CHERRY: Dynamic Cache Management For Fast-Efficient Startup in PMM Based Cloud System

Sanghyeon, Eum *(UNIST)*, Hyunjoon, Jeong *(UNIST)*

***Abstract*—In few years, the paradigm of cloud service has been microservice architecture for serverless computing. To achieve it, their services has changed into container-based services, and this transition made it possible for providers to supply them in lighter environment. However, these services, especially Function as a Service (FaaS), involve restart overhead problem and most of its solution are focused on the warm-start problems. On the other hand, our project targets cold-start overhead problem of container-based service. To solve this problem, our project presents management system using persistent memory (PMM) and introduces two main techniques. First one is container layer caching operation using persistent memory, and second one is dynamic management operation following PMM usage. Our results show that the layer caching in PMM achieves higher performance in cold-start state than SSD and NVMe. Moreover, the system provides more flexible utilization of PMM by adopting dynamic transition of PMM capacity limitation.**

***Index Terms*—Docker container, persistent memory (PMM), image layer caching.**

I. Introduction

In recent year, a lot of cloud vendor has adopted microservice architecture to provide their services. This paradigm has evolved further, making it build fully or semi-fully serverless applications using FaaS. Most of cloud providers provide FaaS service. [12] FaaS is a way to implement serverless computing. It offers an user-friendly, event-driven interface for developing cloud based application. Unlike traditional cloud, user who use FaaS do not explictly provision or configure virtual machines(e.g. Docker[6], LXC[7]). FaaS users upload the source code of their functions to cloud platform. Functions start on triggering event, such as HTTP request. The provider have to provisioning the required resource, providing the high performance for functions and billing user for function executions. [12]

Cloud vendor want to provide high function performance to user, using the least resource. But container startup latency is the main reason for not meeting this goal. Function execution needs the code which is instructions (e.g. Docker image) of function in server's main memory. A case in which container data already exists in memory when a container starts is called a *warm start*. Conversely, starting through container data in persistent storage is called *cold start*. This process can take a long time relative to the function execution.[10] Warm start can satisfy the function's rapid startup time, but this is a big burden on the cloud vendor, especially if function executions are short and infrequent. For this reason, most cases of containerized applications use the fixed keep alive policy for reducing startup latency.[8][9]

This policy can be used if there is enough memory to cache the container data. Also, it is impossible to apply this caching policy if the system does not have enough memory. Enterprises are memcache lots of items for good user experience of their applications.[13] Therefore, it is important to use memory resources efficiently in service platform. In this project, we did not use memory caching for startup latency. instead of memory, we use *persistent memory* which is new storage media. Persistent memory is scarce resource and it does not support enough capacity to store whole container information. Therefore, we use persistent memory as cache manageed dynamically. Cache policy is based on overlapped container layer. Finally, we can reduce cold start latency which is inevitable.

II. Background

A. MicroService Architecture

Microservices are increasing their popularity in industry, being adopted by several big players such as Netflix, Spotify, Amazon and many others and several companies are now following the trend, migrating their systems to microservices. [16] Microservices are small autonomous services deployed independently, with a single and clearly defined purpose. Their independent deployability is advantageous for continuous delivery. They can scale independently from other services, and they can be deployed on the hardware that best suits their needs. Moreover, because of their size, they are easier to maintain and more fault tolerant since the failure of one service will not break the whole system, which could happen in a monolithic system.
Microservices have emerged as a variant of service-oriented architecture (SOA). Their aim is to structure software systems as sets of small services that are deployable on a different platform and running in their own process while communicating with each other through lightweight mechanisms without a need for centralized control.

B. Serverless Computing

Serverless Computing (as known as Function as a Service) is emerging as a new computing paradigm for the deployment of cloud applications, due to the recent shift of application architectures to container and microservice. Serverlesss differ from other cloud computing models in that cloud providers manage both cloud infrastructure and application scaling. Serverless applications are deployed in containers that automatically start on demand.

In the infrastructure-as-a-service (IaaS) cloud computing model, users pre-purchase capacity units. In other words, public cloud providers must pay for server components that are always up and running to run applications. It is also operator's responsibility to scale up server capacity when demand is high and scale down when it is no longer needed. The cloud infrastructure required to run the application remains active even when the application is not in use.

In contrast, serverless architectures only start when applications are needed. When an event triggers an application code for drive, the public cloud provider quickly allocates resources for that code. You will not be charged once the code is executed. In addition to the benefits of cost and efficiency, serverless ease the burden on developers in everyday, trivial tasks such as application scaling and server provisioning.

C. *Function as a Service*

Function-as-a-Service (FaaS) is an event-based computing execution model that runs on a stateless container that uses services to manage server-side logic and state. This allows developers to implement and manage application packages as a function without having to maintain their own infrastructure. FaaS is a way of implementing serverless computing, where developers write business logic and the platform runs it on a Linux container dedicated to management.

Serverless abstract and apply infrastructure issues such as server and resource allocation management or provisioning of developers to the platform, allowing developers to focus on code-writing and delivering business value. Functions are software elements that run business logic, and applications can consist of multiple functions. [17] [18] [19] [20] [21]

D. *Persistent Memory*

There are several types of PM based on different materials. Despite the different underlying materials, the common features offered by PM include (1) byte-addressability, (2) non-volatility and (3) performance in the range of DRAM's. They can be placed on the memory bus, thus appear to software as normal memory and can be accessed directly using load and store instructions.[15]

Intel's Optane DC Persistent Memory. The Intel® Optane™ DC Persistent Memory Module, which we term the Optane DC PMM for shorthand, is the first commercially available NVDIMM that creates a new tier between volatile DRAM and block-based storage. Compared to existing storage devices (including the related NVMe SSDs) that connect to an external interface such as PCIe, the Optane DC PMM has better performance and uses a byte-addressable memory interface. Compared to DRAM, it has higher density and persistence.

Operating modes. Optane DC PMM can be configured to run in two modes: Memory and App Direct mode. Both allow direct access to PM by normal load and store instructions and can leverage CPU caches for higher performance. Under the Memory mode,PMM acts as large memory without persistence; DRAM is used as a cache to hide the longer latency. The App Direct mode provides persistence. There is no DRAM cache in front to hide the high latency; the application should use carefully PM and handle persistence, recovery, concurrency and optimize for performance.[22]

In this project, we use PMM as caching storage. So, we operate in App Direct mode.

III. System Design

In this chapter, we will introduce about dynamic PMM cache management system, just called simply *CHERRY*. The *CHERRY* system consists of two module: *Cache manager* and *PMM monitor*. When the client wants to pull images from registry or run containers from existing images, *Cache manager* of CHERRY catches upcoming containers or images from Docker[6] API. Then, the manager checks whether layers, which are components of target images or containers, can be accepted in PMM. That is, *Cache manager* decides which layers can be cached or evicted from its own policy. To help this decision, additional information about PMM status should be required. As a result, Fig. 1. shows our system overview.

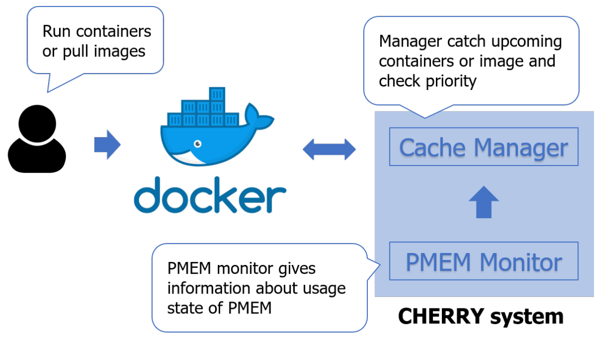
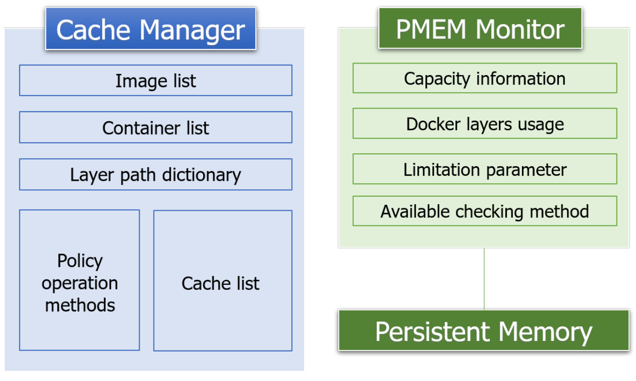

Run containers or pull images
Manager catch upcoming containers or image and check priority
docker
Cache Manager
PMEM Monitor
PMEM monitor gives information about usage state of PMEM
CHERRY system

Fig. 1: Overview of CHERRY system

A. *Cache Manager*

In *CHERRY* system, the main role of Cache manager module is to operate caching appropriate layer by its policy. Before applying it, the manager requires some status information for caching operation. Fig. 2(a) describes lists of variables in the manager. In case of Docker[6], its layers in images are tended to be encoded into SHA256 hash code. That is, when the user inspects images, the *RootFS* field does not provide real hash code for the layer. Since the layer directory is hidden, the system requires hash dictionary to decode path in faster way. Since *CHERRY* system depends on Linux command for inspecting images and containers, the lists and layer path dictionary can reduce the overhead of searching layers.

After inspecting layers, cache manager operates its policy. The manager contains policy operation methods, which are related with caching and evicting of layers. These operations are implemented by Linux commands and *inode* modification. Figure 3 shows example caching operation of *CHERRY*, and the access of Docker[6]. There exists image, which has three layers, and one of them has the highest priority. For this layer, the manager moves its data block, which actually contains data of layer, from NVMe to PMM. After data block was shifted safely, the manager copy its *inode*, and modifies its reference. When Docker[6] tries to use cached layer, it will



(a) Cache management module (b) PMM monitor module

Fig. 2: Components of modules

access original path of its target. In this case, modified *inode* would notify cached data blocks.

In case of evicting operation, Figure 4 shows the example its mechanism. When the layer L4, which has the highest priority, the manager read the state of PMM from monitor module. In this case, PMM is already used full resource with user-defined limitation 1GB, the manager determines eviction. The eviction policy follows that the lowest priority release first, so that L2 layer or L3 should be evicted. If there exist layers, which have same priority each other, the eviction order is up to firstly met layer. However, in Figure 4 case, regardless of its order, both L2 and L3 should be evicted finally. Since L4 has 400MB size, the manager have to evict additional layer to retain 400MB at least.

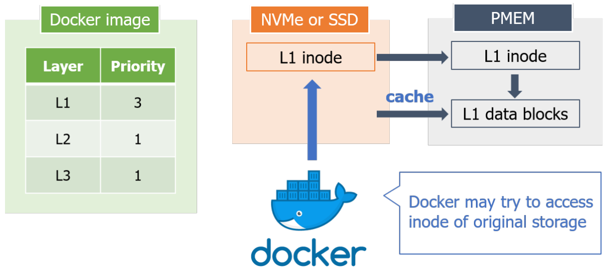


Fig. 3: Example of caching operation of CHERRY

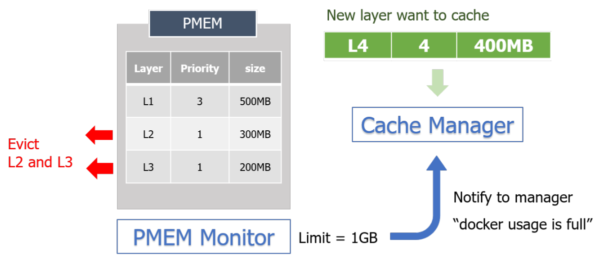


Fig. 4: Example of evicting operation of CHERRY

When the manager finishes caching or evicting operation, it tries to reorganize its cache list. Since finding cached layer from image lists or container lists requires quite large overhead for searching, *CHERRY* manages cached layers separately.

B. PMM Monitor

As forementioned, the role of *PMM monitor* supports the *Cache manager*. It has two main operations: monitoring status of mounted PMM and comparing the size of upcoming layers with its predefined limitation. The monitoring module also designed with two mode like *Cache manager*. Basically, daemon process mode of this module may only used for debugging with general status of PMM. As the status information, monitor module collects total capacity of mounted PMM, it current usage, availability and Docker[6] usage, that is, how much size is using in PMM. In case of daemon process mode, it shows only this four kinds of status as standard output. On the other hand, when the monitor module want to collaborate with caching module, the module needs to limit the usage of PMM for image layers. That is, this limitation parameter refers maximum proportion of layers in PMM, and default value is currently set to 10GB. As mentioned in motivation, PMM is generally scarce resource, so that this limitation parameter contributes more fine-grained usage of PMM. When the layer, which expects to cache in PMM, comes to cache manager, PMM module will get the size of layer and compare with current usage, Docker[6] usage and its limitation parameter. Moreover, the module decides the possibility of allocation in PMM and returns result to caching management module. Fig. 2(b) shows the information providing to *Cache manager* module.

C. Policy of CHERRY system

The policy of CHERRY system basically implemented from greedy algorithm. In case of images, the more sharing layers with other images, the higher priority will be obtained. Likewise, in case of container, the more invoked, layers in the container will get higher priority.

Algorithm 1 CHERRY Caching Manager

```
Input: initial images, PMM status
 1: Cache(common layers of initial images)
 2: for until program exited do
 3:     if Manager catch container execution then
 4:         Give priority +1 to related layers
 5:         targets = layers but not cached
 6:         if sizeof(targets) + PMM.docker > PMM.limit then
 7:             Do eviction(lowest priority layers in PMM)
 8:         end if
 9:         Cache(targets)
10:     end if
11: end for
```

The Algorithm 1 shows pseudo-code of our caching policy. When the manager starts initially, it immediately runs caching operation with images, which already pulled by Docker[6]. (line 1) In this case, the manager gives higher priority in order of the layer, which is sharing number with other images. Moreover, if there exists container executed log in past time, manager provides additional priority. This initialization process depends on Docker[6] process API, which provides trace operation of containers. After initialization step is done, the manager waits execution of containers or pulling images from user. When the manager detects the container execution or pulling images, it calculates the priorities of layers, which are related with newcomer. (line 3-4) The manager generates candidates, who are able to cache in PMM, with calculated priorities and updates manager dictionary and lists. Then, by caching policy, higher priority layers try to move their location with memory checking. This memory checking operation compares targeted layers with PMM limitation, which is user-define variable. If the manager expects overflow of memory, the manager executes eviction. (line 6-7) The target layer of eviction is lower layer than cache-desire layers, and this operation is executed iteratively until the memory size is available.

As a result, our layer caching policy is caused by greedy algorithm by using PMM available size. Although, this approach is not perfectly optimized for PMM utilization, our policy reserved most of size within limitation variable of PMM. Moreover, this approach contributed cold-start time of container, and we showed related result in next experiment section.

IV. Experimental Results

In this experiment, we implemented CHERRY by using Python and Docker[6] API. All performance measurements are taken from Shrimp2 server with Intel Optane Persistent Memory 128G and Samsung 970 PRO NVMe SSD. Our workload is modified Hello-bench with added caching thread. Hello-Bench[4] consist of variety workload.

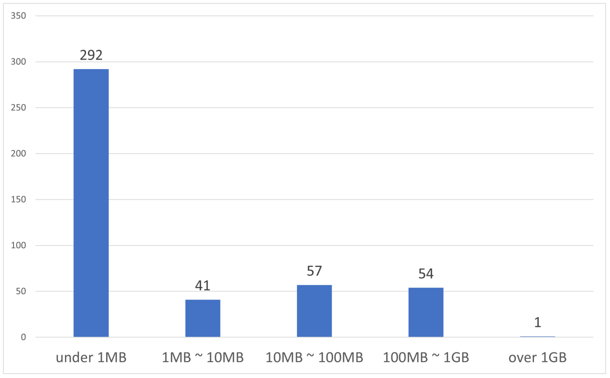


Fig. 5: The number of layer size classification in Hello-bench[4]

Figure 5 shows the number of layers, which are classified of its size. More than half of layers has under 1MB size and only one layer has more than 1GB size. We compared cold startup latency of each storage media and our system. After that, we show that how much time is shortened in specific container image. In this experiments, we can get new observation of image layer caching.

A. Cold-start time

The Figure 6 is a performance measurement for the case using only NVMe and PMM and cache size of our system using only NVMe and PMM. Contrary to the expected results for our experiments, it showed poor performance. The cause for this will be revealed later when analyzing a specific container image. On the other hand, Figure 7 is result of Ubuntu container. Basically, because PMM has a faster disk read speed than NVMe SSD, cold startup latency is halved.

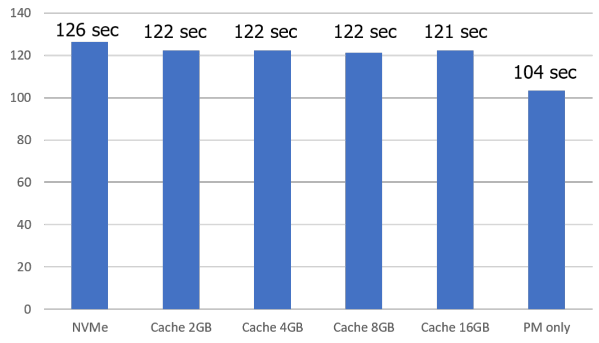


Fig. 6: Startup latency of all images in Hello-bench[4]

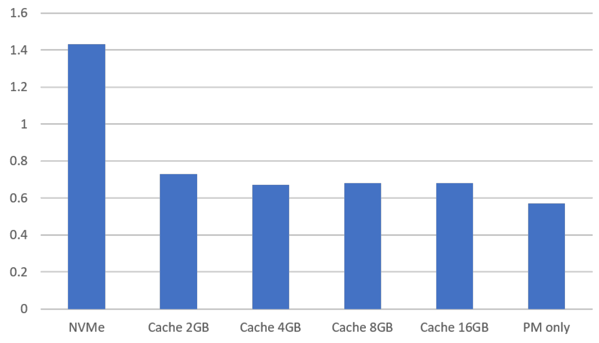


Fig. 7: Startup latency of Ubuntu container

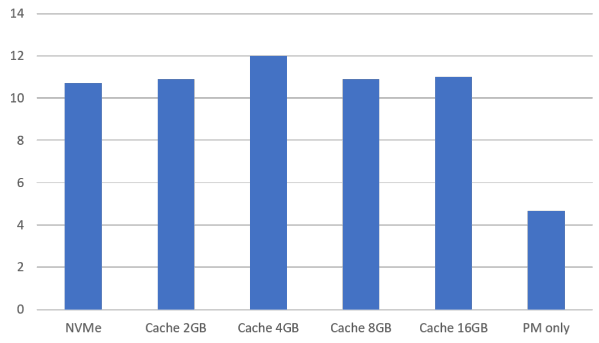


Fig. 8: Startup latency of MySQL container

The Figure 8 show the cold-start latency of MySQL container image. MySQL shows different results from the ubuntu container described above. Increasing the cached layer does not improve performance, but increases latency. Through this, we found that there is an overhead for the cache. Also, the reason that the performance was not improved even though the overlapping layers were cached is because the overlapping layers do not have a significant effect on startup latency.

The Figure 9 shows sibling images that share the same root layer as MySQL. According to the cache policy, our system

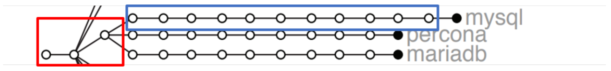


Fig. 9: Layer structure of MySQL container

caches the red area shared with other images in PMM, but it was found that the blue area has a direct effect on startup. Other sibling images showed the same pattern.

B. *PMM utilization*

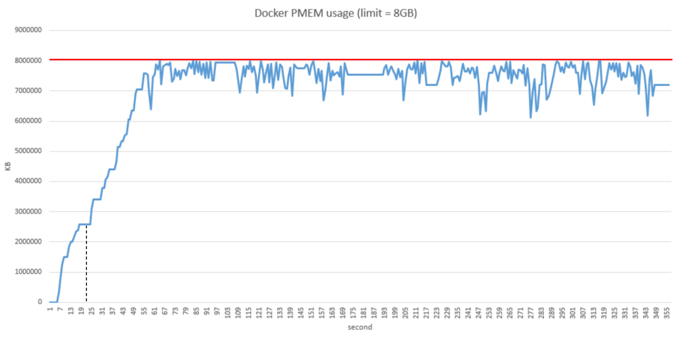


Fig. 10: PMM utilization with hello-bench[4]

The Figure 10 shows the utilization of CHERRY system in PMM. To trace utilization of PMM usage, we set the limitation variable as 10GB, and used hello-bench[4]. The red line means the limitation size of PMM, and we checked usage of Docker[6] layers over time. At first, CHERRY initialize its system up to black dotted line, and this spends 23 seconds, and memory usage was rising dramatically. After 60 seconds from system start, the system repeats caching and eviction, so that the memory usage fluctuates for every second. Although this policy does not completely optimize the utilization, the utilization does not below 6GB. That is, the system ensures at least 77 percent of full utilization.

V. Discussion

PMM based Layered FS There is a research about tiered file system based on PMM [24]. but Docker is dependent on union file system. In this project, we use ext4-DAX filesystem for PMM. Because Overlay[23] filesystem support ext4. However, like NOVA filesystem [25], we need new Filesystem which fully leverage PMM. A PMM-based filesystem that can interact directly with the Docker[6] daemon is expected to result in better performance.

Overhead for heavy images In Figure 3, CHERRY depends on Linux command for moving data blocks and *inode* modification. However, this method bounds caching overhead to storage and Linux command. In case of large size layer, we can observed larger caching time than other layers. If huge layer wants to cache in PMM frequently, CHERRY will spends much time on caching phase than other cases. Moreover, *inode* access can be bounded by storage. Figure 8 shows that mixed usage of PMM and NVMe does not always show better result. Although we assume that the these usage would result in intermediate latency between only PMM and only NVMe, it looks that both caching and eviction operation is bounded by slower storage. The reason why we choose this design is the Docker[**?**] does not support multiple storage for image layers. So, to solve this problem, we have to redesign its image storage structure.

Latency based layer caching As mentioned in the experiment, caching shared image layer is not efficient for reducing startup latency. So, we suggest the cache policy which determine priority based on startup time of layer. In this method, containers that take long time to execute can be start faster.

VI. Conclusion

This project proposes CHERRY to utilize PMM for Docker[6] layer caching. Based on greedy algorithm, we proposes caching policy, which provides higher priority to higher invoked and sharing layers in images. The experiment results with Hello-bench[4] show that the startup time depends on other properties of targeted layer rather than layer invoked time or sharing times. As future works, we propose to design using other insight, instead of sharing layer. Moreover, we face overhead from storage itself due to Docker[6]. Despite of these defects, we expect that cost of PMM will be lower in the future than DRAM, so that our caching system can be meaningful.

Acknowledgment

Thanks for SSL PMM expert Heejin Yoon.

References

[1] Tyler Harter, University of Wisconsin—Madison; Brandon Salmon and Rose Liu, Tintri; Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison, *Slacker: Fast Distribution with Lazy Docker Containers*, FAST'16.

[2] Ali Anwar, Virginia Tech; Mohamed Mohamed and Vasily Tarasov, IBM Research—Almaden; Michael Littley, Virginia Tech; Lukas Rupprecht, IBM Research—Almaden; Yue Cheng, George Mason University; Nannan Zhao, Virginia Tech; Dimitrios Skourtis, Amit S. Warke, and Heiko Ludwig, and Dean Hildebrand, IBM Research—Almaden; Ali R. Butt, Virginia Tech *Improving Docker Registry Design Based on Production Workload Analysis*, FAST'18.

[3] Alexander Fuerst, Prateek Sharma, Indiana University Bloomington USA; *FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching*, ASPLOS'21.

[4] Tyler Harter, University of Wisconsin—Madison, *HelloBench* https://github.com/Tintri/hello-bench

[5] Ao Wang and Jingyuan Zhang, George Mason University; Xiaolong Ma, University of Nevada, Reno; Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, and Vasily Tarasov, IBM Research–Almaden; Feng Yan, University of Nevada, Reno; Yue Cheng, George Mason University, *InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache*, FAST'20.

[6] *Docker*, https://www.docker.com/

[7] *LXC*, https://linuxcontainers.org/lxc/introduction/

[8] Mikhail Shilkov.*Cold Starts in AWS Lambda.* https://mikhail.io/serverless/coldstarts/aws/

[9] Mikhail Shilkov.*Cold Starts in Azure Functions.* https://mikhail.io/serverless/coldstarts/azure/

[10] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. *Peeking Behind the Curtains of Serverless Platforms.*, ATC'18.

[11] *Intel® Optane™ Persistent Memory* https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[12] Mohammad Shahrad and Rodrigo Fonseca and Inigo Goiri and Gohar Chaudhry and Paul Batum and Jason Cooke and Eduardo Laureano and Colby Tresness and Mark Russinovich and Ricardo Bianchini, Microsoft Azure and Microsoft Research, *Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider*, ATC'20.

[13] Mohammad Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li,Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung,Venkateshwaran Venkataramani, Facebook Inc, *Scaling Memcache at Facebook*, NSDI'13.

[14] Jian Yang, Juno Kim, and Morteza Hoseinzadeh, UC San Diego; Joseph Izraelevitz, University of Colorado, Boulder; Steve Swanson, UC San Diego *An Empirical Guide to the Behavior and Use of Scalable Persistent Memory*, FAST'20.

[15] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. *Evaluating persistent memory range indexes*, Proceedings of the VLDB Endowment, Volume 13, Issue 4, December 2019 pp 574–.

[16] Lewis, J. and Fowler, M. (2014). *MicroServices.www.martinfowler.com/articles/microservices.html*

[17] IBM Cloud Functions. *https://cloud.ibm.com/functions/*

[18] Amazon's AWS Lambda. *https://docs.aws.amazon.com/lambda/latest/dg/welcome.html*

[19] Google Cloud Functions. *https://cloud.google.com/functions*

[20] Microsoft Azure Functions. *https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview*

[21] Apache OpenWhisk *https://openwhisk.apache.org/*

[22] Intel Corporation. Intel Optane DC Persistent Memory Readies for Widespread Deployment. 2018. *https://newsroom.intel.com/news/intel-optane-dc-persistent-memory-readies-widespread-deployment/*

[23] Overlay Filesystem, Neil Brown *https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html*

[24] Shengan Zheng, Shanghai Jiao Tong University; Morteza Hoseinzadeh and Steven Swanson, University of California, San Diego *Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks*, FAST'19

[25] Jian Xu and Steven Swanson, University of California, San Diego *NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories*, FAST'16