HReplica: A Dynamic Data Replication Engine with Adaptive Compression for Multi-Tiered Storage

Hariharan Devarajan, Anthony Kougkas, Xian-He Sun
Illinois Institute of Technology, Department of Computer Science
hdevarajan@hawk.iit.edu, {akougkas, sun}@iit.edu

***Abstract*—As the diversity of big data applications increases, their requirements diverge and often conflict with one other. Managing this diversity in any supercomputer or data center is a major challenge for system designers. Data replication is a popular approach to meet several of these requirements, such as low latency, read availability, durability, etc. This approach can be enhanced using new modern heterogeneous hardware and software techniques such as data compression. However, both these enhancements work in isolation to the detriment of both. In this work, we present HReplica: a dynamic data replication engine which harmoniously leverages data compression and hierarchical storage to increase the effectiveness of data replication. We have developed a novel dynamic selection algorithm that facilitates the optimal matching of replication schemes, compression libraries, and tiered storage. Our evaluation shows that HReplica can improve scientific and cloud application performance by 5.2x when compared to other state-of-the-art replication schemes.**

***Index Terms*—data replication, dynamic, selection algorithm, multi-tiered, data compression, intelligent selection, dynamic programming, cloud application, scientific application, big data.**

I. Introduction

Modern scientific and cloud applications are extremely diverse, ranging from cosmology [1], astrophysics [2], fusion [3] simulations to Alice high-energy physics [4], bioinformatics, and computational finance applications [5]. Scientific discovery is driven by the ability to efficiently and reliably process data with high-throughput and low latency [6]. *Data replication*, a widely used technique which plays a pivotal role in achieving these goals [7], can enable several features such as a) low-latency read operations in Alice HEP, b) fault tolerance in cosmology application MADCAP, and c) load balancing for VORPAL plasma physics simulation. Each data replication strategy aims to provide a unique set of features and constraints, giving high performance and reliability to the application. Based on the methodology adopted for replication, different features with constraints could be obtained. Examples include minimize read latency while maximizing write bandwidth [8], minimize read interference while maximizing write latency [9], and n fault tolerance while minimizing storage cost [10].

Researchers further explore data replication due to two orthogonal developments which have directly influenced data replication schemes. First, the invention of new storage hardware devices, where additional levels of memory and storage are stacked in a hierarchy. This is creating an architectural trend where modern supercomputers and data centers have heterogeneous storage devices to provide low latency and high throughput to users. In most cases, these storage devices are tiered in a hierarchy and act as a fast cache for the application. For example, the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC) employs an SSD-based burst buffer technology [11]. Oak Ridge National Lab's Summit also uses fast local NVMe storage for buffering [12]. Data centers of hyperscalers [13] such as Google, Amazon, and Microsoft use fast node-local devices as a cache to speed up data access for applications. This new architectural trend has led data replication schemes to adapt to this heterogeneous [14]–[16] environment. These schemes build a cost model around the performance and capacity of storage devices to optimize read availability, data locality, fault tolerance, etc. The heterogeneity of storage resources adds complexity to replication engines but also adds benefits such as low storage cost, better QoS guarantees, and higher performance. Second, a plethora of data compression algorithms have been invented to account for the explosion of data volume and variety. Data compression is a data reduction technique which encodes data in a way that decreases the data footprint and, thus, the cost of I/O. Hence, this technique can greatly benefit application performance by balancing compression/decompression time with the cost of I/O to the underlying storage system. Data compression is utilized in replication [17], [18] to increase I/O bandwidth, reduce storage footprint, reduce network I/O cost, and enable security. However, data compression increases the latency of operations performed by the clients. Hence, it is crucial to use compression intelligently to balance this trade-off.

Even though both hierarchical storage and data compression improve the effectiveness of data replication, we identify several challenges in the current approaches. *First*, different replication schemes are designed for different application/system requirements. These requirements are often conflicting and none of the solutions provide a tunable choice of these requirements in the same system. *Second*, most replication engines use heterogeneous hardware as separate devices with particular performance and capacity characteristics. However, these storage devices are arranged in a hierarchical setup. The benefit of such a setup stems from hiding slower-but-larger mediums behind the latency of smaller-but-faster mediums. This is achieved by using parallelization techniques such as pipelining, concurrent access lanes, and smart scheduling. As the existing solutions do not consider these hierarchical characteristics, they do not efficiently leverage the potential of the storage system. *Finally*, data compression is a powerful tool to improve the effectiveness of data replication by significantly reducing the I/O and storage cost. Yet, all replication schemes use a single static compression library. Devarajan et. al. [19] showcased the variability of compression performance based on input characteristics (e.g., data type, format, and distribution). To mitigate these problems and maximize the benefits of data replication, we need an adaptive replication selection engine which can dynamically choose data compression and hierarchical storage.

In this work, we propose HReplica: a dynamic data replication engine with adaptive compression for multi-tiered storage environments. At the core of HReplica is an optimization engine which chooses the appropriate replica destination (i.e., node + storage tier), compression library, and replication scheme for a given data input, user requirement, and available system configuration. The optimization traverses the multi-dimensional solution space to find the optimal combination of location, compression algorithm, and replication scheme to achieve the desired user requirement. HReplica uses a dynamic programming approach and a cost-based model, enhanced with reinforcement learning, to predict the performance characteristics of each replication scheme coupled with a compression library and a targeted storage tier. HReplica is designed as a modular component which can be placed on top of any storage solution as its replication engine.

The contributions of HReplica can be summarized as:

1) Demonstrating the benefit of using a storage hierarchy and compression to dynamically tune data replication (III-A).
2) Designing a dynamic replication engine with adaptive compression for tiered storage environments (IV).
3) Introducing a novel dynamic selection engine tailored to application requirements and system specification (IV-C).
4) Quantifying the benefit of such an engine for complex scientific and cloud applications (V).

II. Background and Related Work

A. Data Replication

For many years, data replication has been studied thoroughly in the World Wide Web [20], peer-to-peer networks [21], [22], ad-hoc and sensor networking [23], [24], and mesh networks [25]. Data replication is a technique that aims to increase data availability by creating multiple copies of the same data, called replicas, and distributing them at multiple locations [26]. This technique trades increased I/O cost during writing in order to offer fault tolerance and high data availability for read operations. It is extensively used in supercomputers and data-centers to decrease the user's waiting time and minimize network consumption by utilizing different replicas of the same service [27]. Many distributed storage systems such as parallel file systems [28], key-value stores [29], [30], and distributed log stores [31], [32] use data replication for two main reasons: to optimize data access and/or increase data durability. Data replication can optimize data access by increasing spatial data locality, decreasing I/O and network interference, and enabling better load balancing in a distributed environment. Also, data replication can make data more resilient to hardware and software faults by increasing the number of replicas and placing them in several locations. However, replication also incurs costs such as increased storage footprint, write latency, and network consumption. Hence, the primary goal of a data replication engine is to manage the trade-off between the features provided and the costs incurred. These objectives often conflict with one another. For instance, load balancing within the cluster can inhibit the spatial data locality or decreasing storage footprint by reducing the number of replicas can affect the degree of fault tolerance. Hence, most modern data replication engines focus on providing a specific feature for the applications while minimizing the cost of replication (expressed as constraints). A summary of all existing data replication engines based on their features and costs are presented in Table I. As seen in the table, all state-of-the-art replication schemes provide at most two features while minimizing replication cost. In a multi-tenant environment, balancing data access performance and data durability semantics is a very challenging task and replication engines have to be re-designed to provide tunability and flexibility. This is the primary contribution of HReplica: to create a tunable data replication engine for different application requirements.

B. Data Replication in Heterogeneous Environments

Data replication in heterogeneous environments pose a great challenge in the form of storage reliability and performance. Wenhao et. al. [14] proposed a cost-effective dynamic data replication strategy, which facilitates an incremental replication method, to reduce the storage cost and meet the data reliability requirement at the same time in a diverse, heterogeneous cluster. The replica placement algorithm accounts for the heterogeneity by building a cost model of storage resources such as reliability, performance, and capacity. Navneet et. al. [15] proposed a cross data-center replication method that uses a dynamic knapsack algorithm to optimize the trade-off between the cost of replication and data availability. It re-replicates the replicas from higher-cost data centers to lower-cost data centers without affecting data availability. Jaing et. al. proposes a pattern-directed replication scheme (PRS) [16] to achieve efficient data replication for heterogeneous storage systems. PRS selectively replicates data objects and distributes replicas to various storage devices based on their features. It first groups objects with similar data access patterns and then replicates all objects in a group. Jiong et. al. proposes a heterogeneous-aware data placement algorithm [33] to adaptively balance the amount of data stored in each node to improve the data processing performance. These solutions adaptively balance the performance characteristics and data availability requirements of the application and dynamically choose the number of replicas to improve read performance. All of these works take into consideration the storage drive capacity and the CPU heterogeneity of a node, but do not consider the hierarchical nature of multi-tiered storage [34]. Hierarchical techniques

TABLE I
FEATURES VS CONSTRAINTS IN DATA REPLICATION TECHNIQUES

Constraint (reducing cost) Features	**Low write latency**	**High write bandwidth**	**Low storage cost**
Low read latency	RTRM [9]	MORM [8]	MOE [8], MORG [35]
High read bandwidth	GFS [36], OMR [37]	SSOR [38]	MOE, MORG
Low read interference	RTRM	D2RS [39], HQFR [40]	CIR [14]
Data resiliency		CDRM [41]	DCR2S [42]
Fault tolerance	MinCopysets [43]	RFH [10], MORM	RFH
Load balancing	OMR, SSOR	HQFR	CIR

such as I/O pipelining and concurrency can positively affect data replication. These ideas are explored in this paper.

C. Compression in Data Replication

To reduce the cost of replication and maximize its benefits, some researchers have proposed to use data compression. Russell et. al. proposed a page replication scheme [17] for databases with page-level compression. The compression reduces both the storage footprint as well as the cost of I/O and networking. Navendu et. al. proposes a data geo-replication [18] where it uses delta compression and gzip to reduce the network cost of placing the replicas. This technique showcases that reduction in I/O can greatly decrease the cost of replication. Jinwei et. al. proposed a population-aware, cost-effective and resilient (PMCR) data replication algorithm [44]. PMCR utilizes delta compression to reduce storage and bandwidth cost. These examples show the promise of utilizing data compression for replication. However, all the proposed solutions use data compression in a naive way by applying the same compression technique across all replicas when it has been proven that different compression algorithms show significantly different performance for different inputs [19]. Each compression library optimizes for a compression metric (e.g., compression/decompression speed, compression ratio). However, compression objectives are orthogonal. Thus, choosing a single compression technique for all replica copies can lead to suboptimal results. Our work explores a more dynamic and intelligent compression technique applied on replicas that could lead to higher performance.

III. MOTIVATION AND PROBLEM STATEMENT

A. Motivation

Data replication can benefit from both data compression and heterogeneous storage. In this work, we are motivated to build the first data replication engine that combines the power of intelligent data compression and selective replica placement in a hierarchical storage environment. To better understand the impact of different replication engines, we run a synthetic benchmark and measure metrics such as the replication cost (i.e., write time) and access performance (i.e., read time). In this test, each process first writes a single MB of data in a file per process fashion and repeatedly reads it back 10 times. This cycle repeats 32 times. We ran this test with 2560 processes

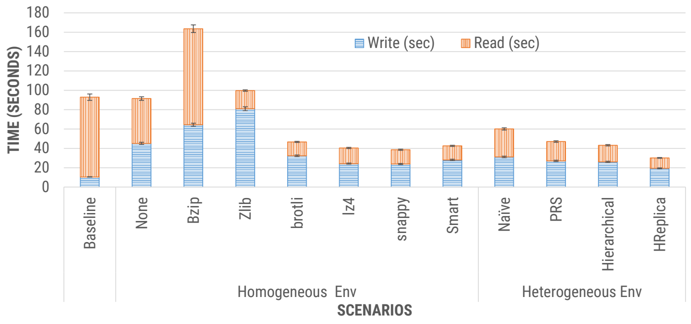


Fig. 1. Synthetic benchmark with data replication

for a total data size of 800 GB organized in an HDF5 file. We investigate how different replication schemes can boost read performance by increasing the data availability. We ran this test on the Ares cluster [45] at Illinois Tech. We deployed two 32-node storage environments, one homogeneous OrangeFS [46] installation running on 32 HDD drives, and one heterogeneous installation using 16 HDD, 8 SSD, and 8 NVMe drives. We have implemented different replication engines and the test cases are: a) Baseline without replication b) Homogeneous replication with/without compression c) Heterogeneous replication but not hierarchical based on PRS [16]. d) HReplica, where the replication scheme uses a combination of smart compression libraries and hierarchical storage to maximize read performance. Figure 1 shows the results, with x-axis showing various configurations as discussed above and y-axis depicting the time elapsed in seconds for both write (i.e., replication cost) and read.

Results show that without using data replication, the benchmark takes 92.67 seconds to complete. When the benchmark runs with homogeneous data replication strategy enabled, the application's read cost is reduced by 1.8x over the baseline with increased write cost of 4.32x. When compression is enabled, significant application performance variability can be observed. Specifically, when light compression is applied (`LZ4`), a 5x reduction in the read time is achieved by the smaller data size (i.e., 5x compression ratio) with only 3x increased write cost. On the other hand, when heavy compression is applied (`Bzip`), the benefit of compression is offset by the prolonged compression time (i.e., 9x compression ratio but 1.5x slower read time). This performance variability can be caused by the ability of each compression library to apply meaningful compression [19]. Furthermore, read performance is increased by 5x when the heterogeneous replication scheme is used. This is due to the better hardware used for storing replicas. If we enable hierarchical optimizations (e.g., operation pipelining and concurrent access to the hierarchical device) we can further increase read performance by 1.35x. Lastly, HReplica combines the benefits of heterogeneous storage, with its hierarchical architecture, and smart compression together. For instance, using a combination of `Snappy` and `Brotli` enables HReplica to increase read performance by 5.7x while penalizing write performance by only 2.8x. Therefore, as results show, combining data replication schemes, compression, and hierarchical replica placement is quite rewarding. This dynamic matching would both increase the resource utilization of the storage resources and enable more effective data replication.

B. Problem Formulation

HReplica aims to maximize the benefits of data replication by building a replication scheme that intelligently combines data compression and multi-tiered storage. Consider an application consisting of n data tasks, a collection of C compression libraries, and a hierarchical environment with L storage tiers. The optimization problem is shown in Table II. In this formulation, higher tiers (i.e., tiers with higher bandwidth and lower latency) have a smaller index. The objective function tries to minimize the replication cost by reducing the synchronous write cost. This is done by compressing replicas and placing them on higher tiers. This depends on the replication requirements (e.g., synchronicity, # of replicas, latency requirements, etc) as well as tier characteristics such as the tier's available capacity, access latency, and bandwidth. In the formulation, parameters r_n, r_m, w_c, w_d, and w_r (i.e., first two are replication metrics and last three are compression priorities) are tuned based on requirements at different levels (i.e., system, application, and operation level). Furthermore, the objective function also considers the possibility of no compression since under certain system configurations, data compression might hurt the overall replication performance [47].

TABLE II
PROBLEM FORMULATION

Given	i, a replication task r, HReplica's replication scheme r_n, number of replicas r_m, mode of replication synchronous/asynchronous C, a set of compression algorithms with each element c t_c, compression time for algorithm c t_d, decompression time for algorithm c r_c, compression ratio for algorithm c w_c, weight for compression time of algorithm c w_d, weight for decompression time of algorithm c w_r, weight for compression ratio of algorithm c L, a set of tiers with each element l
Define	P, indivisible sub-tasks of task i $Size(p)$, size of piece p $Concurrency(L)$, sum of hardware lanes in all tiers $Length(x)$, length of vector x $Duration(p, c, l)$, the time taken to execute sub-task p with compression c on tier l.
Minimize	$\sum_{p=0}^{P} Duration(p, r, c, l)$
Subject to (constraints)	1. Size(p) $\mod 4096 = 0$ 2. Length(P) $\leq Concurrency(T)$ 3. Length(P) $\leq Length(L)$ 4. r$_c \geq 1$ 5. Size(p) $\leq Size(l)$ 6. w$_c + w_d + w_r = 1$

Note, constraints 1-3 ensure a small number of sub-problems keeping the cost of the dynamic programming algorithm low. Additionally, constraint 1 makes sub-problems highly reusable which further reduces the complexity of the algorithm. Constraint 4 ensures the selected compression will actually result in data reduction. Constraint 5 guarantees that a sub-task can fit in a target tier. Finally, constraint 6 normalizes the weights across different metrics of compression.

IV. HReplica: Hierarchical Data Replication

HReplica is a data replication engine that utilizes smart data compression and multi-tiered storage to holistically optimize the replication process and reduce its cost. HReplica decomposes a replication task into a replication scheme optimally matching an appropriate compression library and storage tier by a dynamic and intelligent data replication algorithm. HReplica contains four key components: a Replication Handler (RH), a Pattern Analyzer (PA), a Compression Cost Predictor (CCP), and an Engine. The API transforms each replication request into a 3-tuple task {data buffer, replication requirements, constraint}. The RH analyzes the task data to identify data characteristics and transform the task requirements and constraints into replication and compression metrics. The metrics along with system data (acquired by a Monitoring Interface (MI)) and data access pattern (from a PA) are passed to the Engine. The Engine uses this knowledge to solve the optimization problem and produce a data replication scheme which includes the data decomposition, replication parameters, compression library, and targeted storage tier. Then the scheme is passed to the Replication Executor (RE), which performs replication and stores this information into the Replica Catalog (RC). HReplica is designed with the following principles in mind:

1) **Hierarchy-aware:** HReplica should build an optimal replication scheme by applying the appropriate compression algorithm per tier by leveraging the different tier performance characteristics. This increases the replication effectiveness and storage utilization.
2) **Tunable:** HReplica should produce tunable replication schemes satisfying user-requirements on different levels (system-, application-, and operation-level). Tunability is a core requirement of modern dynamic multi-tenant systems.
3) **Dynamic:** HReplica should be able to dynamically adapt to the changing replication requirements and constraints with negligible overheads upon switching policies. Additionally, it should transparently provide an interface to interact with different replication and compression schemes in real-time.

A. Design and Architecture

Figure 2 shows the architecture of HReplica. HReplica is designed as a data replication plugin available to existing storage systems. System software which performs I/O can be compiled with the library (libhreplica.so) and use its native API (e.g., replicate() and locate()). To increase portability, the HReplica library can also be dynamically linked using LD_PRELOAD. In this case, systems need to build their interface which can be translated to HReplica APIs external to the system software and pre-loaded in the system environment. This way, HReplica can transparently intercept replication tasks and redirect them to its internal API. The replication requirement can be passed at different levels to the engine, a) system level: defined through a configuration, b) application level: defined by a special application initialization API (e.g.,

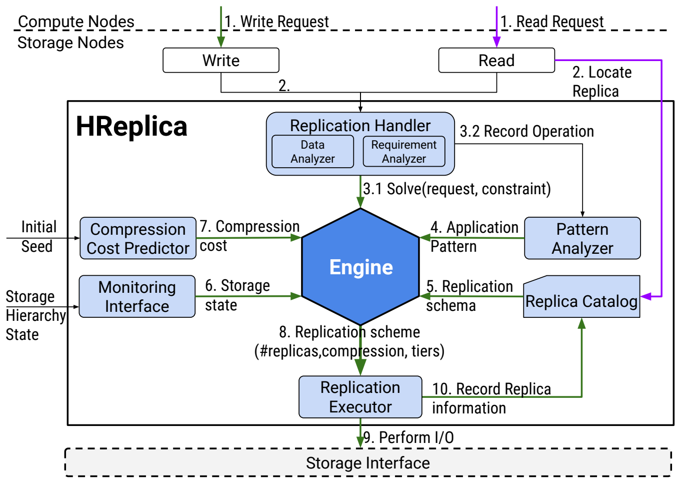


Fig. 2. HReplica architecture

during MPI_Init), and c) operation level: when replication is invoked. In the case of LD_PRELOAD, the operation level requirements are ignored. HReplica targets modern extreme-scale system designs with compute node-local NVMe drives, shared burst buffers, and a remote PFS. However, the design of HReplica is generic and works with n-tiers of a storage hierarchy. The information about the tiers (e.g., bandwidth, device location, interface, etc.) is externally provided by the deployed storage. We use a light weight system monitor service (ideally existing monitoring services in cluster such as Ganglia [48] could be used) which tracks the available capacity of storage.

The flow of HReplica's replicate operation is as follows. In the case of *write* (green arrows in the figure), the storage solution uses HReplica's API with input data, replication requirements, and constraints. The associated data, requirement, and constraint is passed for analysis to the RA (step 2) which identifies the data attributes (e.g., type, distribution, and format). Additionally, the RA combines the operation requirements with those of the application and the system and produces a list of replication metrics (i.e., r_n and r_m) and compression priorities (i.e., w_c, w_d, and w_r), called HReplica Metrics (HRM). In parallel, the input is passed to the PA to analyze the application's access pattern (step 3.2). This prediction is based on a data-centric scoring of data [49] which determines the hotness of the data. The application's data access pattern drives the decomposition decision for replication. The CCP maintains a table of expected costs for each combination of the above data attributes and chosen compression library (as presented in [19]). Furthermore, the MI obtains the current storage status, namely tier availability and their respective remaining capacity. The Engine uses the HRM (step 3.1), application pattern (step 4), current replica distribution (step 5), and storage state (step 6) to produce an optimal replication scheme. During the optimization, each combination of solution consults with the CCP (step 7) to estimate the compression and I/O costs which are then minimized by the optimizer. A replication scheme consists of P pieces of replica data, where each piece p is coupled with a target tier, the ideal compression library, and the replication metrics given by the optimization. The produced scheme is then passed to the RE (step 8), containing the compression interface, which dynamically chooses the required compression algorithm. Once the compression is executed, the replication executor sends the compressed data to the storage interface to perform I/O (step 9) and updates the RC (step 10).

B. Replication Handler

The analyzer has two internal components, a) Data Analyzer and b) Requirement Analyzer.

1) The Data Analyzer (DA): is responsible for deducing the input data characteristics such as type, distribution, and format. For data-type and format inference, HReplica uses state-of-the-art techniques such as sub-sampling, binary decoding, and introspection [19]. The DA also examines the content distribution (as certain distributions are more compressible [50]) and classifies each input buffer as Normal, Gamma, Exponential or Uniform. Distribution detection is performed statically using techniques such as sub-sampling and random partitioning [51]. Lastly, several data attributes can be easily obtained using metadata parsing of self-described portable data representations (e.g., HDF5, NetCDF, Avro, RDD, Parquet, etc.) used in most scientific and cloud applications. Hence, in most practical cases, the DA is fast and reasonably accurate.

2) The Requirement Analyzer (RA): enables the dynamic translation of various user-requirements into replication and compression properties. Additionally, it detects requirement conflicts and resolves them by requirement priority. The default priority of requirements is operation-level over application-level over system-level. In case of conflicts on the same level (e.g., user sets an operation to be synchronous and have low write latency) the module applies the first one only. The supported parameters, which can be tuned at each level, are boolean or integral variables and include: minimize write latency, minimize read latency, maximize read availability, maximize write availability, maximize durability, minimize storage cost, maximize load balancing, and minimize energy cost. These constraints and requirements are often conflicting such as minimize write latency and maximize durability, minimize storage cost and maximize read availability, and minimize storage cost and maximize load balancing. However, some constraints and requirements are complementary, such as maximize write availability and maximize load balancing, minimize energy consumption and minimize storage cost, and maximize durability and maximize read availability. These constraints and requirements directly translate to the HRM that match those constraints and requirements. These mappings are derived heuristically and through literature experience. The replication count, based on heuristics from various systems, is a range with MIN=1, MAX=5 and DEFAULT=3. This is configurable by the user at each level. The default values configured in the library are $r_n = 3$, $r_m = async$, $w_c = w_d = w_r = 0.33$. A summary of these translation is provided in Table III. The dash (i.e. "-") symbol in the table is either derived from other features or just defaulted by the system at the end of resolutions.

Requirement	Type	r_n	r_m	w_c	w_d	w_r
Min read latency	B	Max	-	0	1	0
Max read bandwidth	B	-	-	0	0.5	0.5
Max Durability	I	Max	SYNC	-	-	-
Max Load Balance	B	-	-	0.3	0.3	0.3
Min Low Energy	B	Min	-	0	0	1
Constraint						
Min write latency	B	-	ASYNC	1	0	0
Max write bandwidth	B	-	ASYNC	0.5	0	0.5
Min storage	B	Min	-	0	0	1

TABLE III
MAPPING USER REQUIREMENTS TO HREPLICA METRICS

C. Replication Engine

The Engine, HReplica's central brain, is responsible for devising a replication scheme utilizing the compression algorithms and multi-tiered placement for an incoming replication task. The engine receives the input data characteristics along with the replication metrics from the RA, expected compression cost from the CCP, and the current storage status from the MI. It runs a recursive multi-dimensional optimizer to produce one or more sub-tasks for each replication task. The set of created sub-tasks constitute a scheme that is passed to the RE for execution.

1) Algorithm: When a task is received, the Engine recursively matches and places replicas for all combinations of target tier, compression library, and replication strategies. During the calculation of the cost of each sub-problem, the engine pulls the current status of the tier and estimated compression cost from the MI and CCP respectively. The input is first split based on hotness of data. Each spectrum of hot data is split into pieces. Each split piece forms a sub-problem which is then placed into the hierarchy. For every combination of the sub-problem, if the compressed data can fit in an upper layer, then it will be added to the optimization space as a sub-problem. Otherwise, the task will be split in two parts in multiples of 4096 bytes: one that can fit in the remaining capacity of the current tier and one holding the rest of the I/O task. This satisfies constraints $1-3$ of the problem statement described earlier. Our choice of 4096 bytes is motivated from the page-size of RAM and the block size of storage devices. This will avoid unaligned I/O, a known issue in storage [52]. More importantly, however, this choice makes the memoization highly effective as the sub-problems would be reusable. The cost of replication is only calculated for synchronous writes. Hence, if asynchronous replication is enabled, then the cost of writing replicas won't be added to the cost function. The solution can be expressed as a recursive dynamic programming optimization. The mathematical formulation of the cost function $Duration(p, r, c, l)$ is given in equations 1 and 2, consistent with the problem formulation in Section III-B. Additionally, we describe the recursive algorithm of matching in Algorithm 1. The dynamic programming optimization is almost constant with time complexity of $O(2*(len(L))^2)$, where $len(L)$ is very small (in the order of tens). Hence, the time complexity of the replica placement algorithm is practically $O(1)$.

$$Duration(p,l,c) = \begin{cases} Time(i_0,l,c) & r_m = ASYNC \\ \sum_{j=0}^{r_n} Time(i_j,l,c) & otherwise \end{cases} \quad (1)$$

$$Time(i,l,c) = w_c * t_c + t(i,l) - w_r * \frac{t(i,l)*(r_c-1)}{r_c} + w_d * t_d \quad (2)$$

$Duration$ represents the cost of executing a replication scheme for task i to a tier l with a compression library c. l is the index of a tier in the set of all tiers L, where lower values of l represent higher tiers (e.g., $l = 0$ represents DRAM). c represents the index of the compression library in the set of all compression libraries C, with $c = 0$ representing no compression. When $c = 0$ then t_c and t_d equals 0 and r_c equals 1. Also, w_c, w_d, and w_r represent weights for t_c, t_d and r_c respectively. r_m represents the mode of replication and it carries value of ASYNC/SYNC. The number of replicas is represented by r_n. Finally, i_j represents j^{th} replica of task i.

Algorithm 1: Replication Algorithm

```
1  Procedure Calc($p, c, l$)
2      if $l.size \geq c(p).size$ then
4          Place($p, c, l$)
5      else
7          $parts \leftarrow Split(p, c, l.size)$
9          Place($parts[0], c, l$) + Calc($parts[1], c, l.next$)
10     end
11 Procedure Place($p, c, l$)
12     Min($Duration(p,l,c), Calc(p,l+1,c), Calc(p,l,c+1)$)
```

2) Engine Illustration: A visual representation of the Engine is represented in Figure 3. In the default case where the Engine does not satisfy any replication requirement or constraint, we see three data blocks B1 through B3 are placed in the heterogeneous hardware based on the hotness of the data. Now if the replication requirement was to maximize fault tolerance with the same storage cost, HReplica would apply heavy compression on each data block and, hence, would be able to create more replicas in the same storage space. In the given example, applying heavy compression increases the replica count by 2.5x with an increased write cost (due to more replicas being written) of 1.6x. On the other hand, if the requirement was to reduce read access latency while reducing storage cost, HReplica would have different replicas with different compression libraries with at least one replica having no compression. This would ensure the data access latency would be optimized using the uncompressed data whereas storage cost would be reduced by compressing other replicas. This process would have the best latency that can be achieved for the three blocks while also reducing the storage cost by 10%. This example showcases how the replica engine can be dynamically tuned based on different application requirements and constraints in a multi-tiered storage environment.

D. Replica Catalog (RC)

The main role of the RC is to maintain the replica set information for a given data piece. Specifically, the Engine could split data into multiple pieces and each piece could have different replication factor (i.e., number of replicas). For

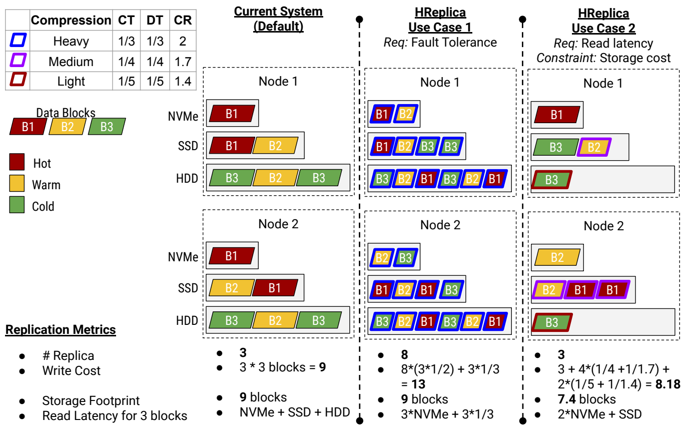


Fig. 3. HReplica Engine Illustration

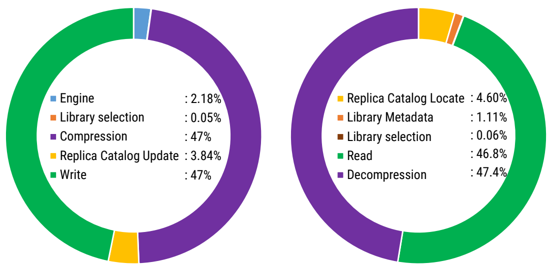
each data piece, the RC maintains a map of a data piece and the locations of its associated replicas in a distributed Hash Map [53]. Upon store, it puts the data piece as the key and the vector of replicas as the value in the Hash Map. Upon a read request, the Replica Catalog pings the Hash Map and gets the associated locations and calculates the distance of each replica to determine the closest one. As the read request might not be aligned, the read could correspond to multiple data pieces and, hence, multiple locations. The Hash Map is crucial for this use case as it provides a uniform and fast O(1) insertion and querying capability, support for concurrent access, fault tolerance in case of power-downs, and low latency.

E. Replication Executor

The Replication Executor (RE) is responsible to execute the replication scheme output from the engine. For a given scheme, the executor first applies the compression algorithm as instructed by the scheme and then places the replicas into the storage. This is fulfilled based on the replication mode, number of replicas selected, and their placement tiers by the Storage Interface. Once the replication is completed, the RC is updated with the location of the replicas. The Compression Manager (CM) provides a unified interface to access all the compression libraries. We utilized a unified compression library framework and integrated it into this interface. The Storage Interface (SI) unifies all I/O calls to all tiers using a simplified interface with APIs `read()` and `write()` using a factory pattern.

F. Implementation

HReplica library's prototype implementation is written in C++ in around 1K lines of code. Additionally, HReplica contains wrappers for C/C++ and Java applications supporting a wide range of applications ranging from scientific computing to Cloud-based software such as Redis, memcached, and Hadoop. The HReplica library can be simply linked to an application (e.g., using `LDFLAGS` or `LDPRELOAD`) to replicate data using its wrapper or can be included as a dynamic link to use its simple APIs. The HReplica library only provides the Engine, DA, RA, and the CCP modules. The rest of the interfaces are implemented as external components to HReplica and should be appropriately implemented based on the storage solution it is used with. For our prototype implementation, we



(a) Write Operation (b) Read Operation

Fig. 4. HReplica anatomy of operations.

use Hermes [34] as a hierarchical storage solution. Hermes uses the HReplica library to perform replication where the storage clients and monitors are provided by Hermes.

V. Evaluation

A. Methodology and Experimental Setup

1) Configurations: We ran all of our experiments on the Ares cluster at the Illinois Institute of Technology [45]. The entire cluster runs on a 40 GBit Ethernet with RoCE capabilities. We configure the buffers, unless specified otherwise, to fit 30% in local NVMe and 70% in burst buffers. This ratio is motivated from the existing hierarchical supercomputer [11] configurations. Cluster specifications are shown in Table IV and Table V shows the configurations tested.

TABLE IV
Testbed Specifications.

Node Type	CPU	RAM	Disk
Compute x64	Intel Xeon Silver 4114 @ 2.20GHz	DDR4 96GB	512GB NVMe SSD
Burst Buffers x4	AMD Dual Opteron 2384 @ 2.7Ghz	DDR3 64GB	2x512GB SSD
Storage x24	AMD Dual Opteron 2384 @ 2.7Ghz	DDR3 32GB	2TB HDD

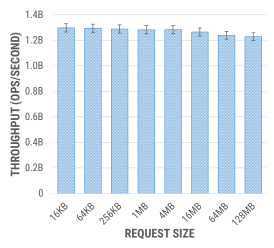
TABLE V
Test Configurations.

Replication Test case	Abbreviation	Heterogeneous	Compression
Baseline	BASE	No	None
Multi-tiered Replication without compression	MTNC	Yes	No
Multi-tiered Replication with single compression	MTSC	Yes	Single
HReplica	HR	Yes	Dynamic

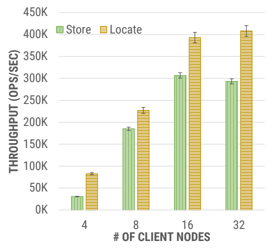
2) Workloads: To evaluate HReplica, we first use micro-benchmark workloads to measure the performance of internal components. These micro-benchmarks perform mixed write and read operations and measure the time taken by the component we are studying. We then calculate other metrics such as throughput based on this time. Additionally, we use real I/O workloads from HPC and Cloud to compare HReplica against current state-of-the-art replication solutions implemented within OrangeFS [46] and Redis [29] respectively. We use VPIC [54] and BD-CATs [55] I/O kernels as HPC workloads to evaluate the data replication engine within OrangeFS and use Yahoo Cloud Services Benchmark (YCSB[1]) to evaluate the data replication engine within Redis Key-Value Store.

3) Performance metrics: We measure the overall time required to perform the replication task in seconds, the number of replicas the engine produced, and the read performance in seconds. We define throughput as the rate of requests processed per second. We executed all tests five times and report the average along with standard deviation.

[1]https://github.com/brianfrankcooper/YCSB

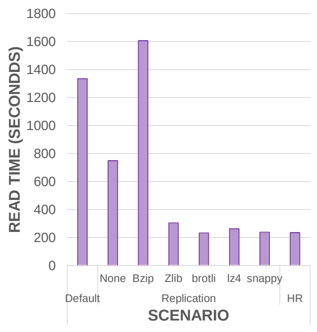


(a) Engine

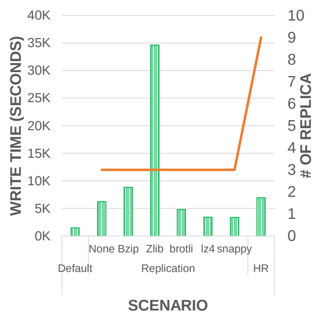


(b) Replica Catalog

Fig. 5. HReplica internal components.



(a) BDCATS with read optimization



(b) VPIC with fault tolerance

Fig. 6. HReplica Optimization

B. HReplica Internal Component Evaluation

1) Overhead Analysis: Each replication task within HReplica is converted by the engine into a replication scheme which contains its replica count, replication destination, and the compression library to be applied. In this test, we want to analyze the relative breakdown of write and read operations with various parts of the HReplica framework to show whether or not the overhead of the engine is greater than its benefits to the storage system. To quantify the overhead, we perform 1K writes and reads of 1MB each and present the overall breakdown into its components as shown in Figure 4 (write 4(a) and read 4(b)). We observe that for both operations 94% of the time is spent on I/O operation or compression/decompression operations. The engine takes about 2% of the overall time whereas the replica catalog takes approximately 4% to record and 4.6% to locate nearest replicas. The rest of the components contribute to about 1% of the overall time. This result shows that the HReplica framework, in practice, has less than 7% overhead for both read and write operations. With this overhead, a storage system can utilize the intelligence present within the engine to dynamically achieve different replication requirements effectively.

2) Replication Engine Performance Analysis: The performance of the algorithm that the engine runs is critical for building the replication scheme within HReplica. Hence, the algorithm should demonstrate high throughput of mapping various replication tasks to their corresponding tiers and compression libraries. To evaluate this, we perform 8K data replication tasks of various sizes and calculate the throughput of the engine. The throughput of the algorithm is shown in Figure 5(a). In this figure, the x-axis represents various task sizes and the y-axis shows the overall throughput (tasks/second). We can observe that until 4MB task size, the throughput of replication engine algorithm is almost constant at about 1.29 billion tasks per second. As the data size increases, the throughput slightly drops (2-5%). This is due to the fact that for bigger tasks, the algorithm has to split the task into several pieces (so that it can fit it into limited capacity tiers) which span across multiple tiers. Overall, this evaluation highlights that the engine's algorithm is very light-weight and has high, constant throughput.

3) Replica Catalog Performance Analysis: The throughput of the Replica Catalog is crucial within a replication framework. The Catalog is responsible for two main operations. First, to store replica information and then find the closest replica to read the data for the client request. Both these operations should demonstrate high throughput. Hence, to measure the capabilities of the catalog, we perform data replication information store and locate operations. Each operation is performed 64K times per client process and we calculate the throughput. Figure 5(b) shows the results. In this figure, the x-axis represents different scales of requests from a client node (each node has 40 client processes) and the y-axis shows the overall throughput achieved in operations per second. We observe that the store operation has a throughput of 305K operations per second which is achieved with 16 client nodes. Additionally, the locate operation reaches a throughput of 410K operations per second with all 32 client nodes. Both operations scale when increasing the clients. Overall, this evaluation demonstrates that the data structure and algorithm used within the Replica Catalog is light-weight and offers high performance.

C. HReplica with Scientific Applications

1) BD-CATS: Scientific applications often involve analyzing huge amounts of data. This process involves parallel reading of data in a repeated pattern. To evaluate the benefit of HReplica for read access optimization, we use BD-CATS-IO to read data produced by a simulation application over eight variables with a data size of 1.2TB. In this test, BD-CATS reads this data over 16 iterations and 2560 processes. We compare HReplica with no replication (shown as baseline), PRS replication without compression (shown as Replication and None), and PRS with several compression libraries. Note, we show the most interesting ones here. The data was written with analysis in mind, hence the replication prioritized fast decompression time and high compression ratio.

Figure 6(a) shows the results of this test. As it can be seen, the read time for BD-CATS-IO without replication is 1334 seconds. When we introduce data replication (PRS algorithm) without compression, the read time gets reduced by 1.78x. Applying compression for replication could reduce the baseline read time by 3.1x (as in case of Snappy and LZ4). However, we see Bzip hurts the overall read time as it has high decompression time (i.e., 2.14x slower than no compression). Hence, it is crucial to apply compression intelligently to optimize read time while considering the decompression time. This is achieved dynamically by HReplica as it considers both compression ratio (which optimizes read I/O) and decompression time. Ultimately, HReplica achieves 3.1x performance.

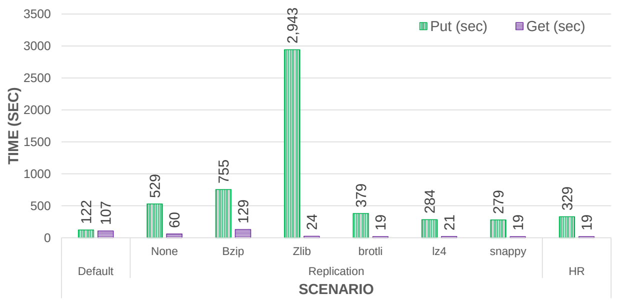


Fig. 7. Redis with read optimization

2) VPIC: Scientific applications produce huge amounts of critical data in the form of simulations, observations, modeling, etc. These applications require fault-tolerant semantics to safeguard the data from unforeseen failures/corruption in the system. To represent this class of application workload, we use VPIC: a large-scale general purpose particle-in-cell simulation where each MPI process produces 8 variables for 8 million particles totalling to a size of 32MB per iteration. This process is repeated over 16 iterations with 2560 MPI processes. The total dataset generated in the form of HDF5 files is 1.2TB in size. We compare HReplica with no replication (shown as baseline), PRS replication without compression (shown as Replication and None), and PRS with several compression libraries. The data is written with high fault tolerance. Hence, HReplica is configured to increase fault tolerance while minimizing write cost.

Figure 6(b) shows the results of this test. We observe the baseline time without replication is 1487 seconds to write the simulation data into the parallel file system. If we apply data replication, we achieve a replication factor of 3 with an increased write cost of 6232 seconds (approx. 4.2x). If we apply compression with PRS algorithm, for the same replication factor, we can achieve 2x faster write performance than PRS using `Snappy` compression. However, for Zlib, the write time increases by 5.5x due to high compression time. HReplica for the same storage cost with no compression can achieve a replication factor of 9 (i.e., 3x more replicas) while having similar write cost with PRS. This test case shows the prioritization of the HReplica engine can be dynamically changed to suit the replication requirement in hand.

D. HReplica with Key Value Store

1) Redis: Cloud applications use Key-Value Stores (KVS) for storing data with a flat namespace. A popular example of this use case is Internet of Things (IoT) devices that produce sensor information (e.g., temperature of each room in a building) and store that information in a key-value store. KVS are often distributed across geographically separated locations, as these sensors are present all over the world. To increase availability of reading temperature data from across the world, KVS utilize data replication. We use Redis as a suitable representative for these KVS and plugin HReplica as a dynamic replication engine to provide a tunable replication requirement within Redis. To evaluate this case, we use YCSB to run a benchmark on a 64 node Redis cluster with heterogeneous hardware of NVMe, SSD, and HDD. We run the read after write workload where each process of the YCSB benchmark makes 8K requests of size 64KB. We use a total of 32 client nodes. We compare HReplica with no replication (shown as baseline), Redis replication without compression (shown as Replication and None), and Redis with several compression libraries. We prioritize read optimization for HReplica.

Figure 7 shows the results of this test. Without replication YCSB takes 122 seconds to write objects into Redis with a read cost of 107 seconds. With Redis' default replication (3 replicas) the write cost increases by 3.8x while reducing read cost by 1.68x. With data compression enabled, this read performance is further improved by 4.8x. However, some compression libraries do not benefit read performance due to high decompression time (e.g., Bzip which slows read performance by 20%). This is automatically balanced by HReplica which uses the appropriate compression libraries (i.e., snappy in this case) to improve the overall read performance by 5.3x.

VI. Conclusions and Future Work

Modern applications are highly data-intensive and I/O is often the bottleneck in their performance. Storage systems utilize data replication as a mechanism to improve read performance through data availability or to increase the fault tolerance of the system. Different replication schemes enable different replication requirements and constraints. As the systems are becoming multi-tenant, we require a dynamic replication scheme which can support multiple conflicting replication requirements while utilizing smart data compression to achieve its goals. In this work, we present HReplica, a dynamic data replication engine with adaptive compression for multi-tiered storage environments. HReplica optimally utilizes heterogeneous storage and smart data compression to enable tunable replication requirements while satisfying various replication constraints dynamically. We introduced a low-cost replication engine which can optimally solve this multi-dimensional problem efficiently. Specifically, HReplica can optimize read performance for data analytic applications such as BD-CATS by 5.2x. The same engine when tuned for fault-tolerance can increase replication factor by 3x for VPIC. We also demonstrated its benefits in a cloud environment by improving the replication engine of Redis by 5.2x for read performance.

We plan to incorporate this technology into the Hermes ecosystem and offer it as a readily available data replication engine to users. HReplica will be open sourced after further rigorous testing and we plan to extend its support to more storage backends.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant no. OCI-1835764 and CSR-1814872.

References

[1] J. Borrill, "Madcap-the microwave anisotropy dataset computational analysis package," *arXiv preprint astro-ph/9911389*, 1999.

[2] R. Latham, C. Daley, W.-k. Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary, "A case study for scientific i/o: improving the flash astrophysics code," *Computational Science & Discovery*, vol. 5, no. 1, p. 015001, 2012.

[3] C. Nieter, J. Cary, D. Smithe, P. Stoltz, and G. Werner, "Simulations of electron effects in superconducting cavities with the vorpal code," in *10th European Particle Accelerator Conference*. Citeseer, 2006, pp. 2269–2271.

[4] S. Gorbunov, D. Rohr, K. Aamodt, T. Alt, H. Appelshauser, A. Arend, M. Bach, B. Becker, S. Bottger, T. Breitner *et al.*, "Alice hlt high speed tracking on gpu," *IEEE Transactions on Nuclear Science*, vol. 58, no. 4, pp. 1845–1851, 2011.

[5] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa, "Science clouds: Early experiences in cloud computing for scientific applications," *Cloud computing and applications*, vol. 2008, pp. 825–830, 2008.

[6] H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman, "Simulation of dynamic data replication strategies in data grids," in *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 2003, pp. 10–pp.

[7] J. Carter, J. Borrill, and L. Oliker, "Performance characteristics of a cosmology package on leading hpc architectures," in *International Conference on High-Performance Computing*. Springer, 2004, pp. 176–188.

[8] S.-Q. Long, Y.-L. Zhao, and W. Chen, "Morm: A multi-objective optimized replication management strategy for cloud storage cluster," *Journal of Systems Architecture*, vol. 60, no. 2, pp. 234–244, 2014.

[9] X. Bai, H. Jin, X. Liao, X. Shi, and Z. Shao, "Rtrm: A response time-based replica management strategy for cloud storage system," in *International Conference on Grid and Pervasive Computing*. Springer, 2013, pp. 124–133.

[10] Y. Qu and N. Xiong, "Rfh: A resilient, fault-tolerant and high-efficient replication algorithm for distributed cloud storage," in *2012 41st International Conference on Parallel Processing*. IEEE, 2012, pp. 520–529.

[11] CRAY, "Cray datawarp applications i/o accelerator," 8 2016. [Online]. Available: https://www.cray.com/products/storage/datawarp

[12] O. R. N. Laboratory, "Oak ridge national laboratory's 200 petaflop supercomputer," 1 2020. [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[13] K. Oh, A. Raghavan, A. Chandra, and J. Weissman, "Redefining data locality for cross-data center storage," in *Proceedings of the 2nd International Workshop on Software-Defined Ecosystems*, 2015, pp. 15–22.

[14] W. Li, Y. Yang, and D. Yuan, "A novel cost-effective dynamic data replication strategy for reliability in cloud data centres," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 2011, pp. 496–502.

[15] N. K. Gill and S. Singh, "A dynamic, cost-aware, optimized data replication strategy for heterogeneous cloud data centers," *Future Generation Computer Systems*, vol. 65, pp. 10–32, 2016.

[16] J. Zhou, Y. Chen, W. Xie, D. Dai, S. He, and W. Wang, "Prs: A pattern-directed replication scheme for heterogeneous object-based storage," *IEEE Transactions on Computers*, 2019.

[17] R. Sears, M. Callaghan, and E. Brewer, "Rose: Compressed, log-structured replication," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 526–537, 2008.

[18] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization," in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies-Volume 4*. USENIX Association, 2005, pp. 21–21.

[19] H. Devarajan, A. Kougkas, and X.-H. Sun, "An Intelligent, Adaptive, and Flexible Data Compression Framework," in *Proceedings of the IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid'19)*. Larnaca, Cyprus: IEEE, 2019.

[20] L. Qiu, V. N. Padmanabhan, and G. M. Voelker, "On the placement of web server replicas," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, vol. 3. IEEE, 2001, pp. 1587–1596.

[21] A. Aazami, S. Ghandeharizadeh, and T. Helmi, "Near optimal number of replicas for continuous media in ad-hoc networks of wireless devices." in *Multimedia Information Systems*, 2004, pp. 40–49.

[22] N. J. Navimipour and F. S. Milani, "Task scheduling in the cloud computing based on the cuckoo search algorithm," *International Journal of Modeling and Optimization*, vol. 5, no. 1, p. 44, 2015.

[23] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM, 2000, pp. 56–67.

[24] B. Tang, H. Gupta, and S. R. Das, "Benefit-based data caching in ad hoc networks," *IEEE transactions on Mobile Computing*, vol. 7, no. 3, pp. 289–304, 2008.

[25] S. Jin and L. Wang, "Content and service replication strategies in multi-hop wireless mesh networks," in *Proceedings of the 8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*. ACM, 2005, pp. 79–86.

[26] S. Goel and R. Buyya, "Data replication strategies in wide-area distributed systems," in *Enterprise service computing: from concept to deployment*. IGI Global, 2007, pp. 211–241.

[27] N. Ahmad, A. A. C. Fauzi, R. M. Sidek, N. M. Zin, and A. H. Beg, "Lowest data replication storage of binary vote assignment data grid," in *International Conference on Networked Digital Technologies*. Springer, 2010, pp. 466–473.

[28] T. . Fraunhofer, "Beegfs, the parallel cluster file system," 1 2020. [Online]. Available: https://www.beegfs.io/content/

[29] R. Labs, "Redis key value store," 1 2020. [Online]. Available: https://redis.io/

[30] M. Inc., "Mongodb the database for modern applications," 7 2020. [Online]. Available: https://www.mongodb.com/

[31] A. Kafka, "Apache kafka, a distributed streaming platform," 7 2020. [Online]. Available: https://kafka.apache.org/

[32] A. BookKeeper, "Apache bookkeeper, a scalable, fault-tolerant, and low-latency storage service optimized for real-time workloads," 7 2020. [Online]. Available: https://bookkeeper.apache.org/

[33] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–9.

[34] A. Kougkas, H. Devarajan, and X.-H. Sun, "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, 2018, pp. 219–230.

[35] O. A.-H. Hassan, L. Ramaswamy, J. Miller, K. Rasheed, and E. R. Canfield, "Replication in overlay networks: A multi-objective optimization approach," in *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Springer, 2008, pp. 512–528.

[36] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," 2003.

[37] Z. Zeng and B. Veeravalli, "Optimal metadata replications and request balancing strategy on cloud data centers," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2934–2940, 2014.

[38] T. Chen, R. Bahsoon, and A.-R. H. Tawil, "Scalable service-oriented replication with flexible consistency guarantee in the cloud," *Information Sciences*, vol. 264, pp. 349–370, 2014.

[39] D.-W. Sun, G.-R. Chang, S. Gao, L.-Z. Jin, and X.-W. Wang, "Modeling a dynamic data replication strategy to increase system availability in cloud computing environments," *Journal of computer science and technology*, vol. 27, no. 2, pp. 256–272, 2012.

[40] J.-W. Lin, C.-H. Chen, and J. M. Chang, "Qos-aware data replication for data-intensive applications in cloud computing systems," *IEEE Transactions on Cloud Computing*, vol. 1, no. 1, pp. 101–115, 2013.

[41] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng, "Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster," in *2010 IEEE international conference on cluster computing*. IEEE, 2010, pp. 188–196.

[42] N. K. Gill and S. Singh, "Dynamic cost-aware re-replication and rebalancing strategy in cloud system," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Springer, 2015, pp. 39–47.

[43] A. Cidon, R. Stutsman, S. Rumble, S. Katti, J. Ousterhout, and M. Rosenblum, "Mincopysets: Derandomizing replication in cloud storage," in *Proc. 10th USENIX Symp. NSDI*, 2013, pp. 1–5.

[44] J. Liu and H. Shen, "A popularity-aware cost-effective replication scheme for high data durability in cloud storage," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 384–389.

[45] Scalable Computing Lab, Illinois Tech, "Ares Supercomputer @ IIT," 2019. [Online]. Available: http://www.cs.iit.edu/~scs/resources.html

[46] OrangeFS, "OrangeFs," 2019. [Online]. Available: http://orangefs.com/

[47] M. A. Roth and S. J. Van Horn, "Database compression," *ACM Sigmod Record*, vol. 22, no. 3, pp. 31–39, 1993.

[48] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[49] H. Devarajan, A. Kougkas, and X.-H. Sun, "Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 62–72.

[50] R. Gribonval, V. Cevher, and M. E. Davies, "Compressible distributions for high-dimensional statistics," *IEEE Transactions on Information Theory*, vol. 58, no. 8, pp. 5016–5034, 2012.

[51] H. Devarajan, A. Kougkas, L. Logan, and X.-H. Sun, "Hcompress: Hierarchical data compression for multi-tiered storage environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020.

[52] X. Zhang, K. Liu, K. Davis, and S. Jiang, "ibridge: Improving unaligned parallel file access with solid-state drives," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 381–392.

[53] H. Devarajan and C. Hogan, "HCL: Hermes Container Library," 2019. [Online]. Available: https://bitbucket.org/scs-io/hcl

[54] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter *et al.*, "Parallel I/O, analysis, and visualization of a trillion particle simulation," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–12.

[55] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, P. Dubey *et al.*, "BD-CATS: big data clustering at trillion particle scale," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.