing environments.

Our goal is to allow applications to exploit memory and local SSDs across multiple nodes to cache and share data efficiently. This is architecturally illustrated in Figure 1 where both compute nodes and dedicated memory server nodes can contribute memory to the cache. The memory contributions can come from various sources, including DRAM, local NVMe SSDs, and GPU memory. Each memory type forms a distinct tier within the cache. A distributed cache manager consolidates per-node resources into a unified cache, maintains awareness of the capacities and access latencies associated with each tier, and manages multiple tiers of the cache by implementing cache policies and data placement algorithms. Data is cached locally to the nodes where there is a higher probability of it being accessed, whenever possible, to enable efficient access patterns. This cache can also be shared across applications running in a cluster. DAOS operations in the applications are intercepted and cached as required in the client-side cache.

3 Solution

Long-term persistent storage in our solution is provided by a high-performance, resilient, parallel distributed file system or object store such as DAOS, which offers scalability, high bandwidth, low latency, and high I/O operations per second (IOPS). DAOS is an open-source object store designed for next-generation workflows that integrate HPC, big data, and AI, enabling efficient data exchange and communication while optimizing performance and cost [13, 14].

Our solution augments DAOS with a globally distributed in-memory cache as shown in Figure 2. This cache is designed to improve application performance by leveraging DRAM and local SSDs to reduce latency and increase bandwidth, even when high-performance flash-based storage is available. Augmenting parallel file systems with a client-side global cache allows the cache to scale independent of the number of storage servers and will allow application clients to take advantage of the memory and network bandwidth associated with compute nodes. This wims at minimizing communication and coordination overheads in distributed applications.

The caching tier is designed to be transparent to applications, enabling unmodified applications using static frameworks like MPI[12] or SHMEM[21] to benefit from the cache simply by using the cache libraries to access the file system. The caching tier optimizes data movement using well-known caching strategies. Cache eviction policies can be controlled by the user, as can data prefetching and data location, enabling more effective management than hardware shared-memory caching algorithms. This is achieved by leveraging the rich metadata maintained by software.

Instead of treating all data as residing solely in storage and viewing data in compute nodes as being temporarily "owned" by the local node, we propose a holistic approach that considers data as existing at varying distances from any compute node within a tiered architecture. Data can reside in the DRAM of compute nodes, be copied into local storage attached to a specific node, or exist across multiple tiers of storage, such as SSDs or HDDs.

Rather than assuming data as statically located in one place, we adopt the perspective that data locality is dynamic. At any given time, based on usage patterns, data flows within the cluster to the access tier that best meets performance and availability requirements. Additionally, since the data locality can be queried, frameworks can

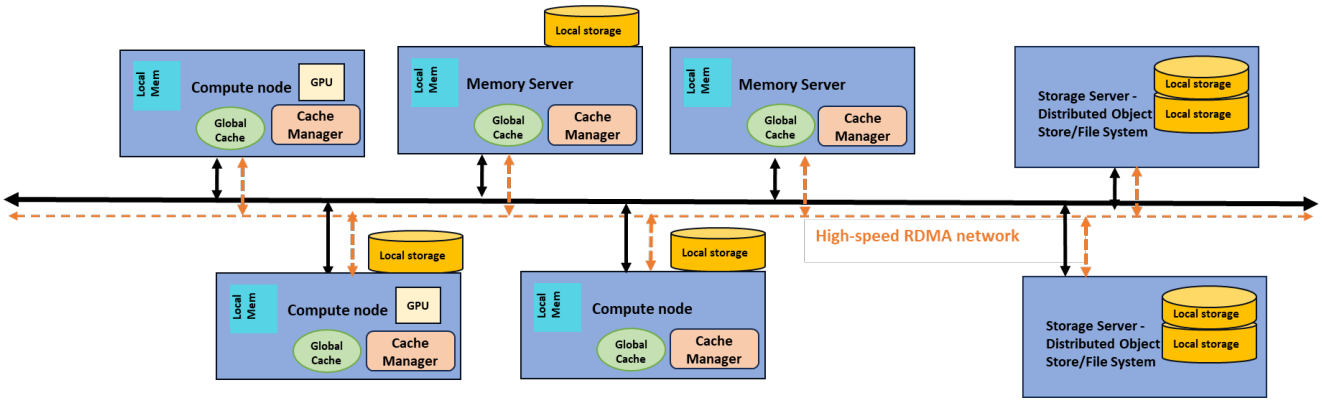


Figure 1: Overview of the caching solution

schedule compute tasks closer to the data, thereby reducing data access latencies and improving overall efficiency.

The global distributed cache (referred to as the cache in the document for brevity) is at the core of the architecture. The cache is designed to hold the working datasets of multiple concurrently executing applications in memory. Frameworks and applications can use an API to access and manipulate the content of the cache. The cache also includes space on local SSDs connected to compute nodes as a spillover area in case the cache runs out of DRAM space. SSDs offer a good balance between the high-speed, limited-capacity DRAM, and the larger-capacity parallel distributed persistent storage tier provided by DAOS or Lustre [1]. By taking advantage of the fast parallel read and write speeds of SSDs, frequently accessed data can be stored and retrieved more quickly, reducing latency and enhancing overall computational throughput, but at a significantly lower per-byte cost when compared to the more expensive DRAM.

The cache manager as shown in Figure 2 which is linked with the application plays a pivotal role in managing metadata across the distributed cache. By providing detailed data locality information, it enables frameworks, such as schedulers, to effectively align computation with data partitions. This capablity ensures data affinity, optimizing performance by reducing data movement and improving access efficiency in distributed environments. Additionally, the cache manager facilitates data movement between DRAM and SSDs based on user-provided hints or operator-defined policies, ensuring efficient utilization of resources and optimal performance for diverse workloads.

A portion compute nodes' DRAM, memory nodes' DRAM, and optionally other memory types such as HBM, is allocated to form a globally distributed cache. This cache transparently supports RDMA-based high-speed data movement within and across applications, enabling low-latency access. When DRAM capacity is exceeded, the cache seamlessly spills data to locally connected SSDs, reducing overall DRAM requirements while maintaining performance. Additionally, the cache manager dynamically relocates data within the caching layer to optimize proximity to computation, leveraging user-defined hints or operator-defined policies to ensure efficient and cost-effective operation.

4 Key Components of the Cache System

The caching system is built to deliver efficient, scalable, and consistent data access in distributed environments. At its core is the cache manager, which orchestrates cache operations, enforces policies, and manages metadata to ensure optimal performance. The system also incorporates an RDMA-based Data Access Framework, enabling high-speed, low-latency data movement across nodes, critical for distributed workloads. Furthermore, the caching system provides a comprehensive API and Configuration Interface, allowing seamless integration with applications, file systems, and object stores, while offering the flexibility to adapt to diverse workload requirements. The following sub-sections delve into the details of these components.

4.1 Cache Manager

The cache manager plays a central role in coordinating cache capacity and usage across different nodes and memory/storage tiers. It monitors cache usage, handles configuration changes, and dynamically adjusts cache capacity to optimize performance and resource utilization.

Multi-Tier Cache Management. A cache manager is present on every node and is responsible for managing multiple tiers of the cache. It enacts caching policies and data placement algorithms to ensure that data is cached locally to the nodes where it is accessed, whenever possible, for efficient access. By dynamically managing data placement, the cache manager optimizes resource utilization and minimizes data access latencies. The cache manager employs various mechanisms to access data across different storage and memory tiers:

- **CPU Memory:** It uses RDMA to access data remote memory and will leverage OpenFAM[24**?** –26] for efficient data transfers between nodes.
- **GPU Memory:** It uses RDMA to access data in GPU memory and will leverage GPUDirect[33] capabilities, when available, for efficient data transfers between GPU memory and the cache.
- **SSDs/Flash Storage:** The cache manager uses operating system or file system interfaces to access data stored in SSDs or flash storage.

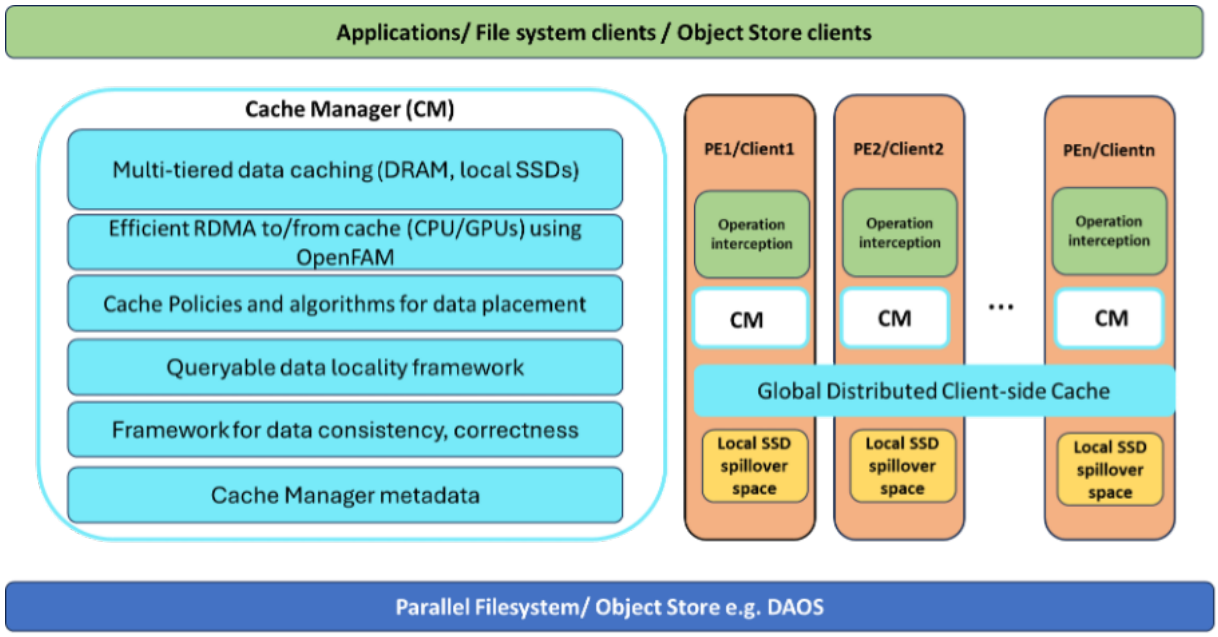


Figure 2: Components of the global cache

- **Distributed File Systems/Object Stores:** The cache manager integrates with distributed file system or object store APIs to access data stored in these systems.

The cache system implements advanced data migration strategies to move data between different memory and storage tiers. These strategies are based on access patterns and performance requirements, ensuring that frequently accessed data resides in faster memory tiers while less critical data is moved to slower, cost-effective storage.

Data Consistency and Correctness. The cache manager provides a robust framework for ensuring data consistency and correctness. This allows the cache to function according to the expected semantics of the underlying object store or file system. The cache manager enforces data consistency through mechanisms such as dirty bit tracking, cache invalidations, and metadata updates. The cache manager provides invalidation mechanisms that can be tailored to application requirements or the underlying storage system. The cache manager offers multiple configurable invalidation mechanisms to ensure data consistency:

- **Automatic Invalidation:** The cache manager will include the capability to automatically invalidate cached data when it receives notifications of changes in the underlying storage.
- **Time-to-Live (TTL) Policy:** Cached data can be invalidated based on a TTL policy, ensuring that stale data is periodically refreshed.

When configured as an application cache, all data accesses are routed through the cache manager, which is directly linked to the application. This setup allows for per-application access control and eliminates the need for invalidation, as the cache manager has complete control over data access. However, when configured as a file system cache, the underlying data may change due to external accesses outside the application or cache manager. In such cases, invalidation mechanisms are necessary to maintain data consistency. This ensures that the cache operates according to the expected semantics of the object store or file system, providing efficient, consistent, and reliable data access in heterogeneous and distributed environments.

Flexible and Configurable Eviction Policies The cache manager is designed to support a wide range of flexible and configurable eviction policies, ranging from basic approaches like LRU (Least Recently Used) and FIFO (First In First Out) to advanced strategies such as ARC (Adaptive Replacement Cache) [17]. Additionally, the system is built to accommodate the integration of new policies in the future. Applications can specify the desired caching policy by leveraging configuration parameters or API calls provided by the cache manager.

This adaptability enables the cache manager to cater to diverse application requirements and seamlessly integrate with various underlying storage systems. By offering configurable caching policies and invalidation mechanisms, the cache manager establishes a robust framework for optimizing performance while ensuring data consistency and integrity across heterogeneous environments. This flexibility is critical for maintaining efficient and reliable data access in dynamic and distributed computing scenarios.

4.2 RDMA-Based Framework and Protocol

A unified RDMA-based[20] framework and protocol is employed for data sharing, offering minimal overhead while being dynamic, scalable, and adaptable to heterogeneous environments. To enable HPC workloads to access remote memory, HPE has developed the OpenFAM API [3, 24, 25], which provides a programming interface

for building applications that leverage large-scale disaggregated memory. The OpenFAM API offers memory management and lightweight data operations, modelled after OpenSHMEM, and includes a reference implementation that supports both scale-up machines and scale-out clusters. Our cache integrates and adapts the OpenFAM software for use within the caching layer. OpenFAM memory servers, residing on compute nodes, facilitate access to memory within the cache. OpenFAM implements high-speed data movement over RDMA networks, including Ethernet, Slingshot, and InfiniBand, while managing memory on individual nodes. OpenFAM framework has a memory manager operates on each node, managing the memory contributions to the global cache. It supports essential operations such as memory mapping, allocation, and the release of cache segments, in response to the requests from the client which is the cache manager. The cache extends OpenFAM by overlaying advanced caching algorithms to optimize performance and enable efficient data sharing in heterogeneous environments comprising both CPUs and GPUs.

4.3 API and Configuration

The cache manager provides a comprehensive set of APIs to manage a distributed caching system. These APIs enable efficient data access, caching, and management across multiple nodes and memory/storage tiers. Below are some of the examples APIs provided by the cache manager:

Object Management

- *cm_open_object*: Opens an object in the cache, retrieves its metadata, and initializes an object handle for subsequent operations.
- *cm_close_object*: Closes an object, releases associated resources, and updates the cache state.
- *cm_delete_object*: Deletes an object from the cache and persistent storage, ensuring all associated metadata and resources are cleaned up.

Data Access

- *cm_read*: Reads data from the cache or underlying storage into a user-provided buffer. It dynamically fetches data from the appropriate tier (e.g., DRAM, SSD, or persistent storage) based on availability and access patterns.
- *cm_write*: Writes data to the cache, updating the appropriate memory tier and metadata. It ensures data consistency and handles segment-level updates.
- *cm_preserve*: Flushes dirty data from the cache to persistent storage, ensuring data durability and consistency.

Metadata and Locality

- *cm_get_cache_config*: Retrieves the cache configuration, including eviction policies, invalidation mechanisms, segment size and other parameters.
- *cm_get_locality*: Provides information about the location of data segments within the cache, enabling applications to optimize data access by co-locating computation with data.
- *cm_set_cache_config*: Sets the cache configuration parameters such as eviction policies, invalidation mechanisms, segment size and others.

Directory Management

- *cm_mkdir*: Creates a directory in the persistent storage layer.
- *cm_rmdir*: Removes a directory from the persistent storage layer.

Most of these APIs are accessible to frameworks utilizing the cache, such as a runtime system or a file system client component such as DAOS File System (DFS) in the case of DAOS, when the cache is employed as a client-side filesystem cache. However, certain APIs, such as those for configuring the cache (e.g., selecting caching policies), are explicitly designed to be user-visible. Overall, the cache manager APIs provide a robust framework for managing a distributed, multi-tiered cache, enabling efficient data access, consistency, and scalability in heterogeneous HPC and AI environments.

5 Evaluation

We present three distinct evaluations in the following subsections. First, we present prior evaluation results of the performance of OpenFAM RDMA operations[24–26]. Second, we present a preliminary evaluation of the cache's performance when used with the PageRank application.

5.1 OpenFAM RDMA Throughput and Latency Numbers

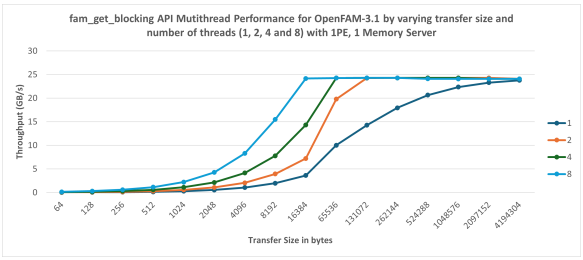
Since OpenFAM serves as the underlying substrate for data transfer and writing to the global cache, we begin by providing its current throughput and latency performance for basic get and put operations. All performance results presented in this section were obtained on a cluster that consists of compute nodes and memory nodes connected over HPE SlingShot interconnect with 25 GB/s link bandwidth. Both the compute and memory nodes have 2 sockets, each with 64 AMD EPYC 7763 cores. The compute nodes have 1 TB memory and the Fabric Attached Memory (FAM) nodes have 4 TB memory each. The nodes are configured with SLES 15 SP4 and OpenFAM 3.1.

For these measurements, a single Processing Element (PE) is configured on a compute node and a single memory server is configured on the memory node. Tests are conducted for both blocking and non-blocking get and put operations for different data sizes. The time taken per call is averaged over 10,000 iterations and used to calculate throughput. Figure 3(a) and Figure 3(b) show the throughput obtained for blocking get and put calls respectively as a function of message size for different numbers of threads.

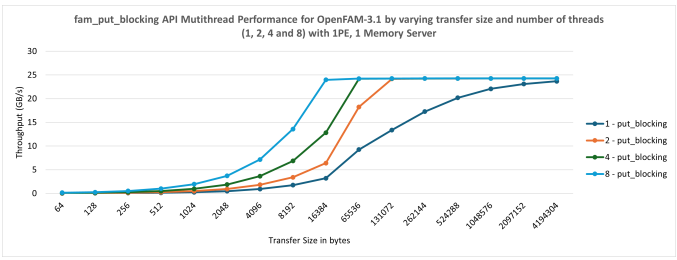
We observe that for blocking calls, the implementation can reach close to link bandwidth at 2 MiB message size with one thread, and at 16 KiB message size with 8 threads.

Figure 4(a) and Figure 4(b) show the throughput obtained with non-blocking get and put calls for different message sizes as the number of threads are varied between 1 to 8. As expected, non-blocking performance is better than blocking performance. The implementation can saturate the network at 16 KiB message size with a single thread, and can achieve close to link bandwidth with 4 KiB messages with 8 threads.

Finally, Figure 5 shows the round-trip latency for OpenFAM blocking get and put calls for short messages (8 bytes - 256 bytes) as the number memory servers is varied. In all cases, the latency is

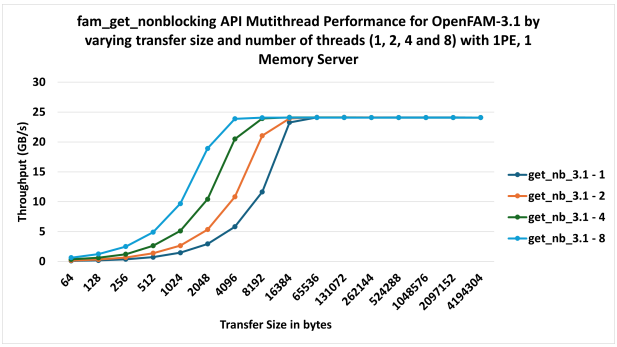


(a) fam get blocking

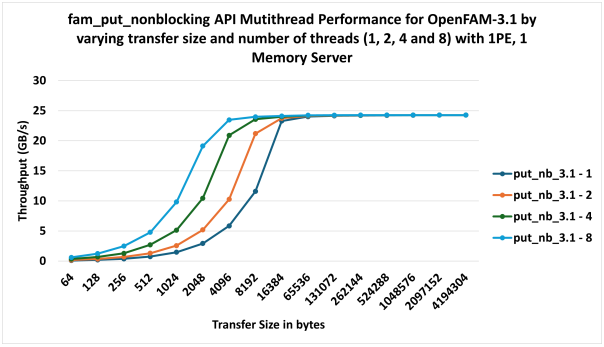


(b) fam put blocking

Figure 3: Performance of OpenFAM blocking operations



(a) fam get non-blocking



(b) fam put non-blocking

Figure 4: Performance of OpenFAM non-blocking operations

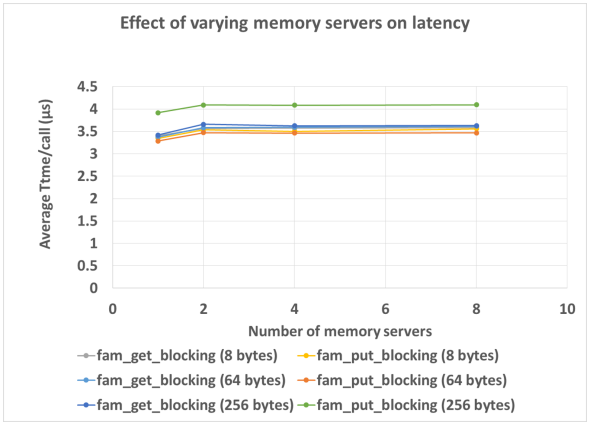


Figure 5: Average latency of OpenFAM operations

less than 5 microseconds, and is constant as the number of memory servers is varied.

From the perspective of the DAOS client, enabling the client-side cache allows DAOS operations to access cached data either locally or remotely in DRAM. This access involves an OpenFAM RDMA get/put operation, as detailed in the latency results above, along with some additional overhead from metadata and few other operations managed by the Cache Manager.

5.2 Preliminary Evaluation of the Cache with PageRank

To demonstrate the potential gains and feasibility of the cache, we have used PageRank[31] as an example application. We demonstrate the benefits of locality-aware scheduling and a client-side cache for the PageRank application. The baseline code used for the demonstration and comparison has no client cache, and the data (input matrix rows) is fetched from remote memory on every iteration of PageRank computation. In the locality aware scheduling version, we have the data read from remote memory on the very first iteration and cached in client buffers (worker buffers) and re-used on subsequent iterations. The locality aware version also schedules the tasks to the same worker that executed a task in the first iteration to take complete benefit of the client cache/buffers. Note that in this experiment the input matrix remains the same across iterations, but the vector changes on every iteration. The matrix size is 1M x 16M and the complete input matrix is pre-loaded to the Global Distributed Cache (FAM - used as proxy). The baseline version is compared with the locality-aware scheduling version for the time taken to complete a given number of iterations. Both versions are run with 16 workers, and 4 memory servers. Each worker runs on a node with two-socket AMD EPYC 7763 64-Core Processor and 1 TiB DRAM, which contributes to the client cache. The remote cache is emulated using 4 OpenFAM memory servers equipped with two-socket AMD EPYC 7763 64-Core Processor and 4 TiB DRAM.

The nodes are connected over the Slingshot interconnect and the remote cache is accessed from the worker through OpenFAM APIs [8], used as proxy for cache APIs.

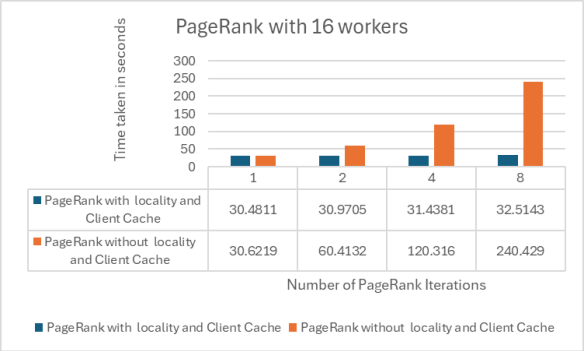


Figure 6: Demonstration of PageRank with locality-aware scheduling and client cache

As shown in Figure 6, the baseline version and the version optimized to take advantage of the client cache take about the same amount of time with the first iteration as both versions are reading data from the remote memory, but we begin to see advantage starting with 2 iterations and with 8 iterations the optimized version is about 7 times better than the baseline version with our preliminary tests. There is an opportunity to fine tune the locality-aware PageRank versions to incorporate variations to the amount data that can fit in the client cache.

6 Current Status

We are developing a prototype on a 52-node Slingshot-based HPC cluster, incorporating CPUs and extending the cache to support data access from GPUs. As part of this effort, we are building a runtime system that leverages the data locality information provided by the cache manager to optimally schedule workloads across the cluster. Currently, we have a functional model of the cache manager along with detailed design specifications. We are collaborating closely with the HPE DAOS team to enable a client-side global cache for DAOS and to evaluate the performance benefits of this addition. Standard DAOS benchmarks will be used to assess these performance improvements.

While our initial focus is on integrating the cache with DAOS, we envision that the cache could eventually support other file systems, such as Lustre, which is also deployed on our cluster.

7 Future Work

This cache is designed to be shared across applications running on nodes within the cluster. It can function as an application cache, allowing data processed by one application to remain in the cache for use by other applications, thereby improving efficiency and reducing redundant data movement. Additionally, the cache can be configured to operate as a file system or object store cache with minimal modifications. It is designed to be compatible with various file systems, such as the DAOS or the Lustre file system, enabling seamless integration into diverse storage environments. This flexibility allows the cache to serve as a unified resource, enhancing performance and scalability for applications running across the cluster. This cache can be shared across applications running in a cluster.

The next planned step involves implementing the required features to ensure the cache operates seamlessly across its various modes, adapting to the selected configuration. The cache is designed to be long-living and shared among multiple applications and workloads. The global cache must support dynamic increases and decreases in capacity. This is essential because the cache will be used by various applications with differing and dynamically changing compute, memory, and storage requirements. Also, nodes can be added or removed due to dynamic resource allocation [2] to existing applications or new applications on the cluster. The cache must adapt to these changes to avoid stranded resources and ensure efficient utilization. The ability to dynamically adjust global cache capacity is planned for future implementation. Some of the additional future evaluation items include enabling direct access to local cache content and implementing configurable invalidation mechanisms and efficient GPU integration and support.

8 Related Work

The general problem of shared client-side multi-level caches is well-understood within the context of distributed file systems [5, 7]. As early as 1992, Blaze noted that the key to scaling a distributed file system lies in its ability to protect the underlying file system from client activity, and that shared client-side caches are an effective mechanism for enabling scalability [5].

Modern HPC systems feature complex, potentially heterogeneous, memory and storage hierarchies, as well as an accompanying rich variety of data movement mechanisms. For example, a number of efforts consider the use of shared burst buffers or fast data tiers that leverage client-side resources to provide high-performance client-side storage to absorb transient ("bursty") I/O patterns [10, 29].

Client-side caching for file systems and object stores is a well-established technique in storage systems. For example, the Lustre file system supports client-side caching[22], where each client uses its own SSD as a local cache and can cache read-only data across multiple clients. Similarly, in-memory client-side caches, such as the Redis in-memory cache[23] and the NFS client-side cache [8, 9], are node-local and provide caching capabilities within individual nodes.

Efficient eviction policies are essential in caching systems, as static strategies like LRU, LFU, and FIFO often fail to accommodate the diverse access patterns of modern applications [17]. This mismatch can lead to sub-optimal performance due to excessive cache misses and inefficient resource usage. To overcome these limitations, Cachelib introduces a pluggable eviction framework that allows users to configure eviction policies based on workload characteristics [4]. This flexibility stands in contrast to traditional caching systems, which rely on fixed policies. However, Cachelib is primarily designed for local node caching, limiting its applicability for large-scale distributed environments.

Our cache, however, is distinct in several ways. It is shared, global, and multi-tiered, offering a unified caching layer that can be queried for data locality. This feature enables frameworks to build efficient scheduling algorithms by co-locating computation with data. More importantly, our shared cache is designed to allow direct access to the portion of the cache that resides locally on a node, enhancing performance and reducing data access latencies. Multilevel caching introduces well-known challenges, such as maintaining data consistency, managing data movement, and avoiding or handling duplicate copies of data. Our approach is informed by prior work on multilevel filesystem caching[2, 16, 18, 30, 32], which provides valuable insights into addressing these challenges.

Additionally, our shared client-side global cache is designed to tackle challenges unique to modern HPC and AI workloads. These include supporting parallel applications that rely on RDMA for efficient data movement, managing caching and data transfers between CPU and GPU memory, and enabling the sharing of cached data across parallel computing frameworks. To address these challenges, we draw on insights from other efforts in the field [19, 27, 28, 30, 33].

9 Summary

The proposed caching solution offers global multi-tiered caching, supporting a range of memory and storage technologies, including HBM, SCM, NVMe, and traditional DRAM. It dynamically allocates and manages cache capacity across these tiers, optimizing for performance and cost trade-offs [19]. The cache is also be queried for data location, enabling frameworks that consume the cache to co-locate compute and data effectively, thereby improving efficiency.

The solution is designed to support heterogeneous compute environments, ensuring compatibility with CPUs, GPUs, and accelerators. It provides efficient data access and caching mechanisms tailored for GPU memory and other specialized compute environments, addressing the unique requirements of these systems.

A unified cache management system is at the core of the architecture, with the cache manager that coordinates cache capacity and usage across different applications, file systems, and object stores. The cache manager dynamically adjusts cache capacity based on workload requirements and resource availability, ensuring optimal utilization of resources.

The solution leverages advanced high-speed interconnect technologies, such as Slingshot, to enable efficient data movement over RDMA and seamless sharing across different memory and storage tiers. This ensures low-latency access and high throughput, even in distributed environments. Finally, the architecture is highly scalable and flexible, designed to support large clusters with diverse workloads. It offers a configurable API, allowing customization and integration with various applications and storage systems, making it adaptable to a wide range of use cases and environments.

10 Acknowledgements

We would like to thank the DAOS team members at HPE for their valuable help and inputs. Special thanks to Pete Haddad, Eric Wu, and Binoy Arnold for their continuous support with the cluster, as well as other hardware and software requirements. Additionally, HPE's internal GitHub Copilot was utilized to generate BibTeX references for Overleaf, formatting and language revision.

References

[1] Lustre: A scalable, high-performance file system cluster, 2003. Accessed: 2025-01-24.
[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Distributed afs, 2025. Accessed: 2025-04-08.
[3] Syed Ismail Faizan Barmawer, Gautham Bhat Kumbla, Mashood Abdulla Kodavanji, Clarete Riana Crasta, Sharad Singhal, and Ramya Ahobala Rao. Client allocation of memory across memory servers, 2024. US Patent App. 17/815,366.
[4] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The cachelib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
[5] M. Blaze and R. Alonso. Dynamic hierarchical caching in large-scale distributed file systems. In *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 521–528, 1992.
[6] Compute Express Link Consortium. Compute express link (cxl), 2025. Accessed: 2025-04-08.
[7] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, pages 19–es, USA, 1994. USENIX Association. event-place: Monterey, California.
[8] daos stack. Daos storage stack, 2020. Accessed: 2020-08-27.
[9] Semiconductor Engineering. High-bandwidth memory, 2025. Accessed: 2025-04-08.
[10] Dave Henseler et al. Architecture and design of cray datawarp. In *Proceedings of the Cray User Group (CUG) Conference 2016*, 2016. Accessed: 2025-04-08.
[11] NVM Express. What is nvme technology?, 2025. Accessed: 2025-04-08.
[12] MPI Forum. Mpi forum documentation, 2025. Accessed: 2025-04-08.
[13] K. Harms. Daos is your future distributed asynchronous object store. Presented at the Argonne Leadership Computing Forum Hands-on HPC Workshop, October 2023. Accessed: 2024-09-30.
[14] M. Hennecke. Understanding daos storage performance scalability. In *Proceedings of the HPC Asia 2023 Workshops*, pages 1–14. ACM, 2023.
[15] IBM. Storage-class memory: The next storage system technology. *IBM Journals & Magazine | IEEE Xplore*, 2010. Accessed: 2025-04-08.
[16] Petros Koutoupis. The lustre distributed filesystem. *Linux J.*, 2011(210), October 2011. Article 3.
[17] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.
[18] Jeffrey C. Mogul. Recovery in spritely nfs. *Comput. Syst.*, 7(2), Spring 1994.
[19] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 561–577. USENIX Association, 2018.
[20] NVIDIA. Rdma architecture overview, 2025. Accessed: 2025-04-08.
[21] OpenSHMEM. Openshmem specification, 2025. Accessed: 2025-04-08.
[22] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, Dan Feng, Tim Süß, and André Brinkmann. Lpcc: Hierarchical persistent client caching for lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*, 2019.
[23] Redis. Client-side caching, 2025. Accessed: 2025-04-08.
[24] S. Singhal et al. Openfam: A library for programming disaggregated memory. In *OpenSHMEM and Related Technologies: OpenSHMEM in the Era of Extreme Heterogeneity*, volume 13694 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2022.
[25] S. Singhal et al. Openfam: Programming disaggregated memory. *Concurrency and Computation: Practice and Experience*, 35(5):e7291, 2023.
[26] Sharad Singhal, Clarete R. Crasta, et al. Openfam: A library for programming disaggregated memory. In *Proceedings of the Cray User Group (CUG) 2022*, 2022. Accessed: 2025-04-08.
[27] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1075–1092, 2024.
[28] H. Tang et al. Toward scalable and asynchronous object-centric data management for hpc. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 113–122, 2018.
[29] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An Ephemeral Burst-Buffer File System for Scientific Applications. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 807–818, November 2016. ISSN: 2167-4337.
[30] Weka. Weka architecture white paper, 2023. Accessed: 2025-04-08.
[31] Wikipedia. Pagerank, 2025. Accessed: 2025-04-08.

[32] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Trans. Comput. Syst.*, 29(2):Article 5, 51 pages, May 2011.

[33] Bo Zhang, Philip E. Davis, Nicolas Morales, Zhao Zhang, Keita Teranishi, and Manish Parashar. Optimizing data movement for gpu-based in-situ workflow using gpudirect rdma. In *Euro-Par 2023: Parallel Processing: 29th International Conference on Parallel and Distributed Computing*, pages 323–338. Springer-Verlag, Berlin, Heidelberg, 2023.

Received ; revised 23 March 2025; accepted