# 컨테이너 기반의 확장성 있는 Fabric-Attached Memory 관리 플랫폼

**이희락†, 이창규†, Awais Khan†, 강현구†, 마진석‡, 석성우‡, 오명훈‡, 김영재†**
**†서강대학교 컴퓨터공학과, ‡한국전자통신연구원**
**{heerock, changgyu, awais, hyeongu, youkim}@sogang.ac.kr, {majinsuk, swsok, mhoonoh}@etri.re.kr**

# Scalable Container-based Software Platform for Fabric-Attached Memory Pool

**Heerock Lee†, Chang-Gyu Lee†, Awais Khan†, Hyeongu Kang†, Jinseok Ma‡**
**Song-Woo Suk‡, Myeonghoon Oh‡, Youngjae Kim†**
**†Sogang University, Seoul, South Korea, ‡ETRI, Daejeon, South Korea**

**요 약**

Recent progress in Non-Volatile Memory (NVM) and Gen-Z Interconnect introduce a new type of memory pool called Fabric-Attached Memory(FAM). Until today, the massive memory pools are built via shared memory but often with severe overhead. Unlike existing solutions, FAM provides scalable memory pool with a near-local access latency. However, such hardware changes entail huge changes in software platform. To this end, we propose a scalable software platform based on the container which is widely deployed in many scalable systems. First, our platform provides extensible control plane to containers by employing broker container that accepts OpenFAM APIs. Second, we designed the data plane using bind mount to keep FAM's direct access with memory semantic. Our control plane showed only average 7.62% drops in IOPS and 8.13% latency increase compared to without container. Also there is only negligible overhead in our data plane.

## 1. INTRODUCTION

Advancements in Non-Volatile Memory (NVM) technology and high-performance interconnect such as Gen-Z [1] enable constructing a new type of storage class memory called Fabric-Attached Memory (FAM). Unlike existing shared memory architecture, which involves mediation between compute nodes to access remote memory, Gen-Z switch makes FAM directly accessible from all compute nodes through its memory-semantic protocol with high-speed fabric. Such properties enable FAM to play an essential role in the Memory-Centric Computing (MCC) [2, 3] which is designed on the idea that *all must fit into the memory*. Recently, HPE The Machine [4] provides one example of such MCC platform.

FAM interconnected through Gen-Z provides memory-intensive applications in manifolds. First, it offers a massive shared memory pool in which every address can be directly accessible from hundreds of nodes. Second, it can mitigate the performance gap between slow storage and memory with NVMs. At last, it provides direct memory `load/store` semantics on the memory objects through the Gen-Z interface. However, direct accessibility and shareability of FAM do not come for free. The memory management framework for FAM is inevitable to this end.

Designing a memory management framework for FAM should consider the following. First, it should provide applications with isolated memory space called a region and control over accesses to the region so that the data cannot be changed or shared unexpectedly. Second, memory management protocol should be independent of applications. Regardless of programming languages or user-level libraries, applications should be able to claim FAM. Last, it should offer direct access to the allocated memory space and keep simple memory access semantic. If accessing the FAM pool accompanies a significant software stack, applications cannot fully exploit the high-performance of the FAM.

A few prior works such as OpenFAM and MOSIQS [3, 5] proposed different programming models for FAM. For instance, OpenFAM offers API for disaggregated NVM pool to the applications. MOSIQS offers a memory shared object storage abstraction on persistent memory pool along with data object indexing. However, they require to rewrite the whole application in order to adopt FAM.

A simple solution is to deploy container on FAM pool. As in HPC, containers are often deployed to enable application isolation and to minimize host OS dependencies [6, 7, 8]. Further, containers provide an abstraction layer to the application with minimal performance degradation with its light-weight, virtualization, and portability properties.

However, there are some challenges to deploy container on MCC architecture equipped with FAM. We list the identified challenges here; First, the policy of provisioning the FAM resources into containers. Second, keeping memory-semantics and direct access of FAM inside of containers. Third, management protocol of FAM and its interface for containers. We elaborate above challenges further in Section 3.1

In this paper, we introduce a scalable container-based software platform for FAM pool. We choose a static resource provisioning policy so that allocation is done when a container is initiated and reclaim it as the container is terminated. Second, we bind mount FAM pool into containers so that they can use memory semantic through memory-mapped IO while providing isolation between other containers. We adopted state-of-the-art OpenFAM [5] programming model for FAM management. However, it only provides API specifications and not the architecture or implementation. Thus, we implement and deploy a broker container that manages the OpenFAM API requests. We use Docker containers and emulate FAM using Intel's Optane DC 3D-XPoint Persistent Memory. We prototype memory management protocol to i) (de)allocate the region in the FAM on application's request and ii) to manage region metadata in FAM. We used STREAM benchmark to evaluate the proposed approach [9]. The evaluation confirms that the container-based application deployment in MCC exhibits little overhead compared to direct application execution on MCC.

## 2. BACKGROUND

### 2.1 Gen-Z Fabric Interconnect

Gen-Z is a new type of fabric interconnect which allows any node attached to the fabric be able to access any memory address of nodes on fabric. Gen-Z also guarantees high bandwidth and near-local latency and provides memory semantic to nodes as they access their own local memory. With Gen-Z interconnect, the

이 논문은 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신 기획 평가원의 지원을 받아 수행된 연구임 (No.2018-0-00503, 메모리중심 차세대 컴퓨팅 시스템 구조 연구)

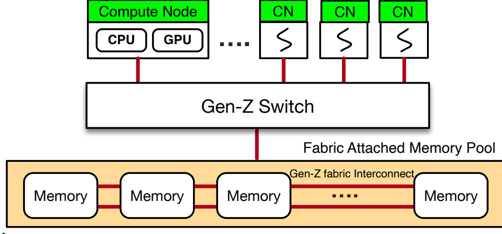


Figure 1: An overview of Fabric-attached memory via Gen-Z switch.

massive number of memory devices can form a single disaggregated memory pool. This is called FAM (Fabric Attached Memory). Figure 1 shows how the FAM is composed via Gen-Z fabric interconnect. Compute nodes such as CPU and GPU are connected to Gen-Z Switch. And memory modules are also connected to the switch not to the compute node. It allows any compute nodes to communicate with any addresses in the memory pool as if it accesses its own local memory using memory-semantic communication like `load/store`. Thus Gen-Z's communication protocol benefits memory-intensive applications in that it can eliminate intermediate interconnection between different compute nodes, which involves massive software stack overhead.

### 2.2 OpenFAM Programming Model

A scalable software stack should provide memory management frameworks for memory-intensive applications to exploit FAM's scalable characteristics. Recently, OpenFAM proposes a programming model for managing FAM with reference API implementation inspired after OpenSHMEM [2] specification. It specifies abundant features from (de)allocating FAM, reading and writing data, atomics to memory ordering. However, there was no prior work to clearly show how to deploy such a memory managing framework into a system with FAM.

### 2.3 Persistent Fabric-Attached Memory Pool

Recent advances in Non-Volatile Memory (NVM) take a seat as a new tier of storage with byte-addressability and high-density. NVM can potentially provide low-latency and high-bandwidth of I/O operations by many orders of magnitude by replacing disk-based storage with high capacity. Aggregated with Gen-Z fabric, NVM serves as FAM pool globally and directly accessible from any compute nodes [3].

### 2.4 Container

The containers are widely used in HPC and scientific computing for managing resources due to their light-weight virtualization and flexibility [10]. Similarly, in MCC, the container-based resource management can bring several benefits. For instance, it can provide flexible environment to deploy various applications that requires isolated regions in FAM pool. Additionally, containers incur less performance overhead compared to virtual machines to serve memory-intensive applications, as shown in [10].

## 3. DESIGN AND IMPLEMENTATION

### 3.1 Design Challenges

**Resource Provisioning:** There are two ways of provisioning FAM resources into containers. With a static provisioning policy, containers are allocated a specific FAM region's size when they are initiated. Unlike the static approach, in a dynamic provisioning policy, FAM regions are allocated to each container as much as containers request while they are running.

**Control Plane:** Since the FAM pool is directly accessible from each container and shared transparently between containers, the FAM pool's management protocol is needed. This management protocol can provide a strong abstraction for application and of-

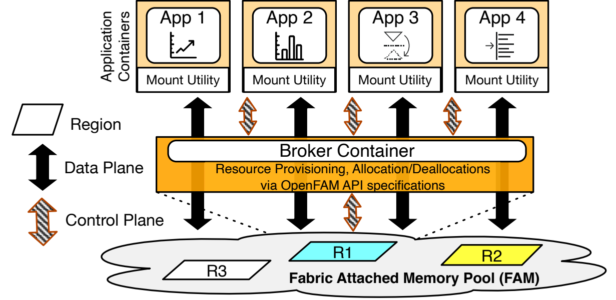


Figure 2: An overview of proposed software architecture.

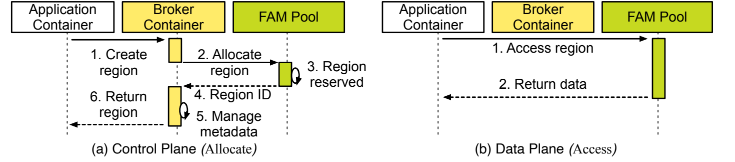


Figure 3: The control and data IO path.

fer consistency for data stored in the region that one container occupies. Otherwise, it entails serious synchronization overhead in that every application must communicate with each other to check the current memory pool status. Management protocol includes (de)allocation of FAM region and metadata for each region's status. Thus, it provides isolated region for each application running in each container.

**Data Plane:** To fully exploit performance of FAM, memory-semantics through Gen-Z fabric should be supported to applications running on each container. In order for that, there would be little software stack for applications to access the pre-allocated FAM region, and the container should provide a direct channel to the Gen-Z interface that finally goes to physical FAM address.

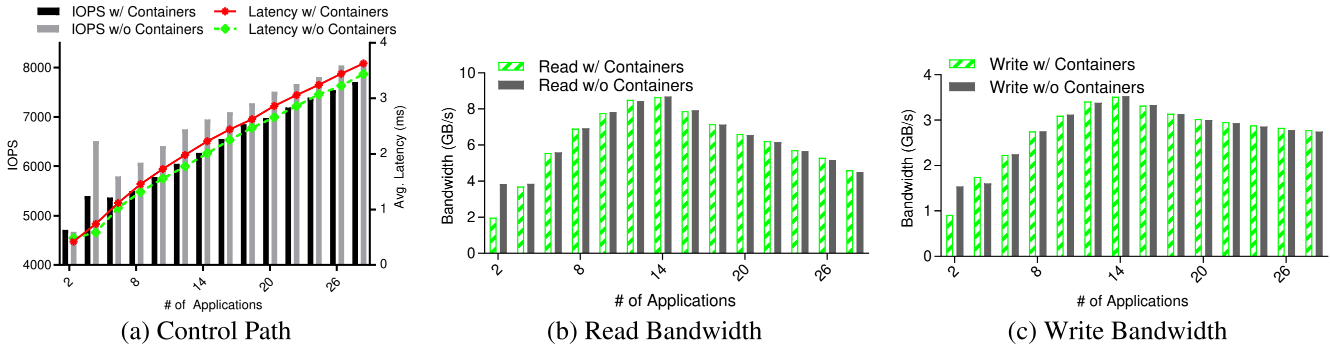
### 3.2 System Overview

Figure 2 illustrates the proposed system design atop a FAM interconnected via Gen-Z protocol. The top layer is the application containers layer, providing an execution environment for various scientific and memory-intensive applications. Each container provides the applications with direct channel to use Gen-Z interface which finally getting to FAM. The broker container provides access control to the FAM region and manages the metadata for each region and global FAM pool status. The metadata includes the free space in the FAM pool and each region's ownership. Our Design provides communication protocol for containers so that the application does not need to care about how the FAM is managed. This inter-container protocol provides a strong abstraction of FAM to applications and reduces the overheads accompanied by communication between many applications without container.

The bottom layer is the FAM pool, which is shared by multiple containers. Memory addresses in the FAM pool are divided by a region unit and allocated to an individual application container through the broker container.

### 3.3 Implementation

**Control Plane:** We implement the control protocol by referring to OpenFAM API. Communication is handled through gRPC and inter-container network by publishing a port to the broker container. The memory management functionality is performed through the memory server application running in the broker container. The memory server keeps information about the total size of the FAM and available space. It gives a unique region ID to each region and stores its ownership so that only one application container can work with an allocated region. In order to keep the static provisioning policy, every application requests a particular size of a




(a) Control Path (b) Read Bandwidth (c) Write Bandwidth

Figure 4: Performance evaluation of the proposed software platform for FAM pool.

region in advance. Figure 3(a) depicts how the control plane works with an example of region allocation. In the first place, application on an application container delivers `create_region()` request to the broker container before it starts actual work. Then the memory server in the broker container gets the request and checks whether there is available space for the requested size. If there is, it creates a region and grants the unique region ID for it, and maps client application ID to mark its ownership. Finally, it returns the region ID to the application that requested the region.

**Data Plane:** Our design aims to provide not only isolated FAM regions but also direct access to FAM. Figure 3(b) illustrates the data plane of our design. After the application container is granted to access the region via broker container, it can freely `load/store` to the region directly since broker container only grants exclusive access. Docker employs multiple layer of file systems to provide access control to the local host file system. To provide direct access to the FAM, our implementation depends on *bind-mount* functionality. *bind-mount* enables the write permission from inside of the container. In other words, application in the container can access to FAM as it does outside of container via *bind-mount*. In our evaluation, we used Intel Optane 3D-XPoint which is equipped as a local NVM since the Gen-Z is not commercially available.

## 4. EVALUATION

**Experimental Setup:** For evaluation, we used a server equipped with two Intel Xeon Platinum 8280M (total 56 cores) with 1.5 TB Intel Optane DC 3D-XPoint, and 768 GB DRAM. We used *numactl* to bind containers with NUMA node. The NVM is configured in interleaved mode. All experiments are performed using ext4-DAX and Linux Kenrel 5.4. To verify performance overhead of our container-based approach, we compare against applications directly running on the host system.

**Scalability Analysis on Control Plane:** Figure 4 (a) shows the communication performance between multiple application containers and the broker container. A single application executed in application container issues a million number of requests, i.e., `create` and `delete`, to the broker container. We vary the number of application containers to clearly articulate the broker container communication overhead in massively parallel requests. We bind each application container to one CPU core.

Figure 4(a) shows the total IOPS and average latency of varying application containers. The results show that IOPS and average latency gradually increase as the number of application containers increases. We also run every application without containers in the host system, i.e., grey bar and green line. With the result, we can figure out the performance degradation from the inter-container network overhead between the broker and application containers. We observed an average of 7.62% drop in IOPS and 8.13% increase in latency with our container-based approach.

**Scalability Analysis on Data Plane:** Figure 4(b) and (c) present the read and write bandwidth of the proposed approach. We evaluate memory bandwidth on STREAM benchmark using PMDK [9]. Each application claims a 15GB region in NVM for calculating three 5GB vectors. We increase the number of application containers with benchmark applications running on it to increase the transactions to memory simultaneously. We observed that bandwidth improves by varying containers, i.e., upto 14 containers. However, bandwidth degrades for both read and write I/Os. We attribute this performance degradation to contention in iMC and XPBuffer in Intel Optane DIMM as mentioned in [3]. Note that, there is a slight performance difference in both approaches in Figure 4(b) and (c), which we claim to be incurred by bind-mount docker utility.

## 5. CONCLUSION

We design and implement a scalable container-based software framework for FAM pool to fully benefit memory-intensive applications from MCC architecture. Specifically, we enable deployment of applications via containers on FAM pool and provide resource provisioning, memory allocations and sharing among multiple processes running inside containers. The evaluation confirms the feasibility of the proposed design.

## 참고 문헌

[1] Gen-Z Consortium. https://genzconsortium.org.
[2] S. W. Poole, A. R. Curtis, O. R. Hernandez, K. Feind, J. A. Kuehn, and G. M. Shipman, “Openshmem: Towards a unified rma model,” 2011.
[3] A. Khan, H. Sim, S. S. Vazhkudai, J. Ma, M.-H. Oh, and Y. Kim, “Persistent memory object storage and indexing for scientific computing,” in MCHPC, 2020.
[4] K. Keeton, “Memory-driven computing.,” in FAST, 2017.
[5] K. Keeton, S. Singhal, and M. Raymond, “The openfam api: A programming model for disaggregated persistent memory,” 2019.
[6] A. Torrez, T. Randles, and R. Priedhorsky, “Hpc container runtimes have minimal or no performance impact,” in CANOPIE-HPC, 2019.
[7] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in Euromicro, 2013.
[8] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, “Portable, high-performance containers for hpc,” arXiv preprint arXiv:1704.03383, 2017.
[9] “PMDK STREAM.” https://software.intel.com/content/www/us/en/develop/ documentation/vtune-cookbook/top/tuning-recipes/ pmdk-application-overhead.html.
[10] M. T. Chung, N. Quang-Hung, M. Nguyen, and N. Thoai, “Using docker in high performance computing applications,” in ICCE, 2016.