OctopusFS: A Distributed File System with Tiered Storage Management

Elena Kakoulli
Cyprus University of Technology
30 Archbishop Kyprianou Str.
3036 Limassol, Cyprus
elena.kakoulli@cut.ac.cy

Herodotos Herodotou
Cyprus University of Technology
30 Archbishop Kyprianou Str.
3036 Limassol, Cyprus
herodotos.herodotou@cut.ac.cy

ABSTRACT

The ever-growing data storage and I/O demands of modern large-scale data analytics are challenging the current distributed storage systems. A promising trend is to exploit the recent improvements in memory, storage media, and networks for sustaining high performance and low cost. While past work explores using memory or SSDs as local storage or combine local with network-attached storage in cluster computing, this work focuses on managing multiple storage tiers in a distributed setting. We present OctopusFS, a novel distributed file system that is aware of heterogeneous storage media (e.g., memory, SSDs, HDDs, NAS) with different capacities and performance characteristics. The system offers a variety of pluggable policies for automating data management across the storage tiers and cluster nodes. The policies employ multi-objective optimization techniques for making intelligent data management decisions based on the requirements of fault tolerance, data and load balancing, and throughput maximization. At the same time, the storage media are explicitly exposed to users and applications, allowing them to choose the distribution and placement of replicas in the cluster based on their own performance and fault tolerance requirements. Our extensive evaluation shows the immediate benefits of using OctopusFS with data-intensive processing systems, such as Hadoop and Spark, in terms of both increased performance and better cluster utilization.

Keywords

distributed file system; tiered storage management

1. INTRODUCTION

The need for improvements in productivity and decision making processes for enterprises has led to considerable innovation in systems for large-scale data analytics. Parallel databases dating back to 1980s have added techniques like columnar data storage and processing [18], while new distributed platforms such as MapReduce [5] and Spark [37] have been developed. Other innovations aimed at creating alternative platforms for more generalized *dataflow* applications, including Dryad [14] and Stratosphere [2]. All aforementioned systems share two common aspects: (1) they run on large *clusters of commodity hardware* in a shared-nothing architecture and (2) they process data often residing in *distributed file systems* like HDFS [30], GFS [8], and QFS [27].

Commodity machines in these clusters have seen significant improvements in terms of memory, storage media, and networking. Memory capacities are constantly increasing, which is leading to the introduction of new in-memory data processing systems (e.g., Spark [37], RAMCloud [26]). On the storage front, flash-based solid state drives (SSDs) offer low access latency and low energy consumption with larger capacities [19]. However, the high price per gigabyte of SSDs makes hard disk drives (HDDs) the predominant storage media in datacenters today [21]. Finally, network-attached storage has also been coupled with direct-attached storage in cluster environments for improving data management [24, 17]. This *heterogeneity* of available storage media with different capacities and performance characteristics must be taken into consideration while designing the next generation of distributed storage and processing systems.

At the same time, modern applications exhibit a variety of I/O patterns: batch processing applications (e.g., MapReduce [5]) care about raw sequential throughput, interactive query processing (e.g., via Hive [33]) benefits from lower latency storage media, whereas other applications (e.g., HBase [7]) make use of random I/O patterns. Hence, it is desirable to have a variety of storage media and let each application choose (or automatically choose for it) the ones that best fit its performance, cost, and durability requirements.

Recent work takes advantage of the increase in memory sizes and targets on improving local data access in distributed applications by using memory caching, storing data directly in memory, or using re-computation through lineage [3, 11, 21, 37]. SSDs have also been used recently as the storage layer for distributed systems, such as key-value stores [6, 25] and MapReduce systems [15, 20]. Finally, [9, 24] focus on improving data retrieval from remote enterprise or cloud storage systems to local computing clusters by utilizing on-disk caching at compute nodes for persistent data.

Whereas previous work explores using memory or SSDs for (or as a cache for) local storage, or combines local with remote storage, our work focuses on managing multiple storage tiers together in a distributed setting. In this paper, we present a novel design for a *multi-tier distributed file system*, called *OctopusFS*, that utilizes multiple storage media (e.g.,

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064023

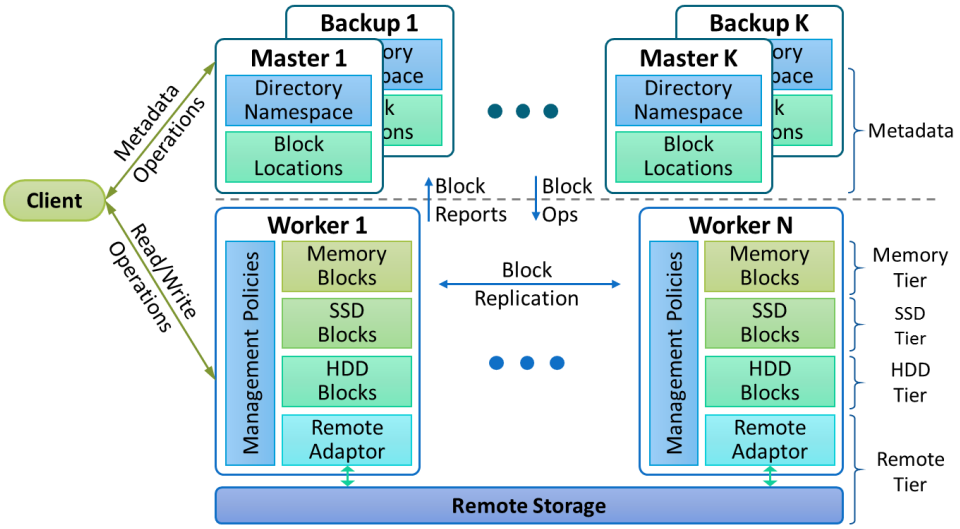

Figure 1: OctopusFS architecture configured with four storage tiers.

memory, SSDs, HDDs, remote storage) with different performance and capacity characteristics.

Our design focuses on two antagonistic system capabilities: *controllability* and *automatability*. On one hand, the heterogeneous storage media are explicitly exposed to users and applications, allowing them to choose the distribution and placement of replicas in the cluster based on their performance and fault tolerance requirements. On the other hand, OctopusFS offers a variety of pluggable policies for automating data management with the dual goal of increasing performance throughput while improving cluster utilization.

Higher-level processing systems can take advantage of the unique controllability features of OctopusFS to improve their efficiency and effectiveness in analyzing large-scale data. We believe this work will open new research directions for improving the functionality of various distributed systems, such as the task scheduling algorithms of MapReduce, the query processing of Pig and Hive, the workload scheduling of Oozie and Spark, and many others (discussed in Section 6).

Automatability features are equally, if not more, important as they simplify and automate data management across storage tiers and cluster nodes, alleviating the need for any manual user intervention. Towards this end, we formulate the problem of making smart data placement and retrieval decisions as a *multi-objective optimization problem* that takes into consideration the trade-offs between the conflicting requirements of data and load balancing, fault tolerance, and throughput maximization. At the same time, OctopusFS offers the de facto features of fault tolerance, scalability, and high availability. Finally, in order to support multitenancy, the system also offers security measures and quota mechanisms per storage media to allow for a fair allocation of limited resources (like memory and SSDs) across users.

In summary, the key contributions of this work are:

1. **The design of OctopusFS**, a novel distributed file system with tiered storage management capabilities
2. **Simple file system API extensions** that provide visibility and controllability of data management
3. **Automated data management policies** based on a multi-objective optimization problem formulation

Our high-level design is inspired by other popular distributed file systems such as GFS and HDFS. Thus, we have chosen to use HDFS as the basis for our implementation and have made it backwards compatible with it. Hence, OctopusFS can be used as a drop-in replacement of HDFS, one of the most widely used file systems in cluster deployments [30]. Our extensive evaluation offers an in-depth study of the OctopusFS features and showcases the immediate benefits of using OctopusFS with (unmodified) data-intensive processing systems, such as Hadoop and Spark, in terms of both increased performance and better cluster utilization.

The rest of the paper is organized as follows. Section 2 presents the overall architecture and API extensions of OctopusFS. Sections 3, 4, and 5 discuss the data placement, retrieval, and replication operations, respectively. Section 6 outlines some key enabling use cases. The experimental evaluation is presented in Section 7, while Section 8 discusses related work and Section 9 concludes the paper.

2. SYSTEM ARCHITECTURE

OctopusFS enables scalable and efficient data storage on compute clusters by utilizing directly-attached HDDs, SSDs, and memory, as well as remote (network-attached or cloud) storage. It is designed to store and retrieve *files*, whose data will be striped across nodes and replicated for fault tolerance. OctopusFS employs a *multi-master/slave* architecture similar to HDFS, shown in Figure 1, that consists of:

- Multiple (*Primary*) *Masters* that manage the directory namespace and regulate access to files
- Multiple *Backup Masters* that maintain an up-to-date image of the namespace and create checkpoints
- Multiple *Workers* that store the data and manage the heterogeneous storage media attached to each node
- A *Client* that exposes an enhanced file system API and allows users/applications to interact with the system

The operational goals of OctopusFS are twofold: (1) ensure efficient and effective utilization of the heterogeneous storage

media present in the cluster, and (2) preserve the scalability and performance benefits of compute-data co-location. Towards this, both Masters and Workers are aware of the different storage media, while the Client exposes locality and storage-media information to the users.

2.1 Primary and Backup Masters

Each Primary Master is responsible for maintaining two metadata collections, the *directory namespace* and the *block locations*. The directory namespace offers a traditional hierarchical file organization as well as typical operations like opening, closing, deleting, and renaming files and directories. The file content is split into large blocks (128MB by default) and each block is independently replicated at multiple Workers [30]. The Master also maintains the mapping of file blocks to Workers and storage media. In order to scale the name service horizontally, multiple Masters are used to form a *federation* [28] and are independent from each other.

Each Primary Master can have a Backup Master for increased fault tolerance and availability. The Backup is responsible for periodically creating and persisting a checkpoint of the namespace metadata so that the system can start from the most recent checkpoint upon a Master's failure. It also maintains an in-memory up-to-date image of the namespace and is standing by to take over in case the Master fails [30].

2.2 Workers

The Workers are typically run one per node in the cluster and are responsible for (i) storing and managing the file blocks on the various storage media, (ii) serving read and write requests from the file system's Client, and (iii) performing block creation, deletion, and replication upon instructions from the Masters, in a similar way as HDFS [30].

Each Worker is configured to use all available storage media in the node it is running on. The same type of storage media (with similar I/O characteristics) across all Workers are logically grouped into a virtual *storage tier*. Equivalently, a storage tier (e.g., the "SSD" tier) will encompass all Workers in a cluster that are associated with the same storage media type (e.g., SSDs). If some nodes in the cluster do not have SSDs, they will not be included in the "SSD" tier. It is also possible for some nodes to have different types of SSDs, for example PCIe and SATA SSDs, with very different performance characteristics. In this case, the system can be configured to use two distinct tiers for them (e.g., "SSD-1" and "SSD-2"). This design allows OctopusFS to (i) distinguish the different storage tiers based on performance rather than physical storage type, and hence, utilize them more efficiently; and (ii) achieve extensibility as new types of storage media like non-volatile RAM (NVRAM) and phase-commit memory (PCM) can be readily added as new storage tiers, even on an existing OctopusFS instance.

The file blocks can be stored and replicated in one or more storage tiers, based on requests from the Client or pluggable management policies. For example, consider the cluster shown in Figure 1 that shows 4 tiers, namely "Memory", "SSD", "HDD", and "Remote". A block may have 3 replicas on the "SSD" tier (on 3 different nodes); or it may have 1 replica on each of the "Memory", "SSD", and "HDD" tier (on 1, 2, or 3 different nodes); or any other combination. Hence, users and applications have tremendous flexibility on how to place and move data in the file system. Sections 3–5 discuss the relevant operations and policies in detail.

2.3 Client

A user or application interacts with OctopusFS through the Client. The Client exposes APIs for all typical file system operations like creating and deleting directories, and reading, writing, and deleting files. Table 1 lists the minimal API extensions that were introduced in the Apache Commons FileSystem API v2.7.0 [12] for enabling tiered storage. In particular, the management of files with respect to the storage tiers is achieved through a *replication vector* that specifies the number of replicas for each storage tier. For example, the replication vector $V = \langle M, S, H, R, U\rangle = \langle 1, 0, 2, 0, 0\rangle$[1] for a file F indicates that F has 1 replica in the "Memory" tier and 2 replicas in the "HDD" tier.

The special entry "U" (which stands for *"Unspecified"*) in the replication vector indicates the number of replicas to be placed on *any* storage tier. OctopusFS is responsible for selecting the actual tiers using the pluggable data placement policies, discussed in Section 3. Therefore, the replication vector mechanism enables the full spectrum of choices between controllability and automatability. An application can either (i) explicitly select all storage tiers for a file (and let $U = 0$); (ii) only set the total number of replicas through U and let the system decide the tiers; or (iii) select some tiers and also provide a number of unspecified replicas in the replication vector (e.g., $V = \langle M, S, H, R, U\rangle = \langle 0, 1, 0, 0, 2\rangle$). Backwards compatibility is also straightforward to achieve by simply replacing the single replication factor r provided in the old API with a replication vector where $U = r$. Finally, a replication vector is encoded into 64 bits and, hence, it is very efficient to use and store.

V can be specified during file creation with the **create** API and can be modified with the **setReplication** API (see Table 1) to achieve various functionalities:

- **Move a file between tiers.** For example, changing $\langle 1, 0, 2, 0, 0\rangle$ to $\langle 1, 1, 1, 0, 0\rangle$ will result in moving 1 replica from the "HDD" tier to the "SSD" tier.
- **Copy a file between tiers.** For example, changing $\langle 1, 0, 2, 0, 0\rangle$ to $\langle 1, 1, 2, 0, 0\rangle$ will result in copying 1 replica to the "SSD" tier. This also increases the overall number of replicas from 3 to 4.
- **Modify the number of replicas within a tier.** For example, changing $\langle 1, 0, 2, 0, 0\rangle$ to $\langle 1, 0, 3, 0, 0\rangle$ will result in increasing the number of HDD replicas from 2 to 3 (and the overall from 3 to 4).
- **Delete a file from a tier.** For example, changing $\langle 1, 0, 2, 0, 0\rangle$ to $\langle 0, 0, 2, 0, 0\rangle$ will result in deleting the in-memory replica. This also decreases the overall number of replicas from 3 to 2.

Each time the replication vector of a file changes, a *network-aware* and *tier-aware* placement policy is invoked for deciding where the addition or deletion of a replica will take place (discussed in Section 3). Finally, the Client exposes (i) the Workers and storage tiers of the block replicas through the **getFileBlockLocations** API and (ii) the active storage tiers in the system along with useful information per tier (e.g., total/remaining capacity, read/write throughput, etc.)

[1] $\langle M, S, H, R, U\rangle$ is a shorthand notation for the tiers ⟨"Memory", "SSD", "HDD", "Remote", "Unspecified"⟩

Table 1: OctopusFS API extensions to the Apache Commons FileSystem API v2.7.0.

API	Comments
FSDataOutputStream **create**(Path f, ReplicationVector repVector, long blockSize)	Creates a file and returns an output stream for writing. The original API uses "short replication" instead of "ReplicationVector repVector".
boolean **setReplication**(Path f, ReplicationVector repVector)	Sets the replication vector for an existing file. The original API uses "short replication" instead of "ReplicationVector repVector"
BlockLocation[] **getFileBlockLocations**(Path f, long start, long len)	Returns the list of block locations containing the data in the requested byte range. Each block location indicates the storage tier.
StorageTierReport[] **getStorageTierReports**()	Returns the list of active storage tiers along with useful information per tier (e.g., total/remaining capacity, read/write throughput).

through the **getStorageTierReports** API. These APIs allow applications to make fully informed decisions for which replica to read from and how to specify replication vectors for files, respectively, in order to improve I/O performance.

2.4 Remote Storage

The remote storage can have the form of another distributed file system running in a different cluster (e.g., HDFS, MapR), a cloud-based storage system (e.g., Amazon's S3, Azure Blob Storage), or network-attached storage. Remote storage can be attached to OctopusFS in two modes: *integrated* and *stand-alone*. In the integrated mode, the remote storage is treated like any other storage media in the cluster and the Workers use it for writing and reading file blocks.

In the stand-alone mode, the remote storage is considered as an independent entity that stores files and it is used as a virtual extension to OctopusFS mounted at a particular directory. As such, the directory namespace is appended with information from the remote storage and provides a unified view and access methods to all data. Applications can then use any of the Workers (through the Client) for reading and writing file blocks. The stand-alone mode is a generalized concept of the idea introduced in *MixApart* [24], which utilizes on-disk caching at compute nodes for persistent data stored in data warehouses. Hence, it is not elaborated further in this paper.

3. DATA PLACEMENT

The awareness of storage media with different performance characteristics adds a significant level of complexity to the main file operations of the system and creates the need for heterogeneous-aware placement and retrieval policies for improving the I/O performance of the cluster. This section focuses on the data-placement aspects of the system.

3.1 File Write Operations

An application adds data to OctopusFS by creating a new file and writing the data to it using the Client. At file creation, the Client can optionally specify a block size as well as a replication vector (recall Section 2.3). The Client then writes the data one block at a time. Upon a block creation, the Client first contacts the Master and obtains a list of locations (i.e., storage media, each belonging to a particular Worker and storage tier) that will host the replicas of that block. This list is determined using a *pluggable block placement policy*, described in Section 3.3.

Next, the Client organizes a Worker-to-Worker pipeline and sends the data. For example, suppose the block locations for writing are $[\langle W1, M\rangle, \langle W3, H\rangle, \langle W6, H\rangle]$. Here, the data is pipelined from the Client to the "Memory" tier in Worker $W1$ to the "HDD" tier in $W3$ to the "HDD" tier in $W6$. When a block is filled, the Client requests new locations to host the replicas of the next block. A new pipeline is organized and the process repeats.

3.2 Data Placement Modeling

Formally, the data placement problem can be defined as:

> Given all available storage media (belonging to Workers $W_1..W_n$ and tiers $T_1..T_k$) and a number of replicas r to place in the cluster, the placement policy must select a list $\overrightarrow{m} = (m_1, ..., m_r)$, $m_i \neq m_j \forall i, j$, of r media for hosting those replicas.

The above formulation deals with the general case of the problem in which the policy must decide both the Workers and the storage tiers. The scenario where the user specifies required tiers in the replication vector, either partially or fully, is a special case discussed at the end of this section.

The block placement policy is tasked with making a critical decision that can have significant impact on the overall data reliability, availability, and system performance, while taking into consideration the Workers, the storage tiers, and various other parameters. Worker nodes are typically spread across multiple racks creating a *hierarchical network topology* [30]. It is important to take this topology into account in order to improve fault tolerance and availability (e.g., by placing replicas across racks). At the same time, the presence of multiple tiers with different I/O characteristics can be utilized to significantly improve the I/O throughput of the system (e.g., by placing more replicas in higher tiers).

Two additional decision criteria are data and load balancing across both the Workers and the storage tiers. In distributed storage systems, there is typically a trade-off between the two aforementioned criteria. Placing a replica on a storage media with plenty of capacity left might be detrimental to overall performance if that media is already serving several concurrent I/O requests. Similarly, placing data on a currently idle storage media may hurt data balance, especially in the presence of long sequential writes. As both the data and load balancing are critical performance factors that can hurt each other, a policy must carefully consider the trade-offs in order to achieve high system throughput.

In total, we have identified four objectives that need to be optimized simultaneously: *data balancing*, *load balancing*, *fault tolerance*, and *throughput maximization*. The trade-offs and intricacies of these objectives have motivated our decision to formulate the placement problem as a *multi-objective optimization problem* (*MOOP*). For each objective that we

need to maximize, we define an objective function and an upper-bound function associated with the optimal solution.

Data balancing objective: The goal of data balancing is to ensure the even distribution of data across the available storage media in the cluster. For each media m_i, the system knows the Worker it is on ($Worker[m_i]$), its storage tier ($Tier[m_i]$), as well as its remaining capacity ($Rem[m_i]$) and total capacity ($Cap[m_i]$). The usage statistics are maintained at each Worker and are frequently reported to the Master during heartbeat operations.

To achieve data balancing, each replica to be stored next should be placed to the storage media with the most remaining capacity percentage, after accounting for the block size to be stored. We consider remaining rather than used capacity to account for the fact that storage media might contain other local data not in the control of our file system, such as log and temporary data of other systems running on the cluster. In addition, we are using a percentage to normalize across storage media with different total capacities.

The data balancing objective function f_{db} is defined as:

$$f_{db}(\overrightarrow{m}) = \sum_{m_i \in \overrightarrow{m}} \frac{Rem[m_i] - blockSize}{Cap[m_i]} \tag{1}$$

Given a list of selected storage media $\overrightarrow{m}$, f_{db} computes the sum of the remaining capacity percent of each selected media, and hence, it is maximized when the r media with the highest remaining percent are selected. The theoretical upper bound function value of f_{db} of Pareto optimal solutions occurs when the media with the highest remaining percent is always selected. Hence, the ideal function f^*_{db} of data balancing is defined as:

$$f^*_{db}(\overrightarrow{m}) = |\overrightarrow{m}| \times \max_{\forall m} \frac{Rem[m]}{Cap[m]} \tag{2}$$

Load balancing objective: The goal of load balancing is to efficiently distribute the I/O requests across all available storage media in the cluster. Each Worker is responsible for maintaining the number of active I/O connections ($NrConn[m_i]$) to each media m_i, which are frequently reported to the Master during heartbeat operations.

The number of connections to a media is inversely proportional to the throughput rate that can be achieved from each connected reader or writer. Hence, to achieve load balancing, each replica to be stored next should be directed to the storage media with the lowest number of active connections.

The load balancing objective function f_{lb} is defined as:

$$f_{lb}(\overrightarrow{m}) = \sum_{m_i \in \overrightarrow{m}} \frac{1}{NrConn[m_i] + 1} \tag{3}$$

f_{lb} is maximized when the r media with the lowest number of connections are selected. The theoretical upper bound function value of f_{lb} of Pareto optimal solutions occurs when the media with the lowest number of connections is always selected. Hence, the ideal function f^*_{lb} of load balancing is:

$$f^*_{lb}(\overrightarrow{m}) = |\overrightarrow{m}| \times \frac{1}{\min_{\forall m}(NrConn[m] + 1)} \tag{4}$$

Fault tolerance objective: Fault tolerance in our setting refers to the ability of the file system to avoid any data loss (to the extend possible). To meet the fault-tolerance requirement, multiple replicas of a block should be stored on different storage tiers on different Workers on different racks. Given the disparate failure rates of storage media (e.g., of SSDs vs. HDDs or even of low-end vs. high-end HDDs), placing replicas on different tiers is a wise decision. Storing replicas across different Workers will obviously protect against node or media failures, while storing them across racks will help in the case of switch or power failures that could take down an entire rack. However, correlating failures that simultaneously disconnect multiple racks from the cluster are very rare [30]. Hence, storing replicas in more than 2 racks offers almost no additional fault tolerance benefits while lowering the write I/O performance.

The fault tolerance objective function f_{ft} is defined as:

$$\begin{aligned} f_{ft}(\overrightarrow{m}) = & \frac{NrTiers[\overrightarrow{m}]}{\min(|\overrightarrow{m}|, k)} + \frac{NrNodes[\overrightarrow{m}]}{\min(|\overrightarrow{m}|, n)} \\ & + \left(t = 1 \ ? \ 1 : \frac{1}{|NrRacks[\overrightarrow{m}] - 2| + 1}\right) \end{aligned} \tag{5}$$

The numbers k, n, and t refer to the total number of storage tiers, nodes, and racks in the cluster, respectively, while $NrTiers[\overrightarrow{m}]$, $NrNodes[\overrightarrow{m}]$, and $NrRacks[\overrightarrow{m}]$ respectively compute the distinct number of tiers, nodes, and racks that appear in the storage media list $\overrightarrow{m}$.

f_{ft} is maximized when the r media are located all on different tiers, on different racks, and on 2 distinct racks (if multiple racks are present). The upper bound function value of f_{lb} is maximized when each one of the ratios in the above equation equals one, and hence, the ideal function f^*_{ft} of fault tolerance is simply a constant:

$$f^*_{ft}(\overrightarrow{m}) = 3 \tag{6}$$

Throughput maximization objective: The final objective seeks to optimize the overall I/O throughput of the file system by taking advantage of the fastest storage tiers present in the cluster. When a Worker is launched, it performs a short I/O-intensive test for measuring the sustained write and read throughputs of each storage media m_i, denoted by $WThru[m_i]$ and $RThru[m_i]$, respectively. The values are subsequently averaged per storage tier and reported to the Master.

To maximize throughput, each replica to be stored next should be placed to the storage media with the highest write throughput. In order to generate normalized values, we do not use the raw throughput values but rather the ratio of each media's throughput to the maximum one among all tiers. Further, we use the logarithm of those values in order to scale them down and decrease the large differences that may exist between some media such as memory and HDDs.

The throughput maximization objective function f_{tm} is:

$$f_{tm}(\overrightarrow{m}) = \sum_{m_i \in \overrightarrow{m}} \frac{\log(WThru[m_i])}{\log(\max_{\forall m} WThru[m])} \tag{7}$$

Given a list of selected storage media $\overrightarrow{m}$, f_{tm} computes the sum of the throughput ratios explained above, and hence, it is maximized when the r media with the highest throughput are selected. The theoretical upper bound function value of f_{tm} of Pareto optimal solutions occurs when the media with the highest throughput is always selected, for which all

Algorithm 1 Algorithm for solving an instance of MOOP

```
1: procedure SolveMoop(mediaOptions[], chosenMedia[])
2:     bestScore = ∞, bestMedia = null
3:     for each option in mediaOptions do
4:         chosenMedia.add(option)
5:         score = ‖f(chosenMedia) − z*(chosenMedia)‖
6:         if score < bestScore then
7:             bestScore = score
8:             bestMedia = option
9:         chosenMedia.remove(option)
10:    return bestMedia                    ▷ Best storage media
```

ratios become equal to 1. Hence, the ideal function f^*_{tm} of throughput maximization is defined as:

$$f^*_{tm}(\overrightarrow{m}) = |\overrightarrow{m}| \tag{8}$$

Multi-objective optimization problem: The four aforementioned objective functions are combined to define the vector-valued objective function $f : \overrightarrow{M} \rightarrow \mathbb{R}^4$ as:

$$f(\overrightarrow{m}) = \Big(f_{db}(\overrightarrow{m}), f_{lb}(\overrightarrow{m}), f_{ft}(\overrightarrow{m}), f_{tm}(\overrightarrow{m})\Big)^T \tag{9}$$

The set $\overrightarrow{M}$ represents the *feasible decision space*, that is the set of all r-length combinations of storage media that can be used for hosting r replicas for a file block. There are two constraints affecting the feasible set. First, all storage media in a potential solution $\overrightarrow{m} = (m_1, ..., m_r)$ must be unique, i.e., $m_i \neq m_j \forall i, j$, since no media should ever store the same block twice. Second, any storage media m_i can appear in a solution only when there is enough space to hold it, that is, the following constraint must hold: $Rem[m_i] - blockSize \geq 0$.

Given the conflicting nature of our objectives, there does not exist a feasible solution that maximizes all objective functions simultaneously. Hence, we focus on the Pareto optimal solutions, which are bounded by the *ideal objective vector* z^* defined as:

$$z^*(\overrightarrow{m}) = \Big(f^*_{db}(\overrightarrow{m}), f^*_{lb}(\overrightarrow{m}), f^*_{ft}(\overrightarrow{m}), f^*_{tm}(\overrightarrow{m})\Big)^T \tag{10}$$

Even though z^* corresponds to a non-existent solution, it denotes the vector with the theoretical upper bounds of all objective functions. Hence, we wish to find a solution that is as close as possible to the point of the ideal objective vector. In other words, we use the well-known method of *global criterion* [23] to formulate our MOOP as:

$$\min \|f(\overrightarrow{m}) - z^*(\overrightarrow{m})\|, s.t. \overrightarrow{m} \in \overrightarrow{M} \tag{11}$$

The above formulation belongs to the category of MOOP methods with no articulation of preferences. The main benefit is avoiding the use of weight parameters, whose settings typically fall to the hands of system admins with little knowledge on how to set them effectively. In addition, our objective functions are already normalized into a uniform, dimensionless scale, making this method ideal to use [23].

3.3 MOOP Data Placement Policy

OctopusFS provides a configurable block placement policy as well as a default one that offers a trade-off between minimizing the write cost and maximizing data reliability and read I/O performance. The default policy, called the *MOOP policy*, makes placement decisions based on the solution of the multi-objective optimization problem from Equation 11. There are several techniques for solving such problems [23] but they cannot be efficiently applied in our setting because our decision space $\overrightarrow{M}$ is combinatorial in nature. Therefore, any algorithm attempting to enumerate $\overrightarrow{M}$ would yield $O(s^r)$ potential solutions (where s is the total number of storage media in the cluster and r is the number of requested replicas) while each function evaluation takes $O(r)$. This leads to an exponential running time of $O(r \times s^r)$.

Algorithm 2 MOOP data placement policy

```
1: procedure PlaceReplicas(client, repVector)
2:     chosen[] = ∅
3:     for each entry in repVector do
4:         options = GenOptions(client, chosen, entry)
5:         bestMedia = SolveMoop(options, chosen)
6:         chosen.add(bestMedia)
7:     return chosen                       ▷ Best list of chosen media
```

Instead, we have developed a *greedy algorithm* for finding a near-optimal solution to the MOOP. The key idea is to build the list of selected storage media $\overrightarrow{m} = (m_1, ..., m_r)$ incrementally by evaluating and selecting the best storage media for hosting one replica at a time. In other words, we first solve the MOOP formulation to find m_1; then we solve it again to find m_2; and so on. However, in each iteration (say to find m_i), we always take into account the previously selected media ($m_1..m_{i-1}$).

Algorithm 1 outlines our approach for finding the best storage media to host one replica, given a list of available media to choose from and the list of previously chosen media (line 1). For each available media m (line 3), we evaluate the scenario of adding m into the list of chosen media (line 4) and compute a score based on the MOOP formulation from Equation 11 (line 5). We keep track of the media with the lowest computed score (lines 6-8), which constitutes the final output of the algorithm. Given the algorithm iterates over the list of media options with size s and each function evaluation takes $O(r)$, the overall running time is $O(s \times r)$.

The above procedure could be invoked r consecutive times for selecting the r storage media to host a given block from the list of all available media. However, there are several heuristic techniques that can be used for safely pruning down the large space of storage media options as well as for navigating the search space more efficiently. As discussed earlier in Section 3.2, the best approach to achieving fault tolerance across racks is to distribute replicas on two racks. Hence, after selecting the first replica's location m_1, we can prune all storage options located on the same rack as m_1; and after selecting m_2, we can restrict the search space to only those two racks. In addition, the Client's location is also known to the system. If the Client is collocated with one of the Workers, it is best to consider storing the first replica on that Worker for better performance and predictability. Furthermore, the two constraints on the feasible decisions space can be handled with ease by simply removing from the list of options (i) all already chosen media and (ii) all media with remaining capacity less than the block size. Finally, if the user has provided a replication vector asking for replicas on specific tiers, those requests are used for pruning the space of options accordingly.

Algorithm 2 contains the data placement algorithm used by the MOOP policy. Given the location of the client and the requested replication vector (line 1), the algorithm will incrementally build the best list of storage media to host the replicas. In each iteration (lines 3-6) over the replication vector, the algorithm will first generate a list of available media based on the heuristics discussed above (line 4). Next, it will invoke Algorithm 1 to generate the best media for hosting the next replica (line 5) and add it to the list (line 6). As the loop will execute r times, the total running of our algorithm is $O(s \times r^2)$, a huge improvement over the exponential time $O(r \times s^r)$ explained before. In addition, given that r is typically very small (with the most popular value being 3), the algorithm is essentially linear with respect to the number of storage media, and hence, very efficient. Finally, since memory is volatile, its use is optional by the data placement policy and it is disabled by default. When enabled, it will not place more than 1/3 of the replicas in memory.

Similar to query optimizers in conventional database systems, the MOOP policy does not aim at finding the optimal placement but rather a good solution near the optimal one. This is achieved by exploiting the *optimal substructure property (OSP)* exhibited by each of the objective functions individually; i.e., the best r media for maximizing a particular objective function include the best $r-1$ media. For example, maximizing f_{db} is equivalent to finding the r media with the highest remaining capacity percentage, which include the $r-1$ media with the highest remaining capacity percentage. Even though the formulation of the problem in Equation 11 does not necessarily exhibit OSP, the selection of the r best media is based on the best $r-1$ media, while for $r = 1$, the algorithm returns the optimal solution. Hence, the MOOP policy is able to find a near-optimal solution.

4. DATA RETRIEVAL

This section describes the process by which users and applications access data stored in OctopusFS.

4.1 File Read Operations

When an application reads a file, the Client first contacts the Master for the list of locations (i.e., storage media, each belonging to a particular Worker and storage tier) that host replicas of the file blocks. The Master returns the list ordered based on a *pluggable data retrieval policy*, discussed below. The order is determined using the network location of the Client, the network topology of the Workers, as well as the storage tiers that host the replicas. The Client then contacts the first Worker directly and requests the transfer of the desired block. In case of a read failure, the Client contacts the next Worker on the list for reading the data.

4.2 Data Retrieval Policy

Formally, the data retrieval problem can be defined as:

> Given the list of storage media $(m_1, ..., m_r)$ belonging to some Workers and storage tiers as well as the location of the Client, the retrieval policy must provide an ordering $(m_{i_1}, ..., m_{i_r})$ for the Client to read from.

A data retrieval policy strives to maximize the read I/O efficiency of the file system. Similar to the data placement policy, the retrieval policy must take into consideration both the network location of the Workers as well as the tiers hosting the replicas. On one hand, the Client should read the replica from its nearest Worker in order to reduce inter-rack and inter-node traffic and generally improve the read performance. On the other hand, the Client should access the replica from the fastest tier for improving the I/O latency. However, the nearest replica may be on a slow tier whereas a replica on a faster tier is on a more distant node. For example, given a network transfer rate of 10 Gbps and a memory transfer rate of 40 Gbps, it might be more efficient to read an in-memory replica from a nearby node rather than reading a local HDD-based replica with a transfer rate of 2 Gbps.

There are two more important aspects to consider: the current network and storage media utilization. The available (network or media) bandwidth gets split among all connected readers and writers and can decrease significantly if the device gets overloaded. In the example above, if there are already 10 active network connections to the node, the expected transfer rate from it will be 1 Gbps not 10. In this case, a local read is probably the best option. The policy must be aware of the various trade-offs and aim for selecting the replica ordering that provides the highest overall I/O.

Our data retrieval policy implements a replica ordering algorithm that takes into consideration the following:

- the Client location (may be on or off the cluster)
- the replica locations (Worker and storage tiers)
- the network topology and the average data transfer rates from each Worker ($NetThru[W_j]$)
- the read throughput rate of each media ($RThru[m_i]$)
- the number of active network connections per Worker ($NrConn[W_j]$)
- the number of active I/O connections per storage media ($NrConn[m_i]$)

For each replica location (i.e., storage media m_i on Worker W_j), the policy calculates the potential rate of transferring the block data from there to the Client using the formula:

$$\min\left(\frac{NetThru[W_j]}{NrConn[W_j]}, \frac{RThru[m_i]}{NrConn[m_i]}\right) \tag{12}$$

Finally, it sorts the locations based on the decreasing calculated transfer rates. Locations with the same rates for which the network is the bottleneck are sorted only based on the storage media throughput rates. If those rates are the same as well, the corresponding locations are then shuffled randomly to help spread the load more evenly.

5. REPLICATION MANAGEMENT

One of the main purposes of the Master is to ensure that each block always has the intended number of replicas on each storage tier. The Master can detect the situations of under- or over-replication during the periodic block reports received from the Workers.

When a block becomes *under-replicated* (e.g., due to block corruption or Worker failure), the Master must select one (or more) Workers for hosting the new replica(s). Block replication follows a similar policy to the block placement one described in Section 3.3. In particular, it uses **SolveMoop**

(recall Algorithm 1) to select the best storage media for replacing the lost replica, given (i) the list of available storage media generated from **GenOptions**, and (ii) the existing replica locations. The Worker that is meant to host the new replica will utilize the data retrieval policy (described in Section 4.2) for copying from the most efficient location.

When a block becomes *over-replicated* on some particular tier, the Master must select a replica to remove. Block removal decisions are also based on the MOOP formulation from Section 3.2. Given the list of current replica locations $\overrightarrow{m} = (m_1, ..., m_r)$, we generate r lists of size $(r-1)$ by removing each time one location from $\overrightarrow{m}$. We compute the score of each option using Equation 11 and select the location whose removal from $\overrightarrow{m}$ let to the lowest score.

As discussed in Section 2.3, the Client offers an API for altering the replication vector of a file. This API can be used to increase or decrease the number of block replicas within a tier, as well as move or copy blocks from one tier to another. Similar to HDFS [30], these operations are asynchronous by design; that is, the Client will not wait until the copying or removal of blocks is completed. However, the Client will become aware of which locations were selected for future reference. Internally, the system will use the same policies for under- and over-replication of blocks described above.

6. ENABLING USE CASES

One of the most powerful features of OctopusFS is the *fine-grained control* it provides over the various storage media it manages. Applications can explicitly store data in different storage media and change the replication factors of blocks within and across tiers while taking into consideration their workload types (e.g., offline vs. interactive analytics), the expected I/O patterns (e.g., full vs. partial scans), and custom data layouts (e.g., row vs. column format). These capabilities can also provide significant benefits to large-scale analytics frameworks (e.g., MapReduce, Hive, Spark, Impala) in terms of manageability and performance as they can schedule their data processing jobs in both a location-aware and a storage-tier-aware manner. This section outlines a variety of application-centric and data-centric use cases enabled by our novel design that can inspire new research opportunities for data-intensive processing systems.

Multi-level cache management: OctopusFS is a general purpose file system that could be transformed into a *multi-level caching system* for improving I/O performance and cluster utilization. Cache management policies can be implemented both inside and outside the system, allowing applications the maximum possible flexibility on how to take advantage of OctopusFS. The Client, through the use of replication vectors, offers a rich enough API such that an entity that sits on top of OctopusFS can control the number and placement of replicas in the various storage tiers. In this scenario, an application can use its own knowledge and understanding of its workload to increase or decrease the replication factor per tier in order to maximize its performance or meet customer service-level agreements (SLAs). At the same time, OctopusFS offers pluggable policies for managing the storage resources as a cache internally.

MapReduce Task Scheduling: In Hadoop [35], the Job Scheduler is responsible for scheduling the Map and Reduce tasks to execute on the compute nodes of the cluster. Currently, the scheduling is done based on the block locations exposed by HDFS. With OctopusFS, the Job Scheduler can also *exploit the tiering information* of each block for making better scheduling decisions. Furthermore, since the Job Scheduler maintains the job queue and knows which tasks will execute next, it also knows which files will be accessed soon. The new APIs provided by OctopusFS allows for the Job Scheduler to implement a *prefetching mechanism* and instruct OctopusFS to start moving (or copying) block replicas to a higher storage tier. This approach better overlaps I/O with task processing and increases cluster utilization.

Workload scheduling: Analytical workloads are typically expressed as directed acyclic graphs of jobs [35, 37]. In such workloads, the output data from one job becomes the input to the following job(s), and hence, smart *intermediate data placement* can have great benefits to the overall workload execution time. OctopusFS provides the flexibility of placing the intermediate data in local memory or SSDs in order to speed up the overall processing. In addition, the workload processing system has intricate knowledge of any *common data* among jobs or workloads and can utilize OctopusFS for dynamically moving data up and down the storage tiers.

Interactive analytics: Apart from the typical batch oriented analytics, *interactive* data exploration is becoming increasingly important. In addition, more complex *iterative* algorithms for machine learning and graph processing are becoming popular [37]. A common attribute for these types of applications is the need to share data across multiple analysis steps (e.g., multiple queries from the user, or multiple steps of an iterative computation). By allowing explicit memory management, OctopusFS empowers applications to pin their working sets in cluster memory. Fault tolerance is provided by keeping multiple replicas in memory or creating extra replicas on persistent storage. Recent work on lineage (e.g., [37, 21]) is complementary to our work and could be used for achieving fault tolerance without replication.

Table 2: Average write and read throughput (MB/s) per storage media in the cluster.

Storage Media	Write Throughput	Read Throughput
Memory	1897.4	3224.8
SSD	340.6	419.5
HDD	126.3	177.1

7. EXPERIMENTAL EVALUATION

The experimental setup used is a 10-node cluster running CentOS Linux 7.2 with 1 Master and 9 Workers. The Master node has a 64-bit, 8-core, 3.2GHz CPU, 64GB RAM, and a 2.1TB RAID 5 storage configuration. Each Worker node has a 64-bit, 8-core, 2.4GHz CPU, 24GB RAM, one 120GB SATA SSD, and three 500GB SAS HDDs. OctopusFS is configured to use 4GB, 64GB, and 400GB of memory, SSD, and HDD space, respectively, for storing blocks on each Worker node. Table 2 lists the average write and read throughput for each storage media type as measured by the Workers. Our evaluation methodology is as follows:

1. We study the effect of tiered storage in a cluster.
2. We evaluate the data placement policy along with the four optimization objectives and compare it against the original HDFS and a rule-based policy.

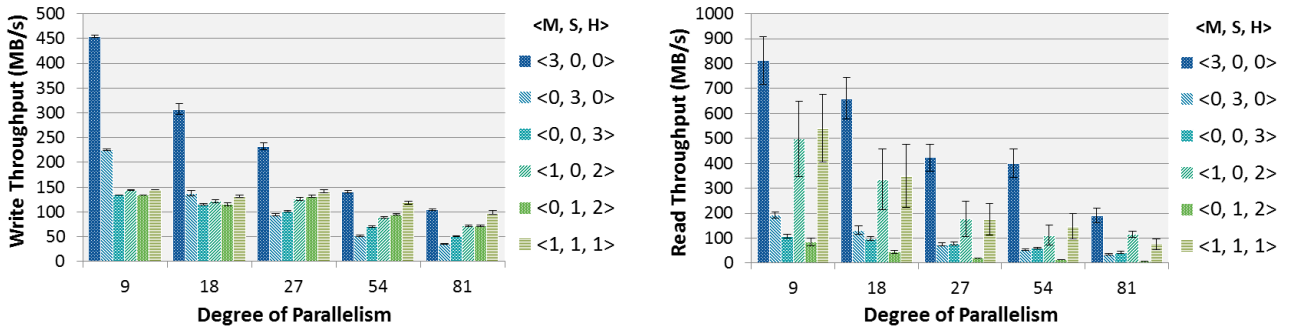

Figure 2: (a) Average write and (b) average read throughput per Worker for five degrees of parallelism and six replication vectors ($\langle M, S, H\rangle$ = ⟨"Memory", "SSD", "HDD"⟩).

3. We evaluate the data retrieval policy.
4. We study the efficiency of namespace operations.
5. We compare the use of OctopusFS against HDFS in managing data for Hadoop and Spark workloads.
6. We study the effectiveness of our controllability features with Pegasus [16], a graph mining system.

For our evaluation we used three well-known benchmarks: (i) **DFSIO** [30], a distributed I/O benchmark that measures average throughput for write and read operations; (ii) **S-Live Test** [31], which is used for stress testing the namespace operations of a distributed file system; and (iii) **HiBench** [13], a benchmark suite consisting of both synthetic micro-benchmarks and real-world applications executed on both Hadoop MapReduce [35] and Spark [37].

7.1 Effect of Tiered Storage

This section studies the effect of storing block replicas on multiple storage tiers. We used the DFSIO benchmark to write 10GB of data with a total replication factor of 3 (i.e., 30GB of data is stored on OctopusFS) and then read it while measuring the system performance. We controlled the placement of replicas to tiers by explicitly specifying the replication vector during file creation. Since the I/O bandwidth of any storage media is sensitive to the number of concurrent writers/readers, we repeated the experiments multiple times while varying the number of parallel writers/readers.

The average write and read throughput per Worker node are shown in Figures 2(a) and 2(b), respectively. We focus on the average throughput per node because the total bandwidth is linear with the number of nodes [30]. We use three replication vectors for testing storing all 3 replicas on the same tier and three for storing them on multiple tiers. As expected, storing all replicas in memory yields the highest performance, which decreases as we increase the degree of parallelism d due to network congestion. Interestingly, storing all replicas in the "SSD" tier is better compared to the "HDD" tier only for small d, while it gets worse as we increase d. This is explained by the facts that (i) our SSDs are only ~2.6x faster than our HDDs and (ii) we have 3 HDDs per node. For example, when $d = 27$, 3 blocks (on average) are concurrently stored on each node. For $V = \langle M, S, H\rangle = \langle 0, 3, 0\rangle$, all 3 blocks are placed on the SSD while for $V = \langle 0, 0, 3\rangle$ the 3 blocks are distributed across the 3 HDDs. Another interesting observation is that placing replicas on multiple tiers does not have any effect for small d since the data are written in a pipeline and the performance is bottlenecked by storing at least 1 replica on HDD. However, as d increases, the network bandwidth per thread decreases more than per media, so the benefits of using multiple tiers (and media) start to show and can lead to up to 2x performance improvement compared to storing all replicas on HDDs.

The read throughput patterns for the first three replication vectors are very similar to the write throughput ones analyzed above. One unique observation here is that by placing just 1 replica in memory, the average read throughput increases 2–5x over storing all replicas on HDDs. At the same time, the error bars (showing the standard error of the mean) indicate a large variation in the read throughput rate because some reads are local (i.e., the Client is located on the same machine as the node hosting a replica to read) while others are non-local. Note that the way this experiment was performed yielded only about 1/3 local reads. Hadoop and Spark are known to typically achieve about 90% locality rates, which would yield even larger performance gains [13].

7.2 Data Placement Evaluation

The purpose of this section is threefold. First, in order to study the individual effects of each one of the optimization objectives introduced in Section 3.2, we implemented and tested four placement policies, one for each objective. Second, we evaluate the benefits of the *MOOP* policy (recall Section 3.3) compared to the original HDFS placement policy, both in terms of write and read I/O performance. In order to further analyze the effect of SSDs, we configure the original HDFS with two settings: (a) HDFS using only HDDs for storing replicas (*"Original HDFS"*); and (b) HDFS using both HDDs and SSDs for storing replicas (*"HDFS with SSD"*) but without differentiating between the two storage media types (since it is not capable of doing so). Finally, we compare the *MOOP* policy against a *Rule-based* policy in order to validate the need for a model-based approach. The *Rule-based* policy takes both network topology and storage tiers into account by placing replicas across the tiers in a round robin fashion on randomly-selected nodes across two racks. For these experiments, we run the DFSIO benchmark with degree of parallelism 27 for writing and reading 40GB of data with a replication vector of $U = 3$.

Figure 3(a) shows the average write throughput per Worker for the 8 data placement policies during the generation of the 40GB of data, while Figure 4 shows remaining capacity percent per storage tier. The Throughput Maximiza-

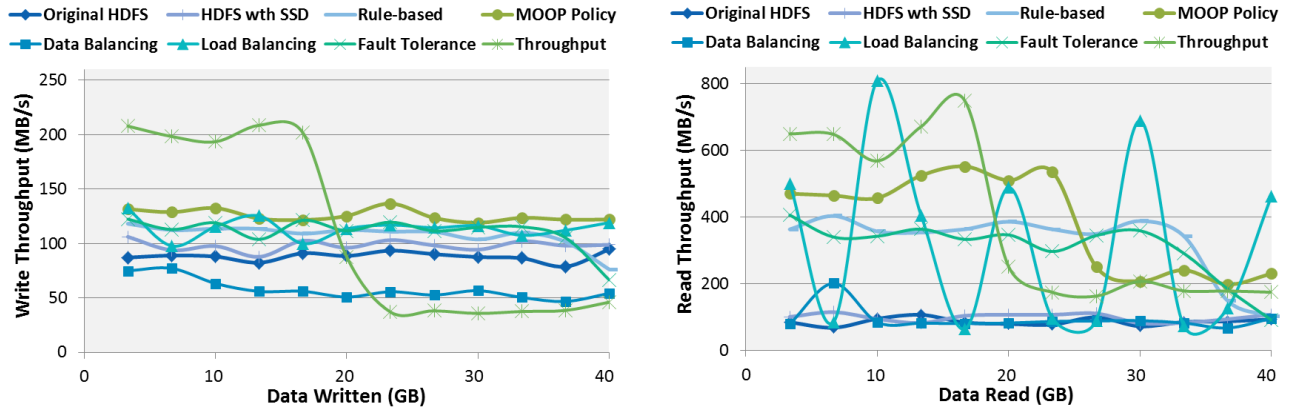

Figure 3: (a) Average write and (b) average read throughput per Worker for eight data placement policies.

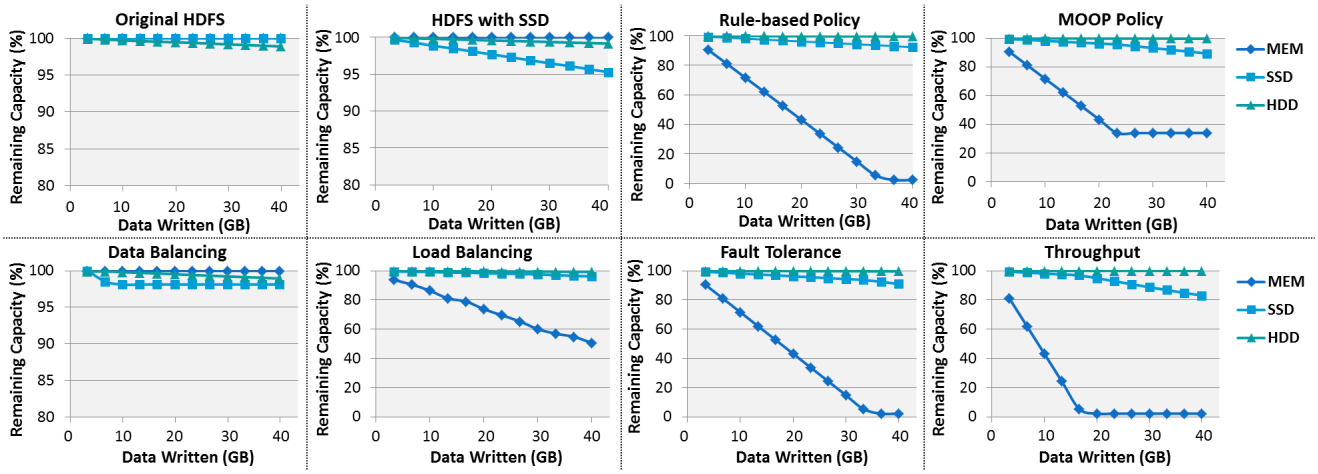

Figure 4: Remaining capacity percent per storage tier for eight data placement policies.

tion (*TM*) policy starts with the highest throughput of ~200 MB/s as it is heavily exploiting the "Memory" tier. However, throughput quickly degrades as the memory space gets exhausted and all the burden falls on the SSDs. Both the Load Balancing (*LB*) and the Fault Tolerance (*FT*) policies offer relatively good performance that averages ~110 MB/s with some minor fluctuations, as they utilize all three tiers more uniformly. Recall that *FT* strives to maximize the use of all tiers and we enabled the use of the "Memory" tier for fairness. The Data Balancing (*DB*) policy tries to equate the percent remaining capacities of the storage media. Given the large differences in total capacities, *DB* is biased towards the "HDD" tier and yields the lowest performance (~58 MB/s), as it completely ignores any performance metrics.

The *MOOP* policy successfully combines the strengths of the four individual-objective policies and achieves the best overall and most reliable performance of ~125 MB/s. Initially, the *MOOP* policy uses all three tiers for placing replicas and achieves good performance. As the memory usage increases, the policy starts utilizing the "SSD" tier more while spreading the load on the "HDD" tier among the individual devices (see Figure 4). On the other hand, the *Original HDFS* policy spreads the load fairly evenly across all disks on the Workers with an average and steady throughput of ~88 MB/s. Allowing HDFS to also store data on SSDs adds a moderate improvement to the write throughput averaging ~98 MB/s, even though 25% of the data gets placed on SSDs. This result highlights the need for smart management of the available storage media; simply adding better media in a cluster will not yield significant performance improvements. Overall, the *MOOP* policy achieves a 42% and 29% increase on average write throughput over *Original HDFS* and *HDFS with SSD*, respectively.

In comparison, the write throughput of the *Rule-based* policy at ~108 MB/s is better compared to both versions of the HDFS policies but inferior to the performance achieved by the *MOOP* policy. Taking into account the current load and remaining capacity per storage media provides a significant advantage to the *MOOP* policy, which is able to achieve a 17% increase of write throughput per Worker node in the cluster compared to the *Rule-based* policy.

In order to study how data placement effects the read performance, the average read throughput per Worker is shown in Figure 3(b). The *TM* policy exhibits a similar pattern with the write throughput, where very good read throughput is achieved until memory gets exhausted. The *LB* policy exhibits an interesting oscillating trend explained by combining two facts: (i) *LB* tries to evenly distribute I/O requests across all storage media and (ii) the cluster has 3 times as many HDDs as SSD or Memory media. Hence, it will periodically write many blocks having all three replicas on HDDs, which in turn leads to periods of slower

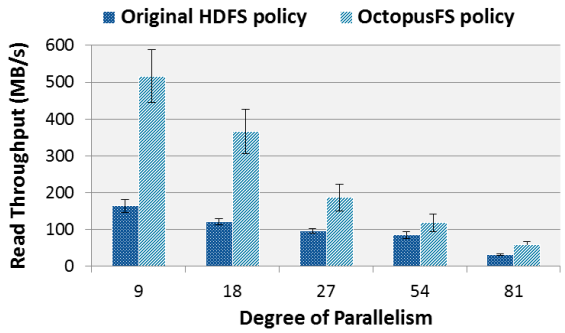

Figure 5: Average read throughput per Worker for five degrees of parallelism and two retrieval policies.

read performance while reading those blocks. The *FT* and *DB* policies exhibit average and poor read performance, respectively, while the *Rule-based* policy behaves similarly to the *FT* one. The *Original HDFS* policy offers the lowest overall read performance, while the addition of SSDs led to marginal improvement. Once again, *MOOP* offers good and reliable performance, even though there is some decline in throughput after it starts utilizing memory less. Even at that point, it significantly outperforms both HDFS policies with a 2.1x better read throughput rate.

7.3 Data Retrieval Evaluation

This section compares the OctopusFS data retrieval policy described on Section 4.2 against the original HDFS policy, which only focuses on data locality (and ignores the storage tiers). DFSIO was used to generate 10GB of data using the *MOOP* data placement policy, followed by reading the data with each of the two policies. Figure 5 shows the average read throughput per Worker achieved for various degrees of parallelism. In all cases, the OctopusFS retrieval policy exhibits much higher throughput rates compared to HDFS as it schedules more reads from replicas stored on higher tiers. As the degree of parallelism d increases—and along it network congestion—the throughput rates decrease for both policies, as expected. The benefits from using OctopusFS also decrease from ~4x to ~2x as d increases, but stay significant from efficiently utilizing all storage tiers.

7.4 Efficiency of Namespace Operations

The OctopusFS Master (and equivalently, the HDFS NameNode) is responsible for handling all directory namespace operations efficiently. This experiment utilizes the S-Live Test [31] for stressing both the Master and the NameNode with the same workload of typical file system operations. The experiment was repeated four times and the average numbers of successful operations per second per Worker are shown in Table 3 along with the standard error of the mean. The results indicate that OctopusFS, despite the extra processing related to the tiers and the management policies, offers very similar performance to the original HDFS. Most operations exhibit less than 1% overhead, which is within the standard errors recorded during the experiments.

The memory overhead introduced by the extra information kept by OctopusFS is also trivial. The replication vector for each file is encoded into 64 bytes (48 bytes more than the old replication factor). In addition, the Master keeps 448 bytes of extra statistics for each storage media and 704 bytes for each tier. Even for a huge cluster with 3500 nodes and more than a dozen storage media each [30], the additional overhead is only a few tens of megabytes.

Table 3: HDFS vs. OctopusFS on namespace operations per second per Worker.

Operation	HDFS	OctopusFS
Make directory	140.5 ± 4.6	135.9 ± 6.1
List files	7089.0 ± 43.8	7143.0 ± 85.3
Create file	54.9 ± 1.6	53.4 ± 1.6
Open file	5937.4 ± 116.3	5897.1 ± 41.2
Rename file	111.5 ± 6.2	111.1 ± 3.7
Delete file	49.8 ± 1.7	47.1 ± 1.5

7.5 Evaluation over Hadoop and Spark

This set of experiments evaluates the end-to-end benefits OctopusFS can immediately provide to data processing workloads running on any platform that supports HDFS. We used the HiBench benchmark [13], which provides implementations for various workloads on both Hadoop MapReduce and Spark. In total, nine workloads were used spanning three categories: micro benchmarks (Sort, Wordcount, Terasort), OLAP queries (Scan, Join, Aggregation), and machine learning analytics (Pagerank, Bayesian Classification, k-means Clustering). We executed all workloads over Hadoop v2.7.0 and Spark v1.6.2, once storing data in HDFS and once in OctopusFS. None of the systems were modified as OctopusFS is backwards compatible with HDFS.

Figure 6 shows the normalized execution time with OctopusFS over HDFS for all workloads on both platforms. (The actual execution times of the workloads varied between 1 and 42 minutes.) The OctopusFS usage translated into performance gains for every single workload ranging from 6% up to 72% (i.e., 3x speedup) of execution time improvement. The performance benefits for Hadoop MapReduce were profound with an average of 35% improvement (~1.8x speedup), while for Spark they were more modest, with an average of 17% improvement (~1.2x speedup). The lesser benefits for Spark are expected given it already utilizes cluster memory heavily. We are confident that extending both platforms to take advantage of the controllability features offered by OctopusFS can lead to even larger performance gains.

7.6 Evaluation of an Enabling Use Case

Section 6 presented various use cases enabled by the fine-grained control OctopusFS provides to applications. As a proof of concept, we have modified Pegasus [16], a graph mining system running over Hadoop, to include two optimizations. First, when an iterative workload starts running, Pegasus will identify the datasets that are reused in each iteration and will instruct OctopusFS to *prefetch* (i.e., move) one replica into the Memory tier. Second, Pegasus will identify short-lived *intermediate data* produced and consumed between jobs, and instruct OctopusFS to store one copy in the "Memory" tier. For the experiments, we generated a graph with 2 million vertices (3.3 GB in size) and run four typical graph mining workloads: Pagerank, Connected Components (ConComp), Graph Diameter and Radius (HADI), and Random Walk with Restart (RWR) [16]. All workloads converged and concluded in less than four iterations.

For comparison purposes, we first executed the workloads using the unmodified Pegasus over HDFS and OctopusFS. Then, we used the modified Pegasus enabling the two opti-

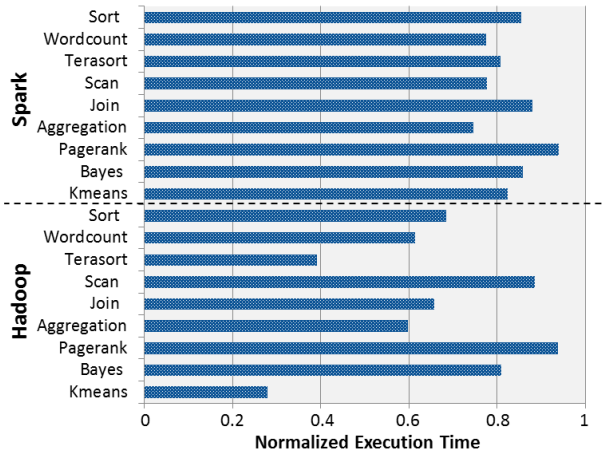

Figure 6: Normalized execution time of Hadoop and Spark workloads using OctopusFS over using HDFS.

mizations separately and together. Figure 7 shows the normalized execution time over HDFS for all workloads. The automated policies of OctopusFS are able to provide performance benefits on their own ranging from 15% to 34% over HDFS. The addition of the two optimizations in Pegasus increases the gains further: prefetching adds 3–7% of gains while intermediate data handling adds 7–16% of gains over OctopusFS. The latter gains are substantial in some cases due to the large amount of intermediate data. For example, the HADI workload generates about 18 GB of intermediate data per iteration. As the two optimizations are complementary to each other, enabling both leads to even greater benefits of 10–18% over OctopusFS and 25-52% over HDFS.

8. RELATED WORK

Distributed file systems: HDFS [30] has been the basis for our implementation. We have made significant modifications and additions to HDFS, including the introduction of performance-based storage tiering, the use of memory and remote storage for hosting blocks, the notion of the replication vector, and the use of automated data management policies. A recent version of HDFS has added the notion of a "storage type" to the attached media but it is based only on the type of the devices, not their performance characteristics [1]. In addition, there is support for a limited number of static policies for storing and moving files but all block replicas must be hosted on the same tier [29]. These policies require manual configuration and are meant primarily for archival purposes [4, 32]. The *MapR File System* [22] and *Ceph* [34] have similar architectures to HDFS but both offer a distributed metadata service as opposed to the centralized HDFS NameNode. The *Quantcast File System (QFS)* [27] employs erasure coding rather than replication as its fault tolerance mechanism for reducing the amount of storage used.

Multi-tier file storage: *Hierarchical storage management (HSM)* provides a policy-based way of managing data across a storage hierarchy that typically consists of arrays of tapes, compressed disks, and high-performance disks [36]. The main focus is on archiving inactive files to the lower tiers and retrieving them upon reference. Unlike our system, HSM does not offer any locality or storage-media awareness to higher-level applications. *Storage-tier-aware file systems* form the evolution of HSM and are aware of abstract device

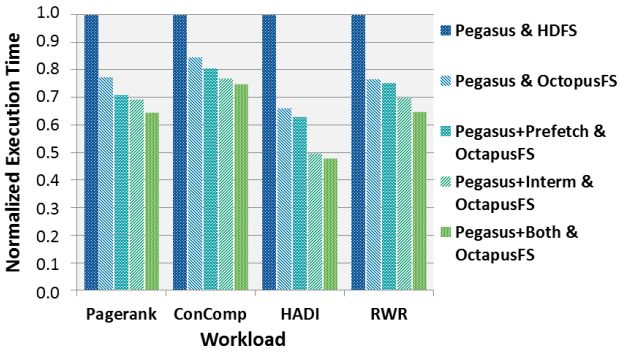

Figure 7: Normalized execution time of Pegasus workloads with various optimizations.

"types" (arrays of SSDs and HDDs). They typically offer some control over initial placement and movement of data based on policies [10]. However, each file must reside completely on a tier and there is no notion of locality awareness.

Memory usage in compute clusters: Memory caching is a standard technique for improving local data access in distributed applications. *PACMan* [3] is a memory caching system that explores memory locality of data-intensive parallel jobs but always places all replicas of all file blocks in memory. Furthermore, PACMan does not allow applications to specify hot data in memory for subsequent efficient accesses. *Resilient Distributed Datasets (RDDs)* are a distributed memory abstraction and a new programming interface for in-memory computation [37]. In addition, it allows applications to persist a specified dataset in memory for reuse and uses lineage for fault tolerance. RDDs are specialized for iterative algorithms and interactive data mining tools, whereas OctopusFS offers a general-purpose file system with superset capabilities. *Tachyon* [21] also incorporates lineage to speedup mainly the write throughput of workloads consisting of batch jobs.

Remote storage use in clusters: *MixApart* [24] and *Rhea* [9] focus on improving data retrieval from remote enterprise storage systems to local computing clusters by utilizing on-disk caching at the local disks. In addition, MixApart schedules tasks according to both on-disk caching and remote data. Rhea uses static analysis techniques of application code to generate storage-side filters, which remove irrelevant or redundant data transfers from storage nodes to compute nodes. However, the target applications of Rhea are not I/O intensive; they are supposed to have high data selectivity.

9. CONCLUSIONS

This paper presented the design for a novel distributed file system (OctopusFS) for cluster computing that is aware of various heterogeneous storage media, such as memory, SSDs, HDDs, and remote storage, with different capacities and performance characteristics. OctopusFS contains automated data-driven policies for managing the placement and retrieval of data across the nodes and the storage tiers of the cluster. In addition, it exposes the network locations and storage tiers of the data in order to allow higher-level applications to make locality-aware and tier-aware decisions, opening up new interesting research directions. Overall, OctopusFS offers a flexible storage solution that can achieve better I/O performance and cluster utilization.

10. REFERENCES

[1] A. Agarwal. *Enable Support for Heterogeneous Storages in HDFS*, 2015. https://issues.apache.org/jira/browse/HDFS-2832.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, et al. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[3] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, et al. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proc. of the 9th Symp. on Networked Systems Design and Implementation (NSDI)*, pages 267–280. USENIX, 2012.

[4] B. Antony. *HDFS Storage Efficiency Using Tiered Storage*, 2015. http://www.ebaytechblog.com/2015/01/12/hdfs-storage-efficiency-using-tiered-storage/.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pages 25–36. ACM, 2011.

[7] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[9] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. I. Rowstron. Rhea: Automatic Filtering for Unstructured Cloud Storage. In *Proc. of the 10th Symp. on Networked Systems Design and Implementation (NSDI)*, pages 343–355. USENIX, 2013.

[10] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami. Cost Effective Storage using Extent Based Dynamic Tiering. In *Proc. of the 9th Conf. on File and Storage Technologies (FAST)*, volume 11, pages 20–34. USENIX, 2011.

[11] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proc. of the 9th Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–8. USENIX, 2010.

[12] *Apache Hadoop FileSystem API*, 2016. https://hadoop.apache.org/docs/stable2/hadoop-project-dist/hadoop-common/filesystem/index.html.

[13] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *New Frontiers in Information and Software as Services*, pages 209–228. Springer, 2011.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[15] H. Jeon, K. El Maghraoui, and G. B. Kandiraju. Investigating Hybrid SSD FTL Schemes for Hadoop Workloads. In *Proc. of the 2013 ACM Intl. Conf. on Computing Frontiers (CF)*, pages 20:1–20:10. ACM, 2013.

[16] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: Mining Peta-scale Graphs. *Knowledge and Information Systems*, 27(2):303–325, 2011.

[17] K. Krish, A. Khasymski, A. R. Butt, S. Tiwari, and M. Bhandarkar. AptStore: Dynamic Storage Management for Hadoop. In *Proc. of the 5th IEEE Intl. Conf. on Cloud Computing Technology and Science (CloudCom)*, pages 33–41. IEEE, 2013.

[18] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proc. of the VLDB Endowment*, 5(12):1790–1801, 2012.

[19] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proc. of the 2008 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1075–1086. ACM, 2008.

[20] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A Platform for Scalable One-pass Analytics Using MapReduce. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pages 985–996. ACM, 2011.

[21] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proc. of the 5th Symposium on Cloud Computing (SoCC)*, pages 1–15. ACM, 2014.

[22] *MapR File System*, 2013. http://www.mapr.com/products/apache-hadoop.

[23] R. T. Marler and J. S. Arora. Survey of Multi-Objective Optimization Methods for Engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[24] M. Mihailescu, G. Soundararajan, and C. Amza. MixApart: Decoupled Analytics for Shared Storage Systems. In *Proc. of the 11th Conf. on File and Storage Technologies (FAST)*, pages 133–146. USENIX, 2013.

[25] H. Ogawa, H. Nakada, R. Takano, and T. Kudoh. SSS: An Implementation of Key-Value Store Based MapReduce Framework. In *Proc. of the 2nd IEEE Intl. Conf. on Cloud Computing Technology and Science (CloudCom)*, pages 754–761. IEEE, 2010.

[26] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, et al. The Case for RAMCloud. *Communications of the ACM*, 54(7):121–130, 2011.

[27] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proc. of the VLDB Endowment*, 6(11):1092–1101, 2013.

[28] K. Pal. *Enhance Existing HDFS Architecture with HDFS Federation*, 2014. http://www.devx.com/opensource/enhance-existing-hdfs-architecture-with-hadoop-federation.html.

[29] A. Purtell. *Support Tiered Storage Policies in HDFS*, 2015. https://issues.apache.org/jira/browse/HDFS-4672.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of the*

26th IEEE Symp. on Mass Storage Systems and Technologies (MSST), pages 1–10. IEEE, 2010.

[31] K. V. Shvachko. *Stress Test for Live Data Verification (S-Live)*, 2010. https://issues.apache.org/jira/secure/attachment/12448004/SLiveTest.pdf.

[32] T.-W. N. Sze. *Support Archival Storage in HDFS*, 2015. https://issues.apache.org/jira/browse/HDFS-6584.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. of the VLDB Endowment*, 2(2):1626–1629, 2009.

[34] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI)*, pages 307–320. USENIX, 2006.

[35] T. White. *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[36] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Trans. on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of the 9th Symp. on Networked Systems Design and Implementation (NSDI)*, pages 15–28. USENIX, 2012.