SSD-Optimized Workload Placement with Real-time Learning in HPC Environments

***Abstract*—As the market becomes mature, SSD drives have emerged as a viable alternative storage hardware for conventional hard drives due to their high speed and decreasing cost. However, the durability of SSD chips are still are considerably limited and thus, pure SSD based solutions have not been viable as a general storage device. In this paper, we focus on the problem of workload and storage placement in HPC environments, and propose a hybrid solution, namely SSD-Crush, where SSD and hard drives are used together as the hardware for the Crush algorithm, which is a very widely used algorithm for workload placement and storage distribution. Based on an analytical model, we propose an optimized scheme that places workload on storage units provided by SSD or hard drives such that the performance or lifetime may be optimized. The scheme is also enhanced with an adaptive learning algorithm where machine learning techniques are employed to find "shortcuts" in runtime algorithm performance. We present experiments based on both a simulator and an Amazon-EC2 powered testbed to show that the analytic model approach can dynamically adjust storage placement as workloads evolve to enhance performance or lifetime.**

I. Introduction

Providing resilient and fast storage service for HPC applications remains a critical challenge given the large demands on storage space as well as the possibility of drive failures. Therefore, it is crucial to schedule workloads properly among multiple drives based on the nature of workloads as well as the properties of the underlying hardware. On the other hand, both the workloads and the underlying hardware evolve over time, meaning that algorithms have to be designed to take these changes into their consideration.

One particular trend that has emerged in recent years is the use of SSD drives in storage to provide *premium* services, as the reading and writing speed for SSD drives are typically faster compared to the hard drives. On the other hand, SSD drives also have more different failure patterns as repeated reading and writing of the same blocks will cause the drives to fail. How to integrate SSD drives successfully into the design of a storage area network becomes a challenge that modern HPC applications have to handle.

Existing algorithms to integrate SSD drives can already be found in the literature []. These methods, while effective, are largely based on heuristic algorithm designs that are either developed in isolation with the runtime workload, or are based on static assumptions on the workload patterns, making them unsuitable when the underlying parameters change over time. Furthermore, they do not consider drive failures, which has considerable impact when underlying hardware becomes unavailable.

We note that these identified problems could not be solved easily if the algorithms do not self-adapt themselves. The fundamental reason is that new challenges will arise that lie outside the scope of the original solution, making them unsuitable eventually. Therefore, we aim to develop an algorithm that can self-adapt to the underlying hardware and software changes. Our work is based on the idea of allocating objects based on their predicted access frequency. For those objects that are most frequently accessed, we move them to the SSD-powered drives so that their access latency can be minimized. This work serves as an extension to the classical CRUSH algorithm. Furthermore, our work is enabled by recent progress on machine learning techniques. Specifically, we develop an improvement based on the CRUSH algorithm that has been widely used for data storage placement in high performance computing platforms. In this improvement algorithm, we take into account the additional types of underlying hardware platforms by modeling them in a parametric manner, which we then calculate the most efficient manner of replica deployment. Following the initial deployment, we need to periodically recalculate the deployment when something changes. In such scenarios, we will reduce the time for re-calculation by classifying on the types of changes, and apply the changes based on the previously learnt strategies. This way, we no longer need to re-calculate the whole changes every time something is modified.

Our developed algorithm has the following assumptions. First, we consider that the storage hardware consists of both conventional hard drives and SSD drives. SSD drives are typically smaller in capacity, but faster in terms of speed. Hard drives and SSD drives have different parameters for failures. Second, the user may request changes to the placement of replications for data, as well as the number of replications, over the workload lifetime. We assume such changes could come in a high rate, leading to rapid replacement changes to be generated. Our developed system is based on these assumptions. The whole developed algorithm serves as an improvement deployed on the CRUSH algorithm running for the Cerph file system. We deploy the system on Amazon EC2 for practical deployment and testing purposes.

The key contributions of the developed algorithm is listed as follows:

First, we present an access-frequency sensitive algorithm for extending the CRUSH algorithm. The core methodology involves modeling the different storage services based on their access latency parameters, and try to minimize the overall access time through an optimization problem.

Second, we observe that given that users' request patterns change over time, it is time-consuming to re-solve the problem every time the user input changes. Therefore, we develop a learning algorithm to classify the input patterns, and generate the output parameters automatically after the training phase.

This approach allows the extension of CRUSH to handle different application requirements highly efficiently.

Third, we systematically evaluate the algorithm on Amazon EC2 and a simulator for performance improvements.

The rest of this paper is organized as follows. We describe the related work in Section II. The design is described in Section III. The system analysis is described in Section **??**. The performance evaluation is given in Section **??**. The application case study is given in Section **??**. We provide conclusions in Section **??**.

II. Related Work

In this section, we are going to provide a brief introduction on several existing works which are related to our paper. We classify these existing works into two categories. The first category consists of existing works on data placement algorithms for distributed storage systems, while the second consists of those on hybrid storage systems which aim to leverage SSD drives to improve data access performance.

As large-scale distributed storage systems have been extensively used in HPC area, the problem of distributing several petabytes of data among hundreds or thousands of storage devices becomes more and more critical. To address this problem, many data placement algorithms have been proposed. For instance, Distributed Hash Tables (DHTs) have been used to place and locate data objects in P2P systems [] [] []. Another replica placement scheme called chain placement was also proposed and applied to some P2P and LAN storage systems [] [] []. Honicky and Miller presented a family of algorithms named RUSH [] which utilizes a mapping function to evenly map replicated objects to a scalable collection of storage devices and meanwhile supports the efficient addition and removal of weighted devices.

To resolve the reliability and replication issues RUSH algorithms suffered in practice usage, Weil et al. proposed a scalable pseudo-random data distribution algorithm named CRUSH [] upon which our algorithm is based. Besides optimally distributing data to available resources and efficiently reorganizing data after adding or removing storage devices, CRUSH exploits flexible constrains on replica placement to maximize the data safety in the situation of hardware failures. CRUSH algorithm allows the administrator to assign different weights to storage devices so that administrator can control the relative amount of data each device is responsible for storing. However, the device weights used in CRUSH algorithm only reflect the capacities of storage devices, which means CRUSH algorithm might not be effective anymore for hybrid storage systems consists of both SSD and HDD devices since these two kinds of storage devices have totally different performance characteristics.

Existing works on hybrid storage systems usually concentrate on two major directions: either using a small amount of SSD-based storage as a cache between main memory and hard disk drives or integrating SSD and HDD together to form a hybrid storage system. Because of SSD devices' high cost and small capacity, the cache-based solution is understandable and common. For example, Srinivasan et al. designed a block-level cache named Flashcache [] between DRAM and hard disks using SSD devices. Zhang et al. proposed iTransformer [] which employs a small SSD to schedule requests for the data on disks so that high disk efficiency can be achieved. SieveStore [] adopts a selective caching approach in which the access count of each block is tracked and the most popular block is cached in SSD device.

Recently, efforts have been made to combine SSD and HDD together to construct a hybrid storage system in which SSD is used as a storage device same as HDD rather than a cache. Chen et al. designed and implemented a high performance hybrid storage system named Hystor, which identifies data blocks that either can result in long latencies or are semantically critical on hard disks and store them in SSDs for future accesses. In order to reduce random write traffic to SSD, Ren et al. proposed I_CASH [] which exploits the spacial locality of data access and only store seldom-changed reference data blocks on SSD. Besides, ComboDrive [] concatenates SSD and HDD into one address space via a hardware-based solution so that certain data on HDD can be moved into the faster SSD space.

There are two main differences between existing works on hybrid storage systems and our approach: First, most existing works on hybrid storage systems only consider how to improve the utilization of SSD devices while they ignore the reliability and replication issues that data placement algorithms, such as CRUSH, try to resolve in HPC environment; Second, our approach tries to monitor the usage of data blocks and store performance critical data on SSD devices which is kind of similar to several existing works, like [] [], but beyond this, our approach applies machine learning algorithm to predict the future usage of data blocks and store potential critical data on SSD in advance.

III. Algorithm Design

In this section, we introduce the design of the algorithm. We first present the problem formulation. Then, we present an overview of its structure. Finally we present a detailed description of its components and related algorithms.

A. The Problem Formulation

The key problem we are considering for the storage provisioning is to develop a sufficient, yet not wasteful, set of storage allocations from existing hardware for objects to be stored, so that applications can meet their desired performance. This problem of storage allocation is challenging due to: (1) the dynamics of workload, (2) the requirement of users, e.g., no replicas should be stored on the same rack for reliability, etc. and 3) difficulty and overhead of deriving the storage allocation results for each workload and user requirement combination. As a result, it is difficult to determine the storage allocation that will achieve the desired quality of service while minimizing the hardware cost and storage overhead. Therefore, if we model such a problem as a search problem whose solution is from a very large search space, then it is very hard to always reach the optimal configuration when something changes, even such change could be slight.

In most cases we can resort to offline algorithms, where the results are calculated and used for a rather long period of time in storage services. This of course leads to underwhelming

performance and storage waste. The key issue with existing solutions is that they do not handle dynamics in the requests or workloads successfully. Moreover, on-line algorithms may not be fully predictable in their convergence time, leading to long delays and out-of-date solutions. This problem may become even worse when different types of hardware such as SSD is used, or servers may be added or removed due to failures, and so on.

When faced with such unsatisfactory performance, we formulate the following problem: **can we develop a method to adapt in runtime with the help of learning based on the past decisions?** If this is feasible, then we can avoid the re-calculation when a similar scenario is met in practice.

B. Architecture Overview

Our proposed algorithm works together with the storage allocator like the CRUSH algorithm. In fact, it is deployed alongside with the existing algorithms, and watches their operations carefully. Although we can use other types of algorithms, we focus on the case that the CRUSH allocation algorithm is used. The learning algorithm accelerates the management of storage resources in HPC platforms by caching the decisions made by past CRUSH operation, and tries to reuse them if it faces the same or a similar workload or user request. For this algorithm to effectively dealing with dynamic workloads, it first has to learn about workloads/user request combinations, and their associated allocation decisions (e.g., how many storage is allocated on each type of hardware on different racks) during the training phase (e.g., a week of training use). To profile a workload, this algorithm then adopts a profiling algorithm based on the machine learning techniques to compute the workload/request signature for each encountered input. The signature itself is a set of automatically chosen performance and characteristic metrics, such as what the requirements are, what the workload distribution is, and so on. Then the algorithm tries to look up whether a previous resource allocation can be used.

After the training phase, our algorithm will profile the workload periodically or on-demand using for the operations of CRUSH. It then uses each computed signature to automatically classify the encountered workload. If the classifier encounters a similar workload as before, (based on similarity), the algorithm simply applies the previously computed storage allocation result. If the input is new, the algorithm may decide to take the new calculation result into training, by simply running an existing state-of-the-art algorithm to make online decisions.

C. Design Assumptions

We now present our design assumptions. We assume that the user of the service deploys their service across a pool of storage servers. The storage servers could be a set of OSDs, which are intelligent controllers that can map the requests from the user to the storage services. Each of the OSD may provide some level of SLO (Service Level Objective) for the deployed service.

During the system's operation, we assume that users' requests include both reading and writing operations. Note that the writing operations may be dominating for certain workflows, such as checkpoint backups that require periodic operations. The users' workloads may also change over time, which requires that the system should adapt to the requirements and demands.

D. Parametric Workload Profiling

As the key idea of our approach is by caching results of past storage decisions and quickly reusing them once a similar workload demand is met later. Therefore, for CRUSH-SSD to be effective in dealing with dynamic workloads, it first needs to learn about the workloads and their associated storage decisions in the past during the learning phase. To profile a workload, CRUSH-SSD algorithm uses a profiler that is deployed along the path of the users' requests. The profiler serves requests in the profiling environment, allowing us to collect all metrics that are needed for the characterization process. Note that such profiling is occurring independently of the OSD servers that handle such requests concurrently.

Note that the advantage of having the profiler operating independently of the OSDs lies in that this does not require our profiler to be tightly integrated into the rest of the system. Therefore, the profiler can be easily replaced as needed, which gives us additional flexibility.

Our next task is to pick the workload signature properly. For the profiling to be most effective, it is critical to pick a set of metrics that form a unique *signature* that can uniquely identify all types of workload behaviors. We next discuss how we can collect such workload describing metrics.

Ideally, the collected metrics for the incoming workloads should allow us to uniquely identify the workloads without requiring knowledge about the details of the workloads themselves. Leveraging these low-level metrics, however, raises another question: given the complex nature of the workloads, can we rely on these metrics as reliable signatures to distinguish between different workloads? This task is a feature selection procedure where we aim to select a set of most relevant features for workload prediction. In our evaluation section, we will evaluate this question through realistic measurements based studies.

E. Learning based Workload Classification

Once the features are selected, we then collect training data for a period of time. The task then becomes a classification problem, where we need to evaluate the effect of selected features on classification accuracy. As the problem has been studied in machine learning literature for many years, instead of developing new models, we simply apply various mature methods in the machine learning domain to the dataset to perform the profiling task.

More specifically, suppose that we select N-tuple that forms a workload signature as:

$$Signature = m^1, m^2, ..., m^n$$

where the m_i represents a measured metric of i. Note that the metric selection process is fully automated and transparent to the user.

Once the signatures are collected, it is important that we can classify later workloads based on the training periods. This is based on the assumption that user workloads will have inherent repeating patterns, where similar workloads may be encountered again and again.

IV. Analytical Modeling of Algorithm Performance

In this section, we model the system performance using analytic approaches.

V. System Evaluation

A. Evaluation Overview

In this section, we systematically present the evaluation of the proposed algorithm. We implement the algorithm in both simulations and a testbed on the Amazon EC2 platform. We focus on its performance metrics including the throughput, reading latency, replication ratio, and availability. Details are presented below.

Metrics definitions and goals

B. Setup of Evaluations

The configuration details of the evaluation setup.

C. Evaluation Results

We plot the results in the following figures.

VI. Application Case Study

A. Problem Overview

Describes the workload problem

B. Details

Describes the details of the implementation for this application case study.

VII. Conclusions