Quantifying CPU and Network savings in Computational Storage Systems

Jayjeet Chakraborty
jayjeetc@ucsc.edu

June 16, 2023

Abstract

Computational Storage allows offloading compute operations from clients to storage servers to be able to mitigate performance implications due to client-side CPU bottleneck and reduced scalability. Since the compute capability of the client is distributed across the storage tier in computational storage, we might be able to use cheap low-power CPUs on the storage servers, to be able to retrieve equivalent performance as client-side computation while enjoying the benefits of compute offloading such as reduced network bandwidth. In this paper, we aim to find out how much resource savings in terms of CPU and Network can one achieve by adopting computational storage.

1 Introduction

In the past decade, I/O hardware like the Disks and the Network interconnects have become faster than ever. With the introduction of protocols such as NVMe, SSDs can now sustain a throughput of 2-3 GB/s. Also, with innovations in networking hardware and RDMA technologies like Infiniband and RoCE becoming mainstream, it is common to find link speeds of 25-100 Gbps in modern data centers. But, with Moore's law coming to an end, CPU processing capabilities have failed to keep up with the fast I/O devices, causing the performance bottleneck to shift from the I/O devices to the CPU. The notion of "a fast CPU and slow disk" is rendered invalid in today's high-performance systems as shown by Trivedi et. al. [TSP+18]. One solution to this problem is to move the processing from the client to the storage layer so that the single client CPU is no more the bottleneck. Offloading compute to the storage layer gets the computation access to the storage layer CPUs in a scalable manner enabling the system to benefit from the high-performance I/O devices. Another benefit of using computational storage is that it reduces unnecessary data movement from the storage to the client resulting in network bandwidth savings, reduced latency, and high throughput. Since the computation is taken from the client and distributed amongst the storage layer, every storage server does a fraction of the total work and does not utilize all of its CPU cycles. Also, when only the result of a computation is sent back to the client from the storage layer, only a fraction of the network bandwidth is used. In this paper, we aim to quantify the resource savings in terms of CPU and network bandwidth that one can expect to get by using computational storage. We aim to find out if adopting computational storage can help lower infrastructure costs by enabling the use of cheaper CPUs and interconnects while preserving competitive performance. We use Skyhook [CJR+22], as our candidate computational storage system for performing this study. It is described in detail in Section 2. Overall, the paper is organized as follows: Section 2 goes over the background information that is required to understand this paper, Section 3 describes the experiments in detail along with the observations, and finally, Section 4 discusses future work and concluding thoughts.

2 Background

This section discusses in detail the different systems and technologies that we use to perform our experiments.

2.1 Apache Arrow

Apache Arrow [Tea18] is an in-memory columnar format optimized for efficient analytics operations on modern hardware. It describes a standard data format for exchanging structured data between different systems without serialization or deserialization. The Arrow format allows compute engines and query execution engines to maximize their efficiency when scanning and iterating large chunks of data. The contiguous columnar layout of Arrow enables vectorization using the latest SIMD operations on modern hardware. Besides being an in-memory format, it is also a collection of different data processing components that allow building parts of a data processing system. Some of the most-used components are Flight, a gRPC-based data transfer protocol [fli19]; Feather, A Arrow-based columnar persistent storage format [Wic]; Gandiva, An LLVM-based expression compiler [gan18]; Dataset API, An abstraction for realizing datasets over a directory of files [Arr]. Arrow is also language-independent as it has APIs in several different programming languages such as C++, Java, Python, Rust, R, JavaScript, and Julia. Several popular data processing systems such as Spark, Dask [Roc15], Ray [MNW+18], Pandas, Parquet [par] have added support for Arrow data and Arrow data sources.

2.2 Ceph

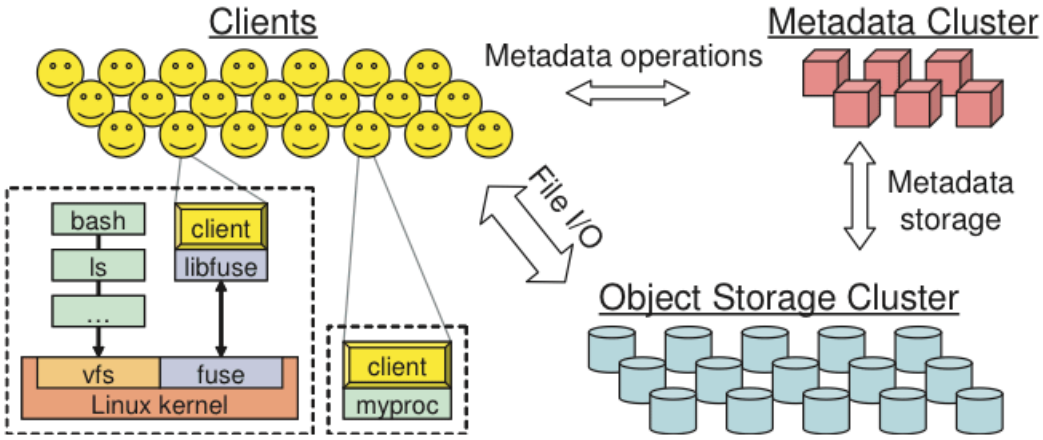

Figure 1: Architecture of Ceph.

Ceph [WBM+06] is a petabyte-scale distributed and programmable object storage system providing 3-in-1 interfaces for object, file, and block-level storage. Ceph was developed as a part of a Ph.D. thesis at UC Santa Cruz and was first published in 2006. Ceph does not use any other filesystems internally; rather, it manages the HDDs and SSDs directly with its custom storage backend, BlueStore, part of its RADOS [WLBM07] object-store. Ceph was developed for commodity hardware, and hence it can replicate data across a cluster, employing several techniques such as erasure coding, replication, and snapshots. Ceph is unique as it does not have a single point of failure. This is because of the CRUSH [WBMM06] map feature that Ceph provides. CRUSH maps contain object-OSD mappings, which the Ceph client uses

to calculate the location of an object in a Ceph cluster. After figuring out the location of an object, the client directly connects to a Ceph OSD and reads the object. Ceph also provides a plugin-based object-store extension mechanism through its Object Class SDK [obj]. This SDK allows writing embeddable plugins (in the form of shared libraries) in C++ and Lua containing logic to access and modify objects inside the storage servers within the RADOS I/O path. The SDK provides a subset of POSIX-like system calls such as `cls_cxx_read`, `cls_cxx_write`, and `cls_cxx_stat` to be able to read, write, and stat objects respectively. This SDK is heavily used by several Ceph components such as Ceph RBD (Rados Block Device) and CephFS (the Ceph filesystem). The architecture of Ceph is given in Figure 1.

2.3 Skyhook

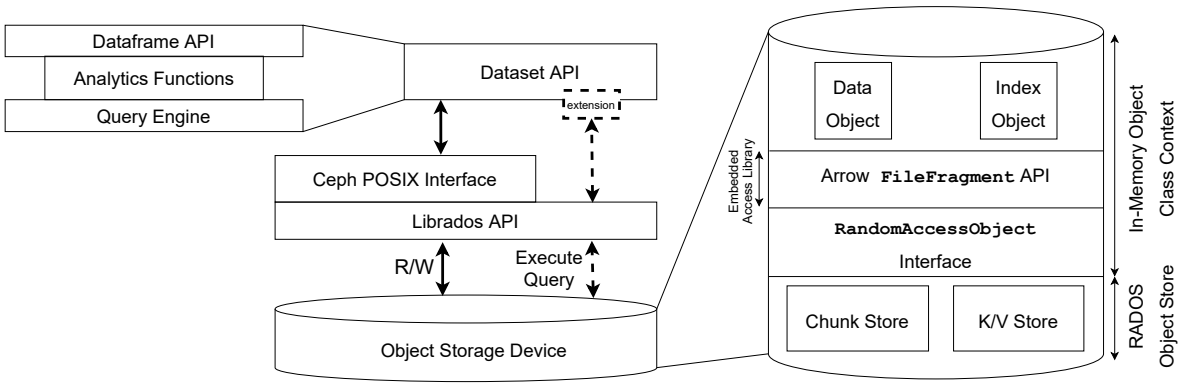

Figure 2: Architecture of Skyhook.

Skyhook [CJR+22] is a programmable storage system built on top of Ceph that can offload query executions from the client to the storage layer. Skyhook supports scanning datasets containing files of different formats such as Parquet, Feather, CSV, JSON, or any other format as long as they are supported by Apache Arrow. Skyhook is built as a storage-side plugin for Ceph, which, when embedded inside Ceph OSDs in the form of shared libraries, allows executing queries within the Ceph storage servers. Skyhook is exposed to the clients using a Arrow `FileFormat` API extension called the `SkyhookFileFormat` API. This API, when used with the Arrow `Dataset` API, allows offloading dataset scans instantly. Internally, `SkyhookFileFormat` leverages Ceph filesystem metadata containing file striping information to map files in CephFS to RADOS objects and applies computation on those objects directly, bypassing the POSIX layer. In the storage plugin, we reuse the `ParquetFileFormat` of Arrow out-of-the-box to scan RADOS objects containing Parquet binary data. Generally, Arrow APIs cannot scan RADOS objects. However, we make this possible by creating a thin random-access filesystem shim called the `RandomAccessObject` API that wraps a RADOS object, keeps track of the file pointer, and provides a file-like view over the objects. This interface plugs into Arrow APIs seamlessly and allows scanning objects as files. One exception for Skyhook is that it requires files to be written in a specific way. To be able to scan Parquet files with Skyhook, they should be self-contained within a single RADOS object. This $1:1$ mapping is required to be able to translate filenames into object IDs easily and to let Arrow APIs scan a RADOS object as a complete Parquet file, as Arrow APIs cannot scan multiple physical files as a single logical file. To ensure every file goes into a single object while writing files to CephFS, we change the stripe unit of CephFS to match the size of the Parquet files being written. The architecture of Skyhook is given in Figure 2.

3 Experiments

In this section, we describe our experiment setup, how we perform our experiments, and our observations. We perform 2 kinds of experiments to show CPU and network bandwidth savings from offloading compute to the storage layer. The experiments were performed on CloudLab, the NSF-funded bare-metal-as-a-service infrastructure [DRM+19]. For our experiments, we exclusively used machines with an 8-core Intel Xeon D-1548 2.0 GHz processor (with hyperthreading enabled), 64 GB DRAM, a 256 GB NVMe drive, and a 10 GbE network interface. These bare-metal servers are codenamed "$m510$" in CloudLab. We used Ceph octopus v15.2.16 on Ubuntu 20.04 (Linux kernel $5.4.0-100$-generic) along with Skyhook v0.4.0 for our experiments. We used Ceph clusters of sizes 1, 8, and 16 for our experiments as these sizes are pretty common for a standard Ceph deployment. Every storage server had a single Ceph OSD running on the local NVMe drive. We configured the OSDs to use 8 threads to prevent any contention due to hyperthreading in the storage servers. The number of placement groups was increased from 128 to 512 to prevent contention from PG locking. A CephFS interface was created on a replicated data pool and was mounted in user mode using the `ceph-fuse` utility. We repeat each experiment 5-10 times to produce somewhat statistically significant results.

3.1 CPU Frequency

One of the goals of this paper is to find out the minimum amount of processing capability required per storage server CPU to be able to generate the same performance as a fast client-side CPU can provide. We take Skyhook as our computational storage system and execute queries to select 1% of the rows both on the client and the server-side. We use the `cpufrequtils` toolchain provided by Linux to change the CPU speed on the storage servers. We discuss how to use this tool in detail in Appendix A. Since the lowest frequency that we can set our cores at was 800 MHz, we performed our experiments in the frequency range of 800 MHz and 2000 MHz. We repeated this experiment on a Skyhook cluster of 1, 8, and 16 storage servers respectively to find out how much CPU one can save per server by scaling out. We performed weak scaling by increasing our dataset size slowly to twice the number of files as we scaled out from 8 to 16 storage servers. Our dataset was comprised of replicated 32MB uncompressed Parquet files containing data from the infamous NYC taxi dataset [yel]. Each file consisted of 1.4 million rows and 17 columns. For every cluster size, we start with a client-side query execution with both the client and server-side CPUs at 2000 MHz. Then we perform a storage side query execution with both the sides still at 2000 MHz. After that, we slowly tune down the storage side CPU frequencies by a difference of 200 MHz at each step and perform storage side query executions for each step. We repeat this till we find the query performance to match the client-side query execution at 2000 MHz on both sides. We observe that in a single server Skyhook cluster, we cannot use a low-power CPU on the storage because there is only a single storage layer CPU to match the client CPU, so both their frequencies should be the same. On the other hand, when we have a cluster with 8 and 16 servers respectively, we can tune down the CPU frequencies to about 1100 MHz and 800 MHz respectively, and still get similar or better performance. Ideally, for the 16 server cases, we should have been fine running the storage server CPUs at 600 MHz but the kernel only allowed us to tune down the CPUs to 800 MHz. Figure 3 shows our observations for this experiment.

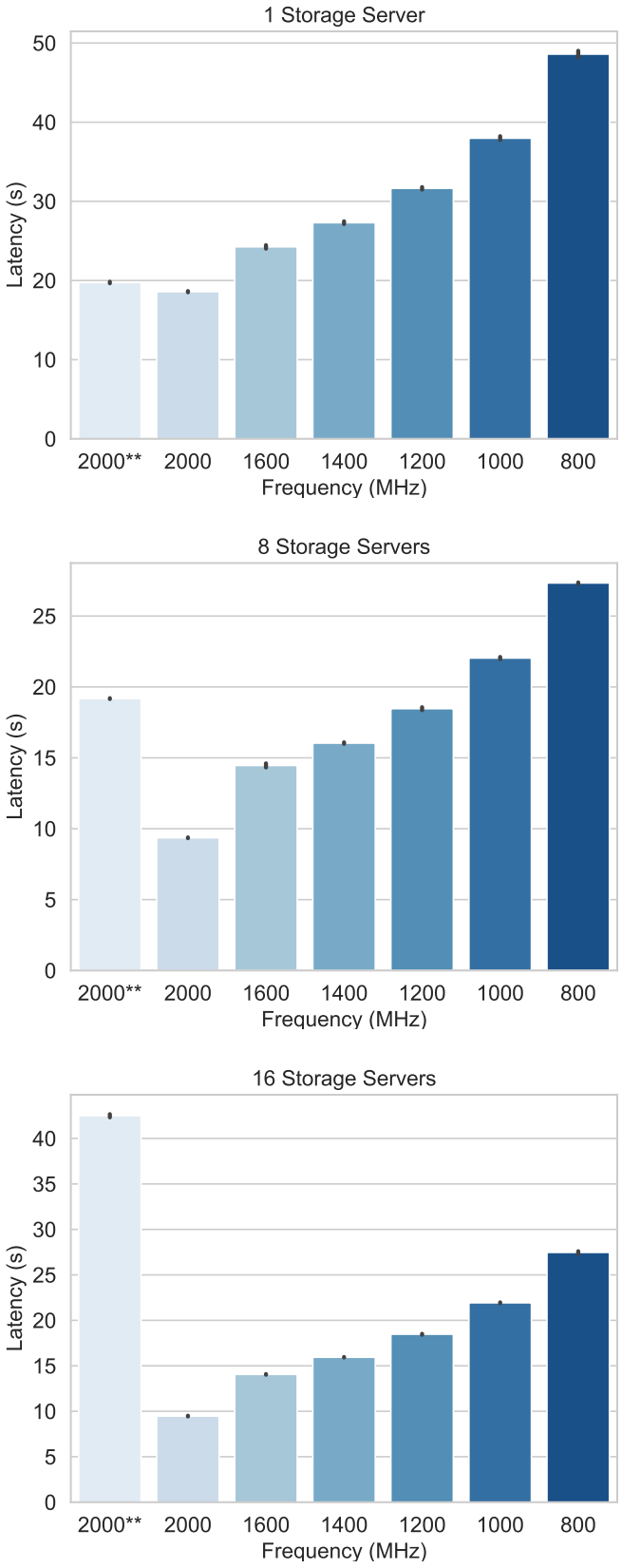

Figure 3: Latency vs Storage server CPU frequency plot for a Skyhook (storage-side) query execution with 1% query selectivity. One exception is that the leftmost bar (**) in every plot shows the latency of a client-side execution with a 2000 MHz CPU serving as the baseline.

3.2 Network Bandwidth

We performed an experiment to find out if offloading compute with Skyhook allows using a network with less bandwidth. We used a setup with 8 storage nodes and a dataset of 460 Parquet files similar to the CPU experiment above and performed 1% row selectivity experiments. We used the `wondershaper` Linux utility to constrain bandwidth at the network interfaces of the client and storage servers. Appendix B shows how to use the `wondershaper` utility. We start with a client-side query execution with all interfaces at 10 Gbps, then we do a server-side execution, and then we constrain the network interfaces on both sides to 5 Gbps and perform a server-side execution again. Unfortunately, after constraining the bandwidth to 5 Gbps, the query performance did not stay the same as before constraining but dropped by a factor of more than 10. Ideally, performance should not be hampered by moving to low bandwidth networks in computational storage systems, but this observation was unexpected and we aim to find out the reason behind this in future work. For now, we think that thinning the network resulted in the storage layer not being able to accept as much requests as it should be to maintain the query performance.

3.3 Cost

In our experiments, we show that CPUs with less power can be used on the storage servers in a computational storage system. We observe that a storage side CPU of about 1100 MHz and 800 MHz suffices for a cluster of 8 and 16 servers respectively instead of a 2 GHz CPU. According to Intel's website, a 8-core Xeon D-1548 (2 GHz) processor costs $594, whereas a 2-core Intel Celeron 847 (1.1 GHz) costs $134 and an Intel E2160 Pentium Dual-Core Processor (800 MHz) costs only about $90. This shows that a lot of infrastructure costs can be saved by moving to computational storage and the savings become even more significant in large clusters.

4 Conclusion and Future Work

This paper studies resource savings that computational storage systems can bring in terms of CPU and network bandwidth as compared to systems where computation happens on the client. We use Skyhook as our candidate computation storage system for performing the experiments. Our results show that using computational storage certainly helps save CPU resources on the storage servers and the savings increases as we scale out and distribute our computation across more storage servers. This shows that there is a real possibility of using low-power chips such as FPGAs embedded on SSDs for compute offloading. We performed experiments to find out if a thinner interconnect can be used while preserving the performance in computational storage systems, where we found that it cannot be done, and moving to lower bandwidth networks affects performance severely. Regardless, from previous literature, we know that moving to computational storage allows saving network bandwidth which other applications can use, and this becomes particularly important in data centers where the servers might be far away from each other, such as on different racks. In future work, we need to perform these experiments on actual Smart SSDs (SSDs embedded with FPGAs), so that we can collect metrics that would help create a model of performance versus Smart SSD power. Besides trying to complete the unfinished work of achieving competitive performance in low bandwidth networks, we also need to experiment with faster network interconnects such as 40 Gbps or 100 Gbps links to be able to understand the effect of computational storage in high-speed networks.

5 Acknowledgements

Many thanks to Prof. Peter Alvaro for his fantastic teaching in CSE 232-Distributed Systems and to all of my classmates who helped me make the most out of this class.

References

[Arr] Arrow dataset api. https://arrow.apache.org/docs/python/dataset.html.

[CJR+22] Jayjeet Chakraborty, Ivo Jimenez, Sebastiaan Alvarez Rodriguez, Alexandru Uta, Jeff LeFevre, and Carlos Maltzahn. Skyhook: Towards an arrow-native storage system. *CCGrid*, 2022.

[DRM+19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[fli19] Introducing apache arrow flight: A framework for fast data transport. https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/, 2019.

[gan18] Gandiva: A llvm-based analytical expression compiler for apache arrow. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/, 2018.

[MNW+18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.

[obj] Ceph object class sdk. https://docs.ceph.com/en/latest/rados/api/objclass-sdk/#sdk-for-ceph-object-classes.

[par] Apache parquet. https://parquet.apache.org/.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 126. Citeseer, 2015.

[Tea18] Arrow Development Team. Apache arrow. https://arrow.apache.org, 10 2018.

[TSP+18] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. Albis: High-performance file format for big data systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 615–630, 2018.

[WBM+06] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[WBMM06] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31. IEEE, 2006.

[Wic] Hadley Wickham. Feather: A fast on-disk format for data frames for r and python, powered by apache arrow. https://www.rstudio.com/blog/feather/.

[WLBM07] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 35–44, 2007.

[yel] Nyc yellow taxi trip record data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

A Manipulating CPU frequency

We use the `cpufrequtils` toolchain provided by Linux to change the CPU frequency on the storage servers. To be able to use this tool, we first need to ensure that the `userspace` governor is enabled in the kernel. This can be checked by executing the command below:

```
cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

If there is no `userspace` governor enabled, it means that the machine is using the new `intel_pstate` driver which only has the `powersave` and `performance` governors. To be able to enable the `userspace` governor, we need to disable the `intel_pstate` governor first. This can be done by adding `intel_pstate=disable` in `/etc/default/grab`, followed by an `update-grub` and a reboot. Upon reboot, the `userspace` governor should be available and we can now use cpufrequtils. Given below is an example to change the frequency of Core 0 to 1000 MHz.

```
sudo cpufreq-set -c 0 -f 1000MHz
```

B Manipulating network interface bandwidth

We use the `wondershaper` utility to constrain our network interfaces to a certain network bandwidth. Before using `wondershaper`, we first need to find the network interface that we are interested in using the Linux utility `ifconfig`. Then we need to clear any previous setting on the network interface, for example, `eno1d1`, by executing the command as shown below.

```
sudo wondershaper clear eno1d1
```

Now, we can go ahead and set the upload and download bandwidth on the network interface by executing a command similar to the one shown below. `wondershaper` takes the upload and download bandwidth values in Kbps.

```
sudo wondershaper eno1d1 5120000 5120000
```

In this case, we constrain both the upload and download bandwidth on `eno1d1` to 5 Gbps.