Efficient Data Eviction across Multiple Tiers of Storage.

Jaime Cernuda
Illinois Institute of Technology, Chicago
jcernudagarcia@hawk.iit.edu

Hugo Trivino
Illinois Institute of Technology, Chicago
hhernandeztrivino@hawk.iit.edu

Hariharan Devarajan
Illinois Institute of Technology, Chicago
hdevarajan@hawk.iit.edu

Anthony Kougkas
Illinois Institute of Technology, Chicago
akougkas@iit.edu

Xian-He Sun
Illinois Institute of Technology, Chicago
sun@iit.edu

1 EXTENDED ABSTRACT

Data-intensive computing offers unprecedented opportunities for scientific discovery, high-fidelity insights, and data-driven decision making. However, I/O has become a major challenge for extreme scale computing [4] due to the unparalleled magnitude of data movement. The de facto solution to these challenges has been addressed by using large scale Parallel File Systems (PFS). Modern storage environment have proposed the addition of multiple layers of storage between the applications and the PFS. In these environments, upper layers have a higher bandwidth at the cost of reducing storage capacity, with the PFS as the bottom layer having a hypothetical infinite capacity. This difference in capacities and speeds between storage layers makes it extremely challenging to evict data from upper layers to lower layers efficiently. Currently, the transfer of data between these storage layers is achieved by either caching or buffering the data within the intermediate layers allowing the environment to match the performance gap between two successive layers.

Scientists have proposed various software solutions to address the performance gap between layers. *Data Buffering* is one solution in which the data is accumulated in fast intermediate-storage-layers and are eventually flushed down. For instance, Data Elevator [1], Univistor [5], and Hermes [3] are examples of such a buffering system. A second approach is *Data Aggregation* in which data from various nodes is collected by an aggregation layer composed of specialized hardware, such as burst buffers or I/O forwarders. These layers then flush the data collected into the underlying archival storage (e.g., PFS). Finally, there is also *Collective I/O* in which processes of an application coordinate themselves before performing an I/O operation. This is commonly achieved in HPC clusters by using MPI collective I/O frameworks [2]. These software solutions play a critical role in addressing the growing I/O gap and in enabling efficient scientific discovery.

These techniques not only enhance the I/O efficiency of an application, but they also have in common that they trigger the asynchronous flushing of data to the archival storage. The importance of the latter operation is further exacerbated in applications (e.g, The Square Kilometer Array or the Atlas experiment at the Large Hadron-Collider) where data production exceeds the size of intermediate buffering layers (e.g., RAM, NVMe, or burst buffers) since the overall performance of these applications is closely tied to their ability to move data to/from larger storage clusters in a timely and efficient manner.

Data flushing suffers from several challenges. *Firstly*, although buffering data into faster layers improves the application throughput and latency, the data flushing of these layers into the final archival storage is bound by the speed of the lowest layers. This creates a gap in performance during the data flushing between two layers in which the data can be written faster into one layer than it can be evicted from the current into the subsequent layer. *Secondly*, all software based solutions perform data flushing in batches which increases the read latency for accessing data. *Thirdly*, traditional data flushing is implemented using push-based architectures, an operation which is typically hidden between computational phases. For many applications, it can become impractical and a misallocation of resources to meet this computational requirement since I/O is bounded by the slowest layer. In order to deal with these challenges, we ought to move towards a near-real time pull-based architecture to improve the efficiency of data eviction between layers.

To address the above challenges, we present RFlush, a real-time data flushing platform for multi-tiered storage environments. The core of RFlush is a streaming architecture with exactly once semantics where all operations are decoupled and performed in parallel. This architecture allows RFlush to provide a low latency and autoscaling capabilities while also providing an efficient pipeline for continuous data flushing operations to enable high resource utilization. RFlush stands between the buffering platform and the archival storage, and it performs continuous data flushing operations through a server-pull mechanism, where data from multiple layers is pulled by RFlush and moved into lower layers. Using a data streaming model allows us to move away from traditional batch-based applications and provides us the ability to efficiently flush high volumes of data.

1.1 RFlush Overview

Figure 1 shows the architecture of RFlush. RFlush follows a pull-based client-server model . RFlush has four main components. The *Data Collector* is responsible for managing the pulling requests from the buffering system. It requests the location of the earliest entry and pulls starting from this location. The pulling is done in the order of arrival of messages. The request is then passed to the *Data Grouping*. This component is responsible for grouping requests into a unit called a data segment. The data grouping is then processed in parallel by leveraging the knowledge of the final destination given in the request to perform an efficient data distribution. All requests belonging to the same final destination are accepted into

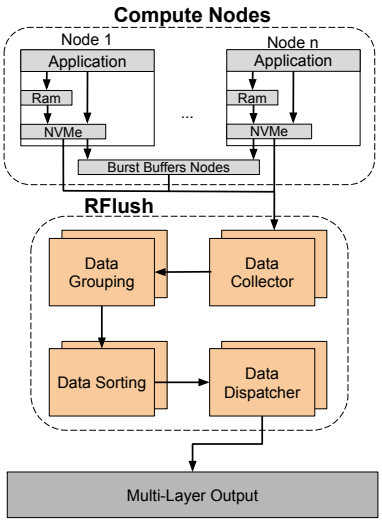


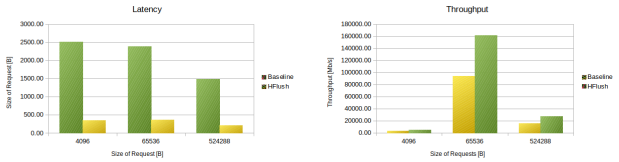
Figure 1: RFlush overview.

the same data segment managed by a specific process in the cluster. The data segment will keep accepting requests until one of two events gets triggered, either the data segment reaches the maximum allowed size or the time spent since the creation of the segments exceeds a user-defined threshold. Once the data segment is closed, the segments is sent to a third component in the pipeline, the *Data Sorter*. This component checks that each data segment is distributed to different process where they are returned to a sorted state that could have been lost due to network issues or data movements with unequal speeds. Finally, the *Data Dispatcher* is the component responsible for writing segments into the corresponding user-defined data object (e.g. a file document, a graph, an object, etc) at the final layer of storage.

As such, the main idea behind RFlush is to flush a number of requests buffered by an intermediate platform, group them into a sorted data segment, and finally, send the data segment to a specific layer. Since all of these processes are implemented under a data streaming model, the data flushing becomes a continuous, real-time operation. Even though RFlush requires some internal buffering, it uses far less buffering compared to previous data flushing methods. Also, by grouping the requests into file segments, RFlush provides higher data throughput and lower latency in situations with a high volume of data production.

1.2 Initial Results

All experiments were conducted on the Ares supercomputer at the Illinois Institute of Technology. Each compute node has a dual Intel(R) Xeon Scalable Silver 4114 @ 2.20GHz (i.e. 40 cores per node), 96 GB RAM, 10Gbit Ethernet with RoCE, and a local 512GB NVMe SSD. Each storage node has two quad-core Opteron 2376 @ 2.3GHz (i.e. 40 cores per node), 32GB DDR2-667 memory, one 250GB Samsung 860 Evo SATA SSD, and one 1TB Seagate 7200K SATA hard drive. The operating system of the cluster is CentOS 7.5.



(a) Latency (b) Throughput

Figure 2: RFlush Results.

For our tests, we used four compute nodes and four storage nodes. The compute nodes emulated a functional application generating request of a constant, user-defined size. The storage nodes where in charge of RFlush managing 4 different processes per node. We compared RFlush with a generic batch-based buffering platform. We ran three experiments using 10 thousand, 100 thousand, and 1 million requests respectively. Each experiment used three different request sizes: 1 KB, 16 KB, and 64 KB. We average the data latency of each of the requests to obtain the average latency of the system and calculate the overall bandwidth of the system to obtain the final results. As we see in figure [**?**], the baseline batch solution shows an average latency of 2000ms across all message sizes while RFlush shows an average latency of 300ms, obtaining a 7X reduction. With respect to bandwidth, RFlush presents a 2X increase over the batch-based solution.

1.3 Conclusion

We have introduced RFLush, a pull-based data flusher that implements a near real-time data eviction policy over multi-tiered storage environments. Initial results have shown a promising solution to a growing problem of evicting data in multi-layered environments especially in environments with extreme scale data generation. The pull-based implementation of RFlush allows a significant reduction on resource dedicated to I/O on the client nodes, while the near real-time nature of the eviction allows for a lower dependency on the inherent performance of the different layers and an improved overall latency on the data flushing with 7X latency reduction and a 2X bandwith increase over batch-based flushing solutions.

REFERENCES

[1] Bin Dong, Suren Byna, Kesheng Wu, Hans Johansen, Jeffrey N Johnson, Noel Keen, et al. 2016. Data elevator: Low-contention data movement in hierarchical storage system. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, Hyderabad, India, 152–161.

[2] Yang Wang Xian-He Sun Chuanhe Huang He, Shuibing and Chenzhong Xu. 2017. Heterogeneity-Aware Collective I/O for Parallel I/O Systems with Hybrid HDD/SSD Servers. *Transactions on Computers* 66, 6 (2017), 1091–1098.

[3] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, USA, 219–230.

[4] John Shalf, Sudip Dosanjh, and John Morrison. 2010. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*. Springer, USA, 1–25.

[5] Teng Wang, Suren Byna, Bin Dong, and Houjun Tang. 2018. UniviStor: Integrated Hierarchical and Distributed Storage for HPC. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, USA, 134–144.