Respecting the block interface – computational storage using *virtual* objects

Ian F. Adams, John Keys, Michael P. Mesnier

Intel Labs

Abstract

Computational storage has remained an elusive goal. Though minimizing data movement by placing computation *close* to storage has quantifiable benefits, many of the previous attempts failed to take root in industry. They either require a departure from the widespread block protocol to one that is more computationally-friendly (e.g., file, object, or key-value), or they introduce significant complexity (state) on top of the block protocol.

We participated in many of these attempts and have since concluded that neither a departure from nor a significant addition to the block protocol is needed. Here we introduce a block-compatible design based on *virtual objects*. Like a real object (e.g., a file), a virtual object contains the metadata that is needed to process the data. We show how numerous offloads are possible using virtual objects and, as one example, demonstrate a 99% reduction in the data movement required to "scrub" object storage for bitrot. We also present our early work with erasure coded data which, unlike RAID, can be easily adapted to computational storage using virtual objects.

1 Introduction

Advances in storage technology are leading to extremely large, fast devices. In particular, we are seeing various forms of non-volatile memory being used for solid-state disks, notably NAND flash and persistent memory. However, I/O speeds are not keeping pace and increasingly more time is needed to transfer data between storage devices and CPUs. The available bandwidth from these fast devices is governed by the I/O stack, which includes a variety of legacy software and hardware. Though improvements to the I/O path are underway and ongoing, they will never address the fundamental problem — moving data takes time.

Consider a modern day storage server with SSDs that can deliver about 3.0 GB/sec (or 24 Gb/sec), and commodity Ethernet speeds limited to 100 Gb/sec. In building a storage server with 16 to 32 drives, this network bottleneck becomes immediately obvious, and trade-offs need to be made. Unfortunately, this often means that the parallel bandwidth of all drives within a storage server will not be available. The same problem exists within a single SSD, as the internal NAND flash bandwidth often exceeds that of the storage controller.

Enter *computational storage*, an oft-attempted approach to address the data movement problem. By keeping computation physically close to its data, we can avoid costly I/O. Various designs have been pursued, but the benefits have never been enough to justify a new storage protocol. In looking back at the decades of computational storage research, there is a common requirement that the block protocol be replaced (or extended) with files, objects, or key-value pairs – all of which provide a convenient handle for performing computation. Such attempts not only introduce too great a disruption in the computer-storage industry (e.g., readers may recall the Object-based Storage Device SCSI standard in T10 [20]), but they ignore the large existing deployments of block storage. In short, the solutions are not compatible with today's business ecosystem. We argue, however, that with the latest generation of high-speed storage devices only further widening the I/O gap, along with growing momentum behind *disaggregated* block storage (NVMeoF and iSCSI), computational storage is becoming essential. Further, existing object-based storage systems (e.g., Ceph, Swift, Hadoop) that offer computational storage will need to be adapted to use disaggregated block storage if they want to continue to co-locate computation with data. This paper represents one possible adaptation.

Rather than start with an object-based solution (file and KV can also be considered objects), we instead start with a block-based one and ask ourselves what is missing. As it turns out, very little, and the short answer is "metadata." We take the metadata that a block storage system needs to make sense of an unordered group of sectors, and we call this a *virtual object*. After all, an object is simply a collection of blocks, a length, and other metadata like timestamps and ownership. Such metadata is small, can very easily be shared with a block storage system, and it allows the storage system to read the object into its memory for subsequent processing.

We wrap the virtual object in what we call a *compute descriptor* and include any additional information that is needed to process the object data (e.g., the method to be executed and its arguments). We then send the compute descriptor from the host server to the storage system (or *target*) using a new SCSI or NVMe command, which we call *execute* (`EXEC`). The data returned by the `EXEC` command is the result of the requested computation.

A client-side library manages the creation of the virtual objects, the compute descriptors, and the issuing of the `EXEC` commands. There are no changes to the block READ and block WRITE commands, and legacy applications can continue to operate with the otherwise unmodified block storage.

We implement our approach using both iSCSI and NVMeoF targets, and we show that a variety of compute offloads are possible (e.g., word count, search, compress, checksum, video transcoding, image transformation, sorting, merging, ...). The same techniques can be applied to other forms of block storage, including disk drives, solid-state drives, disk arrays, and clustered storage systems like Amazon's Elastic Block Storage.

Our design also extends to erasure-coded data. Unlike RAID, which uses a relatively small chunk size per drive (on the order of KiB), modern day erasure coded deployments create much larger chunks (on the order of MiB). This larger chunk size gives each device (a device may be a drive, an entire storage target, etc.) sufficient data to operate on before running into a "boundary condition" where data straddles multiple devices. Comma-separated-values (CSV) represent such a use case: the large chunk size of an erasure-coded CSV file will allow each storage device to see a sufficient number of non-overlapping rows in the file, thereby allowing most operations to be performed in parallel across the devices.

2 Historical perspective and related work

Associative memory [18] is arguably the earliest work on computational storage. This evolved into what was known as "database machines," where compute primitives were embedded in the storage/memory system, solely for avoiding expensive data movement across the memory bus and into the CPU [2, 4, 7, 14, 19]. But none of these efforts took root. A retrospective paper at the time, "Database Machines: An Idea Whose Time Has Passed?" [2] signaled the end of two decades of research. Regardless of where the computation was located, the disk drives were the primary bottleneck.

Research picked up again in the mid-90s. Although primary storage was still based on rotating media, the I/O bottleneck (data movement overhead) was again increasing and motivating the need for computational storage. The focus was also on database optimization [1, 11, 15, 16], but with new attention being paid to large-scale data mining and, in many cases, image processing. The design, which is based on object storage, met a similar fate. The benefits of a new object protocol did not justify the cost, complexity, and subsequent disruption to the storage ecosystem.

With the introduction of Hadoop [3] Map Reduce in the early 2000s, we finally saw widespread computational storage, with HDFS storage nodes also serving as compute nodes. In 2004, Intel proposed a storage architecture for early discard using *in-situ* search [9], but required an object based protocol. Similar work was explored in the HPC realm, including IBM's Active Storage Fabrics [6], with compute added to Blue Gene I/O nodes. Then, with the advent of cloud computing, we saw the introduction object-based storage systems, such as Ceph and Swift. Ceph introduced the notion of dynamic classes [23], and IBM introduced the notion of Storlets for Swift [5, 17]. With each of these, computation can be associated with data, but they all require the use of an object protocol that is not compatible with block storage, and they preclude the use of disaggregated block storage as a substrate (i.e., diskless deployments) if they are to co-locate part or all of their computation with the storage. Our work is complementary as it enables computation "near" data without requiring the higher level stack be entirely co-located as well.

We are also now seeing a return to the earlier research that attempted to place compute logic within the device itself. There has been new device-level research, including both HDDs and SSDs, where the focus has been raising the level of abstraction from sectors (or blocks) to either objects or key-value pairs — both of which provide a convenient interface (the object or the key) with which computation can be associated. In particular, Seagate introduced a key-value HDD [8], Samsung introduced a key-value version of their SSD [12], and researchers have begun exploring the possibilities of processing data within such smart devices [10, 22]. Several startups (e.g., NGD Systems, OpenIO, and ScaleFlux) are also creating computational storage solutions based on object or key-value protocols, or in some cases offloading entire applications to an embedded OS in the storage device.

Other recent research [13], which we feel is closest to ours in philosophy, adds programmable filters to SSDs that persist across multiple read commands. Though adding such persistent "state" to the block protocol can, in some cases, approximate the effect of having a virtual object, we argue that a stateless protocol (virtual objects do not require that state be persisted across I/O commands) is both more general and more scalable. One of the applications of virtual objects could, in fact, be a filter.

3 Design

3.1 Architecture Overview

As an illustrative example, consider searching for a string inside of a document. This is an ideal application, as it gener-

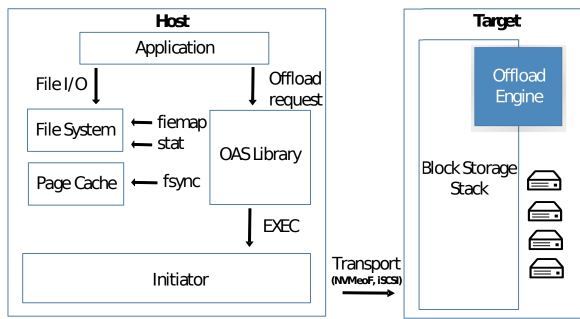

Figure 1: High level architecture and data flow for our compute-in-storage stack.

ally has much larger input than output, maximizing the I/O reduction. On the host, an application interacts with our *OAS* (*Object Aware Storage*) library, so named as it provides sufficient metadata to make the block storage target temporarily *aware* of an object, without requiring that it becomes an object storage system in its own right. A target may be an SSD, an NVMeoF target, etc. without any loss of generality. Applications use our OAS library to create the virtual objects and the compute descriptors with the desired operation – string search in this case – and any arguments (e.g., search for string "foo"). Note virtual objects are created using `stat` and `fiemap` calls. The compute descriptor is embedded into our new `EXEC` command, which is used to transport the descriptor to the storage target. The target identifies the `EXEC` command (as a vendor opcode), extracts the descriptor, and hands it to an offload engine for execution. The result of the search is written to the virtual object(s) or, if it is small enough, returned as inline data, along with status of the `EXEC` command. Figure 1 illustrates this flow.

3.2 Implementation details

Virtual objects: The virtual object is an extent list of block addresses and a data length, describing the data to be operated on. Multiple virtual objects can be included in a compute descriptor. It is important to note that these mappings are ephemeral; they do not persist on the target beyond the lifetime of an operation, allowing us to keep our target stateless, simplifying our design. Virtual object creation is handled by our OAS library, using a combination of `fiemap` – a commonly supported `ioctl` that obtains LBA extent mappings from the underlying file system – and `stat` to retrieve a file length. Note that there can be both virtual input and output objects, the latter being pre-allocated files for landing results, which we describe in more detail later.

Compute descriptor and `EXEC`: The compute descriptor tells the target which operations to run on the virtual input object(s), any function arguments, and optionally the virtual output object(s). For example, the result of our string search example may be a short list of indices, so it can simply be returned directly to the client as an inline payload from the `EXEC` command. Operations with larger results may write to the specified virtual output object(s). An example compute descriptor is shown Figure 2. The `EXEC` command is sent as a vendor specific command (using an `ioctl` for NVMeoF or a SCSI generic command for iSCSI).

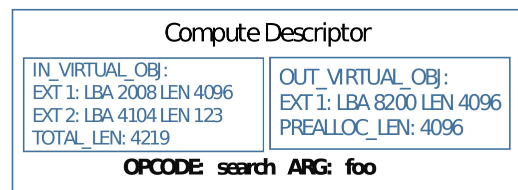

Figure 2: An example compute descriptor for a search operation, looking for string "foo." The virtual input object contains two extents, and the virtual output object contains one.

`EXEC` handling: When the target receives an `EXEC` command, the compute descriptor is extracted and handed to an offload engine, which does the actual computation on the data specified by the virtual input object(s). It is expected that the offload engine is a library of computational services (e.g, checksum, search, word count, sort, merge, ...) – many of which we have implemented.

When virtual output objects are used, the output sizes may not be deterministic (e.g., compressing a file). In these cases, the target must also return the length of the output, and the host must truncate the pre-allocated files to the correct size.

Care must be taken with virtual objects when it comes to cached data. Prior to issuing the EXEC command, an *fsync* must be issued on all virtual input objects. This ensures the storage target has the latest copy of the data. Upon completion of an operation, the host must carefully truncate its pre-allocated virtual output objects to the correct size. By "carefully," we mean that the host must bypass any pre-allocated data buffers that may be present in the OS page cache.

Fortunately, our OAS library can handle all of these details. We maintain a pool of pre-allocated output files. These files are selected in a best-fit manner, depending on the operation in question. In addition, for results that are known to be temporary (but are too large to return as inline data), we use a large pre-allocated file as a circular buffer, and return an offset and length to the application. The application can then read the result at its leisure.

4 Evaluation

We evaluate a non-erasure coded application in detail in the form of checksum offloads for bitrot detection.

Bitrot detection is a process where object data is hashed and compared against a previously computed hash to detect corruption. This corruption may come from a variety of sources, such as software bugs, or latent sector errors [21]. In

traditional object storage with smaller, direct attached storage devices this overhead is relatively minimal. However, two trends are making this common operation increasingly costly. First, storage devices are becoming ever larger, requiring more and more throughput to effectively catch object corruption. Second, we are seeing a move towards "diskless" storage stacks. For example, Ceph, Swift, and MinIO deployments may use disaggregated block storage (NVMeoF or iSCSI) in place of local disks,

This means that bitrot detection, traditionally a purely local operation, now incurs significant network traffic, which can contend with latency sensitive application traffic. For example, a monthly scrubbing cycle for 1 PB of stored data requires approximately 400 MB/sec, continuously.

Experimental setup: The target system is populated with 8 Intel P4600 1.6 TB SSDs, with dual Xeon Gold 6140s. The host/initiator system has dual Intel Xeon E52699s. Both systems run Ubuntu 16.04 with a 4.13.2 kernel. We run our modified host and target NVMeoF drivers to enable the `EXEC` command and offload engine. Host and target are connected via a 40 GbE link using Chelsio T580-LP-CR cards.

We use a benchmarking utility of our own design, implemented in C++. It compares conventional operations (i.e., reading data over the network and computing host side) with offloaded operations that are computed *in-situ* on the block storage target. The result is returned as an NVMe payload instead of a virtual output object.

Each experiment uses 100 16 MB files on each SSD, with 48 worker threads. We measure the throughput from 1 to 8 devices. All conventional operations are measured with a cold page cache, forcing SSD reads, and offloaded operations read directly from the target side SSD(s). We compare conventional and offloaded performance for a pure software CRC, accelerated ISA-L CRC, and an AVX accelerated Highway Hash, all of which are representative examples of bitrot check hashes.

Performance and network evaluation: Our first set of experiments focus on conventional accesses. As we can see in Figure 3, conventional performance is rapidly gated by our 40 GbE link, as expected, at around 4.5 GB/sec.

However, our offloaded checksums scale with the number of devices being read from, bypassing not just the 40 GbE link but a hypothetical 100 GbE link as well, illustrated in Figure 4. The HighwayHash and ISA-L CRC are computed at near SSD speeds, meaning that bitrot checks not only do not take up limited network resources, they actually run faster. In turn, this implies that bitrot checks can be efficiently run more frequently, reducing data loss risks. While subjective, it should also be noted that this is with an otherwise unoptimized stack, so we predict we can continue to scale with the number of SSDs as well as improve our existing performance.

Though not shown here, our offloaded checksums also reduced network traffic by over 99%. This is due to the fact

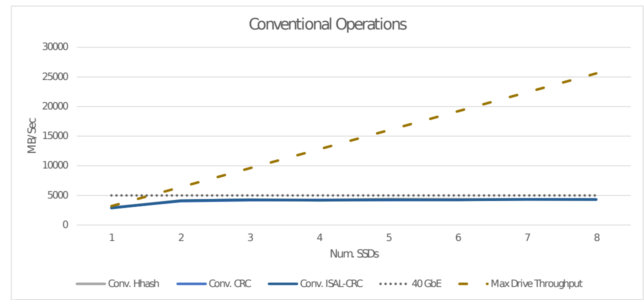

Figure 3: Conventional access throughput for checksums. Dotted lines refer to theoretical maximum throughput of network and SSDs

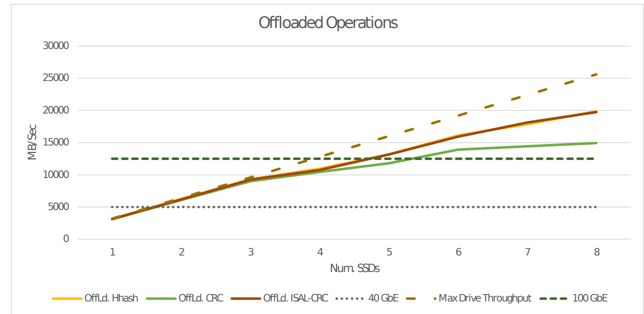

Figure 4: Offload throughput for checksums. Dotted lines refer to theoretical maximum throughput of network and SSDs.

that instead of moving bulky data from the target to host, we are simply returning the digests.

Object storage integration: In order to demonstrate our solution end-to-end, we integrate our checksum offload into the Ceph, MinIO, and Swift object storage stacks. In addition to the network traffic reductions (upwards of 99%), we also see up to a 2x improvement in scrubbing performance.

Overall, we find that it is simple to integrate our computational storage solution into these object stores. Using our OAS libraries, we generally need to change only 20-30 lines of the host scrubbing code, and in some cases we actually shrink the bitrot check functions by moving most of the complexity to our OAS library.

5 Ongoing and future work

Erasure coded deployments: Traditional RAID, where data and parity chunks are often only a few kilobytes in size, breaks computational storage by making it difficult, if not impossible, to meaningfully compute on data without full reassembly of the chunks. Modern erasure coded systems, however, have changed this dynamic with two key facts. First, erasure coded data is often written in relatively large chunks of hundreds of kilobytes to low megabytes in size. Excluding parity, this data is often (assuming it is not encrypted) visi-

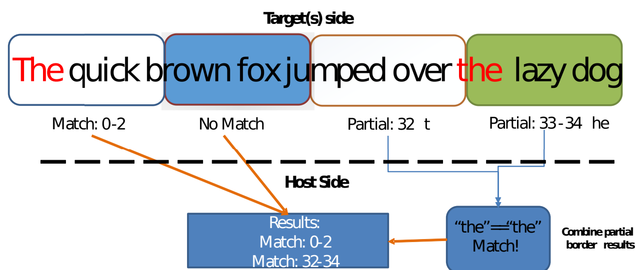

Figure 5: Example of operating on erasure coded data (ignoring parity) across targets. Searching for the string "the," the first target matches the whole word, and the third and fourth have partial matches that are returned to the host. The host-side reconstructs these and returns results to the application.

ble "in the clear" when using common codes such as Reed-Solomon. Second, huge amounts of data are often composed of well structured elements that are significantly smaller than these chunks (e.g., individual CSVs). It then becomes relatively simple for computations to occur on complete elements within a chunk, and those data elements that straddle chunks can be returned and reassembled on the host. Not only do we glean most of the traffic reduction for many types of operations, we run in parallel like a Map-Reduce cluster.

This type of stateless functionality inside of erasure-coded block storage targets is not possible without virtual objects. We would either require a more complex stateful protocol to obtain this information from the host, or centralize and reassemble the data prior to computing on it, eliminating any benefit. Even existing erasure coded object-based approaches require that the object be at least partially reassembled before performing operations. In contrast, we treat each chunk of erasure coded data as a distinct virtual object, illustrated in Figure 5.

To explore our methodology, we are in the process implementing simple offloads, such as word count. In this process, we find that the most challenging aspect is formalizing the erasure coded layouts (e.g., Ceph and MinIO have different layouts, such as the chunk size). Once that is understood, it is a straightforward matter to identify the boundaries between blocks and divvy up work to the various storage targets. For example, with word count, words that potentially cross boundaries are not counted and are returned to the host along with an initial count of words found in a chunk, with the final tally adjusted host side. This not only demonstrates the initial concept, but it shows that we can achieve Map-Reduce like functionality that would otherwise not be possible with disaggregated block storage. We are next looking towards more complex operations such as querying CSV and JSON files in erasure coded deployments.

Additional offloads: We also are exploring many other offloads as well with our initial prototypes. Some of these, such as sort and merge, demonstrate common tool operations, and exercise the handling of large virtual output objects. It is in these use cases that we identify the importance of having pools of pre-allocated storage to draw from, as pre-allocating before every call erodes the performance of our offloads.

Several other application specific offloads are also in progress. For visual data machine learning applications we are investigating image processing, which is currently showing network traffic reductions of up to 90% when integrated into a Caffe based image classification pipeline. We also are exploring LSM-tree compaction for KV stores like RocksDB.

Offload scheduling: A critical area for our future explorations is scheduling and work placement. Not all operations are appropriate for computational storage. There also times where applications can coordinate with computational storage to maximize their throughput by selectively offloading some operations, minimizing resource contention while maximizing throughput.

Security: Security is an often overlooked portion of computational storage research. Since our approach works through the file system, we can leverage basic security features both from the file system and the OS. If an application does not have the permissions to read/write a file, it will fail to create the associated virtual objects and compute descriptor. These checks can be performed in our OAS library.

6 Conclusion

We are encouraged by our early results (a 2x improvement in scrubbing performance and a 99% reduction in I/O traffic), and we are actively exploring many other computational storage use cases with virtual objects. Given the growing trend toward disaggregated block storage (e.g., NVMeoF), we believe computational storage is becoming essential.

In summary, relative to other approaches that we and others in the industry have taken, virtual objects offers a number of unique advantages –

- [Block compatibility] We do not modify the host file system or the block storage system READ/WRITE path. Rather, we offer the application the option of using our EXEC command.
- [Stateless protocol] Our EXEC command is stateless. This simplifies the design and improves scalability.
- [Erasure coding support] Our solution extends to erasure-coded block storage.
- [Simplified security] Because we do not modify the FS, we can leverage existing security mechanisms in the FS and OS.

It was widespread adoption that motivated our design. As such, we believe that these advantages are what make virtual objects a practical (i.e., scalable, cost-effective, and ecosystem-compatible) approach to computational storage.

7 Topics for discussion at HotStorage'19

Broadly, we are interested in feedback and discussion related to the feasibility and general applicability of our approach. In particular, we strongly feel that previous attempts at computational storage are not practical in today's ecosystem and, because of this, will see limited deployments in industry. While we agree that, in a perfect world, a new protocol and "smart" devices that are fully programmable would be ideal, our experiences across multiple decades (and companies that we have worked at) suggest otherwise. In short, many of the challenges with computational storage are not technical, and we feel that a new approach (virtual objects) is needed to get past the underlying business concerns that have historically presented a challenge — expensive storage devices with potentially under-performing CPUs, and the technical and standards upheaval required to integrate them into existing stacks.

On a more technical note, given the early nature of our erasure coding work, we are very interested in feedback from others that have made similar attempts, as very little is found in the literature. We feel that the computational storage is much easier with erasure coded data (when compared to RAID), and we are interested to learn how others might attack the problem for big-data formats beyond just CSV.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel H. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 2-7 1998. ACM Press.

[2] Haran Boral and David J. DeWitt. Database Machines: An Idea Whose Time Has Passed? In *International Workshop on Database Machines*, Munich, September 1983. Springer.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, San Francisco, CA, December 5 2004. The USENIX Association.

[4] David J. DeWitt and Paula B. Hawthorn. A Performance Evaluation of Database Machine Architectures. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB'81)*, Cannes, France, September 9-11 1981. IEEE Computer Society.

[5] Michael Factor. Storlets: Turning Object Storage into a Smart Storage Platform. https://www.ibm.com/blogs/research/2014/05/storlets-turning-object-storage-into-a-smart-storage-platform/.

[6] Blake G. Fitch, Aleksandr Rayshubskiy, Michael C. Pitman, T.J. Ward, and Robert S. Germain. Using the Active Storage Fabrics Model to Address Petascale Storage Challenges. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW'09)*, Portland, OR, November 14 2009. ACM.

[7] D.K. Hsiao. Database Machines are Coming, Database Machines are Coming! *IEEE Computer*, 12(3):7–9, 1979.

[8] James Hughes. Seagate Kinetic Open Storage Platform. In *Proceedings of the 30th IEEE International Conference on Massive Storage Systems and Technology (MSST'14)*, Santa Clara, CA, June 2-6 2014. IEEE.

[9] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, Mahadev Satyanarayanan, Gregory Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, San Francisco, CA, March 2004. The USENIX Association.

[10] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling Cost-effective Data Processing with Smart SSD. In *Proceedings of the 29th IEEE International Conference on Massive Data Storage (MSST'13)*, Long Beach, CA, May 6-10 2013. IEEE.

[11] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *ACM Special Interest Group on Management of Data (SIGMOD) Record*, 27(3):42–52, September 1 1998.

[12] Yang Seok Ki. In-Storage Compute: an Ultimate Solution for Accelerating I/O-intensive Applications. In *Proceedings of the Flash Memory Summit*, Santa Clara, CA, August 10-13 2015.

[13] Gunjae Koo, Kiran Kumar Matam, Te I, Hema Venkata Krishna Giri Nara, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50'17)*, Cambridge, Massachusetts, October 14-18 2017. ACM.

[14] E.A. Ozharahan, S.A. Schuster, and K.C. Smith. RAP - Associative Processor for Database Management. In *American Federation of Information Processing Societies (AFIPS) Conference Proceedings*, Anaheim, California, May 19-22 1975. ACM.

[15] Erik Riedel and Garth Gibson. Active Disks - Remote Execution for Network-Attached Storage. Technical Report CMU-CS-97-198, Carnegie Mellon University, December 1997.

[16] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia Applications. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB'81)*, New York, NY, August 24-27 1998. Morgan Kaufmann.

[17] Eran Rom. Bringing Compute to Open Stack Swift. In *OpenStack Israel*, Tel Aviv, Israel, June 15 2015.

[18] A.E. Slade and H.O. McMahon. A cryotron catalog memory system. In *Proceedings of the Eastern Joint Computer Conference (EJCC 1956)*, New York, NY, December 10-12 1956. National Joint Computer Committee.

[19] S.Y.W. Su and G.J. Lipvski. CASSM: A Cellular System for Very Large Data Bases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'94)*, Framingham, Massachusetts, September 22-24 1954. ACM.

[20] Technical Committee T10. SCSI Storage Interfaces. http://www.t10.org.

[21] Vinodh Venkatesan and Illias Illiadis. Effect of Latent Errors on the Reiliability of Data Storage Systems. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, San Francisco, CA, August 14-16 2013.

[22] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. SSD in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN'16)*, San Francisco, CA, June 26 - July 1 2016. ACM.

[23] Noah Watkins. Using Lua in the Ceph distributed storage system. In *Proceedings of the Lua Workshop*, San Francisco CA, October 16-17 2017.