Futuristic Computational Model for NVRAM Era

Arpit Gupta

Advisor: Debadatta Mishra

ABSTRACT

The introduction of Non-Volatile RAM (NVRAM) has forced the systems community to rethink about all the levels of system stack ranging from memory hierarchy to operating systems. The existing computational models for the conventional volatile RAM are not capable of exploiting the benefits provided by the NVRAMs. So we have tried to build an object based computational model which abstracts a given process to an object which can be handled by other processes to carry out the computation. The objects are created keeping in mind the persistent nature of NVRAMs, making them resistant (and ready to use) across power cycles. The object supports functionalities like attach, detach, calling a function, appending some code to the object, etc. The described model also finds usage in a variant of online computing model by providing object states to the users (as opposed to modern services like AWS Lambda), so that the users don't have to provide data again and again for different invocations of services. Also, we suggest a distributed framework where our model can be used for better utilization of resources.

1. INTRODUCTION

Modern day workloads includes data intensive applications which demand high memory capacity. Conventional DRAMs are no longer able to provide the capacity needed. This forced the research community to come up with a new memory device. Non-Volatile Random Access Memory (NVRAM) is the solution. These NVRAMs have become a topic of interest in the industry. Industries have started investing in this promising field. NVRAM as the name suggest provides persistent memory as opposed to the volatile behaviour of the traditional memory hardware structures like DRAM and SRAM. This requires the systems community to rethink about all the levels of system stack to exploit the benefits provided by the NVRAMs.

The research community indulged in persistent memory has been gaining momentum recently. Many experiments have been carried out to evaluate the performance of already existing operating system mechanisms in the presence of NVMs [1]. These experiments show that although not many operating sytems changes are required to support NVRAM, but the operating systems mechanisms needs to be changed to fully utilize the benefits provided by NVRAMs.

Even today we lack in commercial availability of NVRAMs. This encouraged people in research community to build NVM emulators providing similar access latencies as those expected from a real NVM. Also, the experiments have backed up researchers to come up with upgraded operating systems mechanisms like file systems, programming models etc. Attempts have been made to come up with whole new operating systems to make most use of hybrid memory resources.

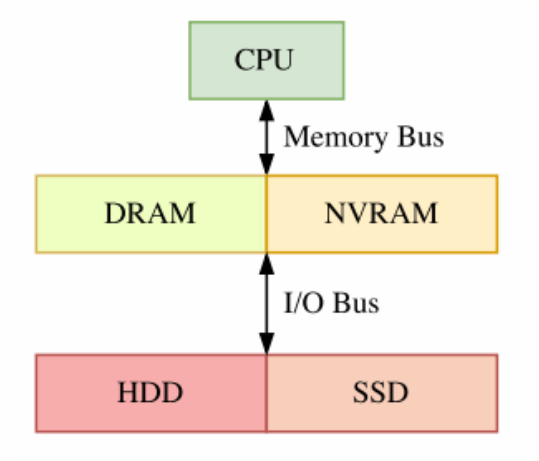

Figure 1: Modern hybrid memory hierarchy

Outline: The rest of the paper is structured in the following manner: In section 2, we present the background and related work on NVM. In section 3, we describe our computational model framework. In section 5 we discuss the utility of our framework in today's era. In section 4, we describe the implementation details of our framework. In section 6, we comment on the evaluation of our framework on some basic applications. In section 7, we can conclude the report and in section 8, we describe our future prospects related to the current framework.

2. RELATED WORK

Jalil et al. [2] surveys state-of-the-art work on integrating NVM into the memory hierarchy. Their work also talks about the hardware and software issues that needs to be tackled for successful integration of NVMs into the memory hierarchy. Many works have been proposed for successful implementation and utilization of NVMs in modern day architecture and operating systems. We summarize some of them below.

2.1 Emulation

Due to the lack of availability of commercially produced

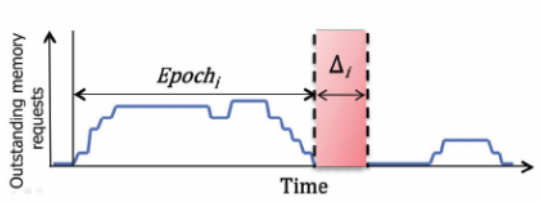

Figure 2: Emulating NVM latency by injecting software delay in Quartz [3]

NVMs many works have proposed emulators for NVMs and hybrid memory systems. We briefly describe two of them: Quartz and HME.

2.1.1 *Quartz [3]*

It is a lightweight performance emulation platform for non-volatile memory. It focuses on modelling persistent memory's primary performance features that contribute to the application performance, mainly bandwidth and latency. The emulation takes before this work were either too slow or too simplistic to capture the intricate characteristics of NVMs. Quartz enables emulation of a variety of NVM bandwidth and latency characteristics for performance evaluation of modern NVMs.

- **Bandwidth Model:** Quartz takes advantage of the DRAM thermal control feature which is provided by modern processors to limit the available bandwidth. Particularly, it uses the thermal control registers found in the integrated memory controller of modern processors. For e.g. Intel provides separate registers for read and write bandwidth throttling. This helps in modelling the asymmetric bandwidth provided by the NVMs.

- **Latency Model:** Since the hardware doesn't provide any means to modify memory latency, Quartz uses software techniques to emulate persistent memory. In general, software techniques can slow down the memory access significantly. So, Quartz focuses on modelling average application perceived latency to be close to persistent memory latency. They achieve this by dynamically injecting software created delays to account for the high persistent memory latency as shown in Figure 2. The delay injections are done at specific time intervals called *epochs*. The delay Δ_i is calculated as follows:

$$\Delta_i = M_i.(NVM_{lat} - DRAM_{lat})$$

where the notations used are as follows

  - Δ_i: software delay injected at the end of $Epoch_i$
  - M_i: total number of memory accesses to the main memory in $Epoch_i$
  - NVM_{lat}: average NVM latency
  - $DRAM_{lat}$: average $DRAM$ latency

2.1.2 *HME [4]*

While there had been many attempts to emulate persistent memory, HME was of the foremost efficient emulators for the hybrid memory systems. HME also provides a programming interface to evaluate the impact of hybrid memory systems on different applications. The implementation of HME is on NUMA based processors. The authors of HME also also redesigned the memory allocator to support memory allocation from DRAM and NVM regions. This is carried out by modifying Linux kernel to identify memory zones of NVM.

- **Bandwidth Model:** Just like other emulators HME makes use of DRAM thermal control interface provided by modern processors.

$$B_N = \varepsilon \times B_D$$

where B_N and B_D denote the bandwidth of NVM and DRAM respectively and ε denotes the user defined bandwidth throttling.

- **Latency Model:** Emulation of read and write latencies is done in a similar way to Qaurtz with a different delay computation. The read delay at the end of $Epoch_i$ is calculated as follows:

$$RD_i = \delta \times M_i \times (NVM_r - RDAM_r)$$

where the traditional delay is multiplied by a coefficient to account for the extra delay for LLC misses. LLC misses often accompanies a page table walk which takes a lot of cycles to access main memory. Here δ is correlated with the LLC miss rate of the application.

To emulate the write latency HME monitors the number of NVM write-back and write-through by employing the $PerformanceMonitorUnit$. The final expression for write delay is as follows:

$$WD_i = M_i \times \Delta_{wtl} + N_I \times \Delta_{wbl}$$

where M_i and N_i denote the number of write through and write back operations in NVM and, Δ_{wtl} and Δ_{wbl} denote the injected write-through and write-back delays respectively.

2.2 Programming Interface

With the incoming of some storage class memory technologies, there is a need for the programming interface between the hybrid memory and programmer. One attempt to design such an interface was by Mnemosyne. We describe it in brief below.

2.2.1 *Mnemosyne [5]*

It is a programming interface for programming with non-volatile memory. It mainly aims to provide an interface for creating and managing persistent memory. Mnemosyne also ensures consistency in the case of failures without which storage class memory might be left in invalid state, leading to crashing of the program the very next time it starts. It allows programmers to allocate persistent data dynamically. Along with this the design of Mnemosyne is compatible with the existing processors. The main goals of Mnemosyne are described as follows:

- **Persistent Regions:** It exposes the SCM to programmers through persistent memory region abstraction. This abstraction is a segment where the application programmers can read and write, and it survives application and system crashes.
- **Consistent Updates:** The primary mechanism Mnemosyne uses to to ensure consistency is by ordering writes to ensure that newer data exists before changing a pointer to reference the new data. Mnemosyne's updates mechanism relies on three hardware-primitives: *write-through stores*, *fences* and *flushes*.
- **Durable Memory Transactions:** Mnemosyne uses some advanced transactional memory techniques and provides a compiler to convert C code into transactions. While programming with Mnemosyne programmer can place the `atomic` keyword before a block of code to update the persistent data structures wherein the compiler produces code that passes memory references to a transactional system.
- **Persistent Memory Leaks:** Mnemosyne prevents memory leaks using two mechanism. Firstly, it necessitates programs to provide the memory with a persistent pointer to ensure that memory is not lost when a crash occurs. Secondly, by swapping data to files, Mnemosyne virtualizes persistent memory. This ensures that a leakage in one program affects only that program and does not reduce the availability of persistent memory for other programs.

2.3 File System Adaptations

With the programming interface and emulation techniques for NVM, already done it was time for the file systems interface to adapt to the non volatile memory. Many efforts have been made on improving the VFS to manage both persistence and memory capacity needs of the application.

2.3.1 Moore [6]

Moore propose a B-tree structure to map physical memory locations to logical addresses. This is particularly efficient in cases of random accesses. The details about the file system can be found in his work.

2.3.2 NOVA [7]

It is a file system to utilize the hybrid memory resources to its fullest besides providing consistency at the same time. NOVA adapts to the traditional file systems to exploit fast random access leveraged by the NVMs.

2.3.3 pVM [8]

It is a system software abstraction that provides the applications with automatic OS-level memory capacity scaling, flexible memory placement policies across NVM and fast object storage. It extends the operating system virtual memory and abstracts NVM as a NUMA node with support for NVM based memory placement mechanisms.

2.4 Operating System

With the adaptation of most of the OS mechanisms to NVRAMs, it was time for the research community to build an operating system itself that is optimized for NVM or hybrid memory model, exploiting the resources to the fullest. One such effort is Twizzler which we discuss below.

2.4.1 Twizzler [9]

Twizzler is an operating system designed for hybrid memory systems. It provides applications direct access to persistent storage. It also provides mechanisms for cross object pointers while removing itself form the common access path to persistent storage. It also provides fine grained security and recoverability. Some of the features of Twizzler are listed below.

- **Persistent Object Access:** Twizzler maintains a namespace of data objects. The object may contain code, data, or both. Each process in Twizzler takes a form of object. Due to the direct access ability and unique naming, cross pointers can be used in Twizzler.
- **Security:** Due to the interface of Twizzler (direct access to objects), it has a need of a richer security model. Twizzler provides security contexts in order to provide isolation and access control.
- **Persistent Kernel State:** The kernel of Twizzler uses persistent object states to determine the state of the kernel. thereby allowing kernel to easily recover after a power failure.
- **Hardware Support:** Although Twizzler can be implemented on existing hardware but extending the processors with new primitives will build room for improved performance. Twizzler propose to reintroduce segmentation and increase the virtual address space size. Some security features can be implemented easily with the help of additional structures like protection lookaside buffers.

Twizzler is implemented as a prototype inside FreeBSD. It is still in its infancy, meaning that it hasn't been implemented completely. Although their research paper talks about the framework, it doesn't talk about the implementation details. This work motivates us to build on similar terms, providing a new object based computation model for the hybrid memory systems.

3. OBJECT MODEL

In a existing framework, the lifetime of a process is limited by power cycles. With the incoming of NVRAM, we have the resources to make the process live across power failures. The existing idea of process cannot be realized directly in hybrid memory systems because of the absence of handle of process, i.e. we don't have a mechanism to resume a process (from undefined state) in existing OS mechanisms. So we propose an object model which can seamlessly be used across power cycles. Also consider a case where a programmer deals with dynamic data stored in a file. To process this data, the programmer has to map the file in memory every time the data changes. In our framework, objects are stateful, meaning they remember the data changes.

We propose an object model where we abstract out a binary to an object which can be handled by other processes to carry

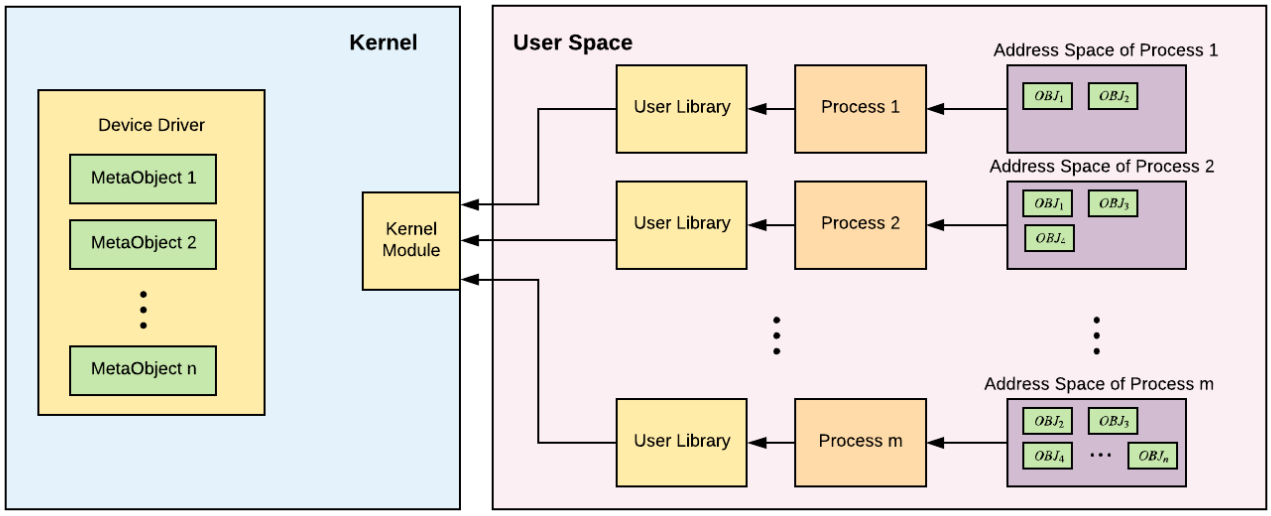

Figure 3: Object Model Framework

out computation using the object code and data. We provide a user library with the help of which user can make all the calls related to the framework as shown in Figure 3. The object has its own code and data segments mapped in the address space of the process (which attaches the object). A process can attach multiple objects to its address space and call functions from them. These objects are shared among all the processes, in the sense that any change to the data of an object by some process results in change of that data globally. This implies that any process attached to that object will observe the changes. The processes can also add functions (compiled code) to the already existing binary of the object, which can later be invoked by any process which attaches this object.

Interface

We provide a framework library which makes all the calls pertaining to the objects. The implementation details of these calls are mentioned in the next section. The framework interface with main functionalities is described below:

- `init_obj(object_binary)`: This allows a process to create a new object with the `object_binary`. The function returns an object id (`obj_id`) which can be used to attach or detach to this object by any process.
- `attach_obj(obj_id)`: This function attaches the process to the object that the process wants to handle. This is a prerequisite call for a process to carry out any computation using object or access data of the object. Basically, this maps the object code and data in the processes address space.
- `call_func(obj_id, func_name)`: This function calls `func_name` function in object `obj_id`.
- `append_code(obj_id, binary)`: This function a appends the given binary in already existing code of the object. This can be used to add new functions to the old object code.
- `detach_obj(obj_id)`: This detaches the object from the address space of the process barring the process to handle object in future. This also unmaps the object's data that was mapped during attach.
- `destroy_obj(obj_id)`: This function destroys the object if all the attached processes to this object have already been detached, otherwise returns error value. This function makes sure that no other process can attach to this object in future.

4. IMPLEMENTATION

The implementation of the framework includes a kernel module and a user library. The user library is responsible for making all the calls related to the object framework. The library makes use of ioctl calls to interact with the device driver. The library takes information about the object from the device and carries out the implementation itself in the process' address space.

Several challenges were faced in the development of our object model. Firstly, it was difficult to start the implementation since the objects could be implemented in many ways like separate execution entities, execute in the process which attaches the object etc. We provide a library `object.h` which the process can link. The library makes `ioctl` calls to keep track of metadata of the objects. We maintain a per-object structure in device driver to make the `attach object` and `detach object` consistent. We discuss the specific details below.

4.1 Implementing Shared Objects

Our framework requires the objects to be shared across the processes, hence we need some inter process communication (IPC) technique. We use existing techniques like `shm`

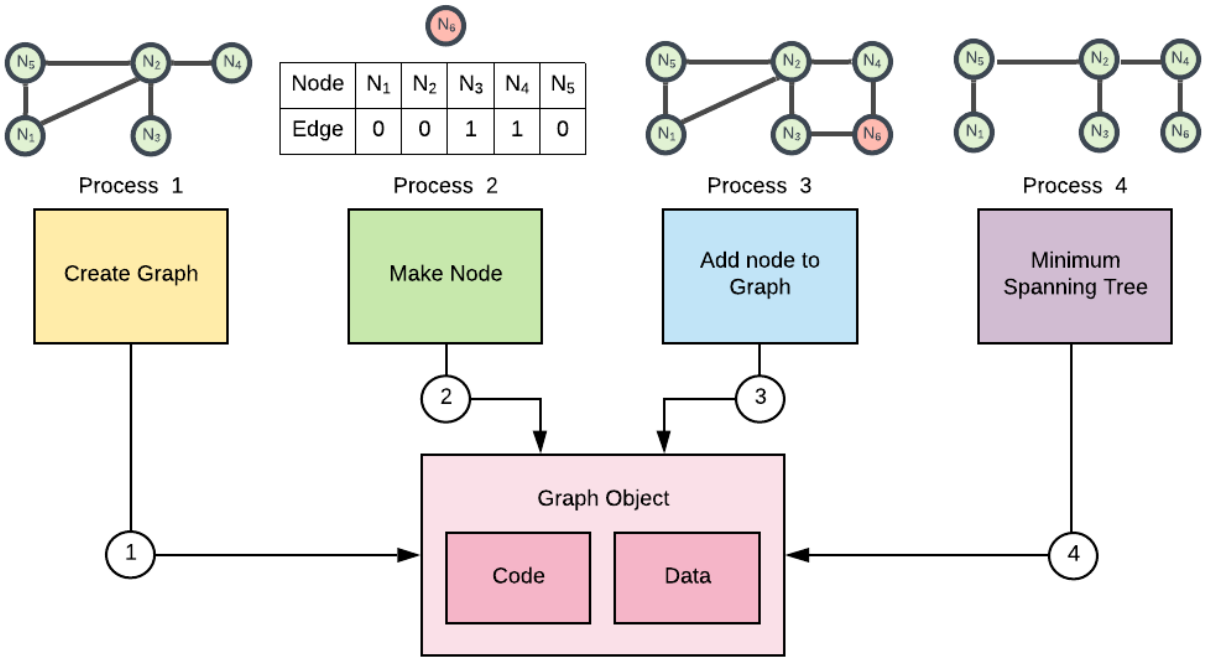

Figure 4: Graph application using our object model

and `mmap` to carry out IPC. We make use of `shm` structures to provide consistency to our model while destroying and detaching.

When a process calls for `init` with a binary, the user library sends a request to device to reserve a free `obj_id`. The library allocates shared memory segment using `shmget`. Since we only map the code and data part of the binary we need to copy the copy particular segments of the file to the shared memory space. This is done by `mmaping` the file and then copying the relevant part of the binary to the shared memory and then `unmapping` the mapped file.

The `attach` functionality is carried out by attaching the process to the shared memory using `shmat` by obtaining relevant key for the object from the device and then thereafter updating the meta data of the object in the device.

4.2 Calling Functions in objects

The functions are called through the user library. The library takes as argument and the function identifier. The library pushes the relevant parameters and return address and then changes the instruction pointer to point to the relevant code in binary. The function address can be obtained using the `elf` library and then checking the symbol table.

4.3 Resolving links in binary

We take compiled code as input to create objects. Since the compiled code can have unresolved references to library functions and data pointers, they need to be resolved. We resolve these similar to the way a linker resolves them. We make use of `elf` library functions to parse the compiled code of object and resolve the relocation by changing the relocation pointers to point to the appropriate location as directed by the symbol table of linking library. Once the relocation structures are obtained, the linked libraries are parsed through using `elf` and the offset of unresolved symbols are obtained by iteratively seraching through the symbol table of all the linked libraries.

5. APPLICATIONS

The described object model can be utilized in various fields. It finds application in cloud services, distributed systems and operating system for NVRAMs. We briefly describe these applications below.

5.1 Object based Operating System

With the huge interest in NVMs these days, it requires that we come up with an operating system that can utilize the resources of non volatile RAM. The object model described can be extended to handle object composition and making object a self sufficient entity. With this extension objects can easily store their states, making them resilient to boot cycles as opposed to the traditional OS processes.

5.2 Lambda Services

In the recent times there has been a sudden surge in the field of cloud computing. One variant of cloud computing is serverless computing, where users can provide code and data to the service which returns the data to be computed. One such service available today is AWS Lambda [10]. The problem with such kind of service is that one needs to feed in code and data everytime one needs to carry out any computation. The objects in our model are stateful and hold the modified

data and code throughout its lifetime. This makes user experience better by allowing them to only tell the function to be executed.

5.3 Integration into Distributed Systems

Our model can also be incorporated in distributed systems allowing for a better utilization of resources. This needs our model to be able to send signals to their master processes, allowing some waiting process to start execution seamlessly after receiving a signal removing the delay in between. This would work similar to pipeline processing carried out in UNIX.

6. EVALUATION

We evaluate our object model using some basic tests and an intensive test which involves using a single object to carry out graph processing by multiple processes simultaneously. The basic tests include:

1. Creating multiple objects by different processes
2. Handling a particular object via multiple processes
3. Calling functions which in turn call some other function from header files
4. Doing computation on global variables (which verifies consistent relocations)
5. Appending code to the code section of the object and executing the appended code

We also evaluate our model on a graph application as shown in Figure 4. In this graph application we create separate processes to carry out different functionalities using the same object. We create a process which creates new nodes in a graph. Apart from that we have another process that adds the node to the adjacency matrix of the graph and finally we have two processes to process the graph. One of these processes, uses the adjacency matrix to show the shortest distance between multiple nodes and second process finds out the connected components in the graph.

The experiments show that all the above mentioned test cases work seamlessly with consistency. If we carry out the graph computation using the traditional techniques then we need to load the data everytime it is changed. Our model is a stateful one, where data remembers its state.

7. CONCLUSIONS

We have implemented a computational framework involving objects, where the object can be fed with code and both. The object can be handled by other processes to carry out computation. The same object can be handled by multiple processes with the latest data and code. The processes can also append code to the already existing object. This work would soon be integrated with an NVM simulator to check our framework's utility in NVRAMs.

8. FUTURE WORK

In future we plan to extend our framework with features like composition of objects, implementing signals among objects and their master processes and improved security via restricted isolation wherever required. We also look forward to build a complete object based kernel from scratch for the support of hybrid memory systems. We also look forward to integrate the object model in one of the distributed systems applications. Also, the object model can be extended to provide APIs for services similar to Lambda services for a better user experience.

9. REFERENCES

[1] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory.," in *HotOS*, vol. 13, pp. 2–2, 2011.

[2] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao, "Emerging nvm: A survey on architectural integration and research challenges," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, pp. 14:1–14:32, Nov. 2017.

[3] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz," in *Proceedings of the 16th Annual Middleware Conference on - Middleware 15*, ACM Press, 2015.

[4] Z. Duan, H. Liu, X. Liao, and H. Jin, "Hme: A lightweight emulator for hybrid memory," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1375–1380, IEEE, 2018.

[5] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *SIGPLAN Not.*, vol. 47, pp. 91–104, Mar. 2011.

[6] T. Moore, "File system for non-volatile computer memory," Aug. 28 2001. US Patent 6,282,605.

[7] J. Xu and S. Swanson, "{NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pp. 323–338, 2016.

[8] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 13:1–13:16, ACM, 2016.

[9] D. Bittman, M. Bryson, Y. Ni, A. Govindjee, I. Cherdak, P. Mehra, D. Long, and E. Miller, "Twizzler: An operating system for next-generation memory hierarchies,"

[10] "Aws lambda services." `https://aws.amazon.com/lambda/`.