F2PUnifyCR: A Flash-friendly Persistent Burst-Buffer File System

ThanOS
Department of Computer Science
Florida State University
Tallahassee, United States

I. Abstract

With the increased amount of supercomputing power, it is now possible to work with large scale data that pose a continuous opportunity for exascale computing that puts immense pressure on underlying persistent data storage. Burst buffers, a distributed array of node-local persistent flash storage devices deployed on most of the leardership supercomputers, are means to efficiently handling the bursty I/O invoked through cutting-edge scientific applications. In order to manage these burst buffers, many ephemeral user level file system solutions, like UnifyCR, are present in the research and industry arena. Because of the intrinsic nature of the flash devices due to background processing overhead, like Garbage Collection, peak write bandwidth is hard to get. In this work, we aim to pinpoint the challenges through experimental evaluation and practice the potentials of research in managing those challenges. We perform thorough experimental studies on the efficiency of the burst buffers with UnifyCR, including impact of garbage collection (GC), necessity of persistence in handling those aforementioned challenges. We provide an empirical study on the effect of GC and necessity of persistence and incorporate flash-friendly and data persistence features for the efficiency and bandwidth of UnifyCR.

II. Introduction

With a view to handling and facilitating the bursty I/O of scientific applications that leverages the resources of large-scale supercomputers, deploying burst buffers between the compute nodes and the backend PFS has been thoroughly researched and applied by researchers worldwide. Burst buffers can be deployed by attaching fast storage devices locally to each compute node (node-local burst buffer) or it can be incorporated by providing an additional layer of fast storage devices that can be remotely shared by the compute nodes (remote or shared burst buffers). The usage of burst buffers can be manifold depending on the workloads it is handling for applications. In order to leverage the capabilities of burst buffers to the utmost level, it is very important to have a standardized software interface across systems. It has to deal with an immense amount of data during the runtime of the applications.

Using node-local burst buffer can achieve scalable write bandwidth as it lets each process write to the local flash drive, but when the files are shared across many processes, it puts the management of metadata and object data of the files under huge challenge. In order to handle all the challenges posed by the bursty and random I/O requests by the Scientific Applications running on leadership Supercomputing clusters, Lawrence Livermore National Laboratory (LLNL) has been actively developing an unified file system named UnifyCR [1]. UnifyCR is a user level file system which can use node-local flash storage devices as burst buffers for distributed and shared files. By leveraging UnifyCR, applications can write to fast, scalable, node-local burst buffers as easily as they do the parallel file system to support both checkpoint/restart which is the most important I/O workload for HPC.

UnifyCR is built on top of the implementation of an ephemeral burst buffer file system named BurstFS [2] developed by Computer Architecture and SysTems Research Lab (CASTL) [3] at Florida State University (FSU) in collaboration with LLNL. It has the capabilities of efficiently handling metadata by it metadata management module named MetaKV [4] and satisfying random read requests made by research and industry standard scientific applications. This file system is called ephemeral because it stays alive until the application runs, unmounts itself when the application completes, and eventually flushes the results and checkpoint information to the underlying persistent file systems. Hence, there is an opportunity of working on improving the object data handling of the files kept on burst buffer in an organized

and flash-friendly manner by using a special file system like Flash-Friendly File Sytem (F2FS) [5]. Moreover, for effecient and cost effective use of the SSD devices while writing, the file system can be equipped with an efficient pipeline for asynchronously flushing the data from burst buffer file system to the underlying persistent file system.

In summary, our project makes the following contributions.

- We perform an in-depth background studies and include those in the literature.
- We add an efficient garbage collection mechanism for the flash devices of the system by incorporating F2FS.
- We develop a dedicated pipeline for asynchronously flushing the data on the SSD devices of UnifyCR to the underlying persistent file system of SSD or HDD.
- We run thorough evaluation of the performance of an established UnifyCR after incorporating our strategies in it.

III. Background

We organize the background knowledge necessary to properly grasp the idea of the project in this section. We arrange the knowledge base into three basic categories, Hardware Level, Kernel Level and User Level.

A. Hardware Level

In this section, we discuss about the hardware level basics that we explore to possess a good perception on the project. To contrast the characteristics and basic mechanisms in conventional rotatiing harddisk drives (HDD) and Solid State Drives (SSD), we depict brief on both HDD and SSD devices. We explore several storage devices and try to have a grasp on how the writing works on each category of devices.

a) Harddisk Drive (HDD): Hardisk Drives (HDDs) work through megnatic spinning disks or platters to read and write data. They consist of one or more platters, an actuator arm and a motor. The megnatically sensitive platters store the data through megnatic orientation, the actuator arm has a read/write head for each platter, and the motor helps to spin the platters and move the arms. In addition, there are I/O controller and firmware that control the hardware and manage the device functionalities and communication among the modules. Each platter consists of tracks. Sectors are the logical division of data into tracks. To construct an address, track and sector numbers are used like a 2-dimensional matrix. This makes the locating of data in a deterministic manner. When a write request is invoked, the I/O controller searches for a nearest available space and writes the data on it. Besides, the firmware detects and corrects the error in the data using different algorithm involving data parity. There is a predefined speed of platters' rotation, i.e. 4200 to 7200 rpm. The read/write rates are directly dependent on this spinning speed. The relation between read/write through the disk is directly proportional to the spinning angular speed. The modification and retrieval of data depends on the functionality of the I/O controller. The controller commands the actuator arm to move to the specific location detected by the sector and track number. The head reads or writes data by detecting the presence or absence of a charge created by certain megnatic orientation. Following a request to update data, the head modifies the megnatic orientation of the atoms in the tracks. In case of data modification, the HDDs can directly overwrite the data in a particular region, rather than keeping the invalid old data. Hence, there is no need for Garbage Collection (GC) in HDDs.

However, there are some limitations in reading and writing in HDDs that are directly related to the usage of mechanical parts in the system. The latency and bandwidth of the data retrieval and update depends on the physical movement of the arms to detect the specific region and changing the megnatic orientation. Besides, HDDs consume more energy than SSDs. Even though HDDs have some drawbacks, these are less expensive and well-proven technology, and can be used to store mass data.

b) Solid State Drive(SSD) and Garbage Collection (GC): Solid State Drives (SSD) provides with storage made of solid state flash memory made out of NOR or NAND gates. NAND flash devices are the most popular ones in this case. It is called as Solid State not only because of the use of solid state electronic instead of mechanics in HDD, but also beacuse it does not have any moving part in it. An SSD is a digital storage device that features multiple interconnected flash memory sticks. SSD features no moving parts. In SSDs, read and write speeds are faster. In addition, total storage capacity is greater reliability is greatly improved over standard thumb drives. SSDs can be considered as large USB drives as mainly use the same basic technology. The technology behind the solid-state drives is NAND which is a kind of flash memory. If we think at the basic lowest level, we would find that floating gate transistors record a charge (or lack of charge) to store data. The gates make a grid pattern that further organized into a varying sizes block. Each row that makes up the grid

is called a page. SSD controller does several functions, such as keeping track data location, reading and writing to retrieve or update data. To retrieve or update data, the SSD controller looks at the address of the data requested and reads the charge status. However, updating data is very complex in SSDs. For example, all the data in a block must be refreshed when any portion of the block gets updated. The data on the old block is copied to a different block, then the block is erased, after that the data gets rewritten with the changes to a new block. Unlike HDDs, here in SSDs, when the drive is idle, a process called garbage collection works through and makes sure the information in the old block is erased and that the block is free to be written to again. There is another process termed as TRIM which informs the SSD that it can skip rewriting certain data when it erases blocks. As in SSDs, there are a finite number of times any block can be rewritten. Because of these reasons, TRIM is an important process that prevents premature wear on the storage drive. Furthermore, to prevent the wear, each block in the drive gets an equal amount of read/write processes by a specific algorithm. This process is called wear leveling and happens automatically as the drive is working. Another aspect, as the read/write process requires data movement, SSDs are usually over-provisioned with storage. In this case, there is always a certain amount of the drive that is not reported to the OS as well as not accessible to the user. This actually allows space for the drive to move and delete items avoiding any effect to the overall storage capacity. Even though there computationally there are several advantages of SSDs, however they have also some notable drawbacks. Firstly, SSDs are newer technology, less matured and are more expensive than HDDs. Sometimes according to the need, it can be very harder to find very large-capacity solid state drives. However, the biggest advantage of a solid-state drive is its simplicity. There are no moving parts, which means there's very little to break. This makes SSDs incredibly reliable and robust. The other advantage is performance. Because there is no spinning disc, read and write speeds are incredibly fast and the risk of data fragmentation gets eliminated. Typically, a solid-state drive will outperform a hard drive in most situations. Whereas an HDD may require a minute or two to fully boot, an SSD can boot in seconds. As a whole, solid state drives are faster to give response and they are lighter as well as more able to withstand being moved and dropped. Furthermore, compared to HDDs, solid state drives use less energy, keeping the computer cooler.

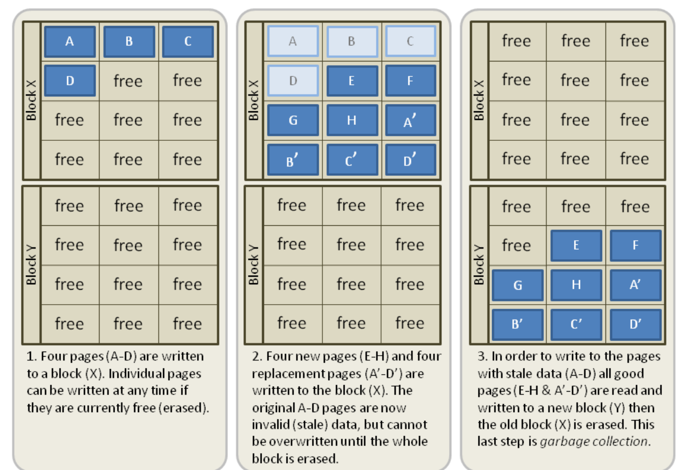


Fig. 1. Garbage Collection in SSD

We can discuss the Fig. 1 to exemplify Garbage Collection in SSD devices. Let us assume, we write a file consisting of four pages, A, B, C and D in Block X. We write another file of four more blocks, E, F, G and H. Then we modify the first file and the modified data are written into A′, B′, C′ and D′. We need to write these on different set of pages as SSD technology does not support direct overwriting of pages. Flash Translation Layer (FTL) in SSD declares the older A, B, C and D pages as invalid later. When there is a command to write more data into Block X, the block needs to be cleaned up first, but the valid data already present in Block X needs to be preserved. Hence, FTL moves those data to another block, Block Y in this case. During this data movement, the invalid data in A, B, C and D pages are removed. This whole functionality is called garbage collection. Hence, in SSDs the last write should suffer I/O degradatation. This is something that SSD devices cannot avoid from happening. So, there should be good software techniques to incorporate efficient GC pipeline to lessen the engendered loss.

B. Kernel Level

In this section, we discuss about different UNIX kernel level file systems and there internal mechanism.

a) UNIX File Systems: File system is one of the most important parts of Operating System that deals with tracking the persistent data kept in the underlying storage devices in a human perceivable format. In general, it has two types of units, a directory and a file. It maintains a tree of files and directories starting from a root directory. A file system is oftentimes designed to leverage the technology of the underlying persistent storage devices and the volatile memory devices. Additionally, it serves

as the software solution for allocating, storing and managing the data. They depend on the operating system and are usually typical for certain storage devices. We explore some of the relevant file systems that are aligned with the context of our work, how they work and their notable advantages and disadvantages. In this subsection we give a primal about some important UNIX file systems.

b) Microscopic Detail of UNIX File System: There are various types of filesystems that have different features and different ways of organizing their data. Some file systems are faster than others, some have additional security features, and some support drives with large storage capacities while others only work on drives with a smaller amount of storage. Some file systems are more robust and resistant to file corruption, while others trade that robustness for additional speed. It is hard to find a filesystem that would work best on all the cases. Each operating system tends to use its own file system, for example Microsoft, Apple, and the Linux kernel developers all work on their own file systems. New file systems could be faster, more stable, scale better to larger storage devices, and have more features than old ones. A file system is not like a partition, which is simply a chunk of storage space, rather it specifies how files are laid out, organized, indexed, and how metadata is associated with them. Most UNIX filesystems maintain a general structure, while the exact details vary significantly. The main components are superblock, inode, directory block, data block and indirection block. The superblock is responsible in maintaining information about the overall filesystem, such the size. An inode contains all information about a file, except its name. The name is stored in the directory along with the number of the inode. A directory entry consists of a filename and the number of the inode which represents the file. The inode contains the numbers of several data blocks, which are used to store the data in the file. There is space only for a few data block numbers in the inode, however, more space for pointers to the data blocks is allocated dynamically. These dynamically allocated blocks are indirect blocks. In this case, to find the data block, one has to find its number in the indirect block first. The block size specifies size that the filesystem will use to read and write data. Larger block sizes will help improve disk I/O performance when using large files, such as databases. Block size limits the maximum supported file size on some filesystems. This is because many modern filesystems are limited not by block size or file size, but by the number of blocks. Before one can use a filesystem, it has to be mounted. The operating system then does various bookkeeping things to make sure that everything works. Since all files in UNIX are in a single directory tree, the mount operation will make it look like the contents of the new filesystem are the contents of an existing subdirectory in some already mounted filesystem.

c) Different Types of File Systems: In this section, we describe different types commonly known file systems.

Extended File System: The extended file system (EXT2-4) was the first file system created specifically for the Linux kernel. Ext2 is an older file system that lacks one of the needed features like journaling. As a result, while writing to an ext2 drive, if the power goes out or a computer crashes, data may get lost. To solve this problem, Ext3 adds these robustness features at the cost of some speed. After that, Ext4 evolved eventually and it is the more modern and faster. It is the default file system on most Linux distributions currently. Ext4 is functionally very similar to ext3, but brings large filesystem support, improved resistance to fragmentation, higher performance, and improved timestamps. In EXT4, instead of fixed blocks, a new concept termed as extents are used for data allocation. An extent is connoted by its starting and ending place on the hard drive. Because of that, it is possible to explain very long, physically contiguous files in a single inode pointer entry. This significantly reduces the number of pointers required to describe the location of all the data in larger files. Furthermore, some other allocation strategies have been implemented in EXT4 to further reduce fragmentation. For example, it reduces fragmentation by scattering newly created files across the disk so that they are not bunched up in one location at the beginning of the disk, as many early PC filesystems did. EXT2 and EXT3, can be mounted as EXT4 to make some minor performance gains. Currently a FS named as better filesystem (Btrfs) is a newer Linux file system that seems very promising. It isnt the default on most Linux distributions at this point, but it is expected to replace Ext4 one day. The goal is to provide additional features that allow Linux to scale to larger amounts of storage.

XFS: XFS is another notable UNIX file system. XFS was developed by Silicon Graphics in order to be used for their IRIX servers. The fundamental point of this file system is its ability to work with files of any size, especially large size. It was specially designed to maintain high performance with large files and filesystems. This FS offers a high level of file optimization, however it

is based on a complex file system structure. The XFS file system uses inodes to store the files metadata and journaling to keep track of system modifications. Only metadata is journaled with this file system. Each inode has a header and a bitmap. XFS stores inodes in a special tree in a specific place on the disk. The system also has a bitmap for free storage blocks.

Journaling Flash File System: Journaling Flash File System (JFFS) [6] and its immediate successor, Journaling Flash File System version 2 (JFFS2) [7] are log-structured file system that is suitable to use with flash memory devices. They are completely alert about the restrictions imposed by flash memory technology. They both operate directly on the flash chips that avoids the inefficiency of involving two Journaling file systems on top of each other. In JFFS, nodes containing data and metadata are stored on the flash chips sequentially. Moreover, there is only one type of node in the log. Each such of those nodes is aligned with a single inode. Compared to JFFS, JFFS2 has supports for the NAND flash devices. This is one of the major differences in their implementation as NAND devices maintains a sequential I/O interface that cannot be memory-mapped for reading. In addition, JFFS2 offers hard links which is not possible in JFFS as there lies limitations in the on-disk format. As a whole, JFFS2 shows better performance compared to JFFS. This mainly because JFFS treats the disk as a purely circular log which generates a great deal of unnecessary I/O. In JFFS2, an efficient garbage collection actually implemented, and the garbage collection algorithm makes this obsolete. Being run in the background, the garbage collector changes dirty blocks into free blocks. This is done by copying valid nodes to a new block, while skipping obsolete ones. After that, it erases the dirty block, then tags it as a free block. This is done so as to prevent confusion if power is lost during an erase operation. However, because of the log-structured design, JFFS2 has some disadvantages too. During the mount time, it is still needed for all nodes to be scanned. This process is very slow, posing a serious problem as flash devices scale upward into the gigabyte range. In addition, writing many small blocks of data leads to negative compression rates, that necessitates the applications to use large write buffers. Moreover, there is no systematic way to tell how much free space is left as usable on a device as this depends both on how well additional data can be compressed and writing sequence.

Flash Friendly File System: Flash Friendly File System (F2FS) is a file system that exploits NAND flash memory-based storage devices, particularly based on Log-structured File System (LFS) [8]. It was designed focusing on eradicating the fundamental issues in LFS, including snowball effect of wandering tree and high cleaning overhead. As a NAND flash memory-based storage device shows different characteristic depending on its internal architecture, F2FS offers several features for both configuration of on-disk layout, and for selection, allocation and cleaning algorithms.

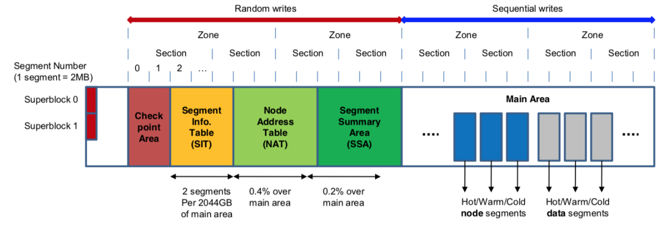


Fig. 2. F2FS layout

In order to support the flash awareness, F2FS provides the feature to enlarge the random write area for better performance as well as it provides the high spatial locality. In addition, it aligns FS data structures to the operational units in FTL. To reduce the wandering tree problem, it uses a notion termed as node which represents inodes as well as various pointer blocks. It introduces Node Address Table (NAT) that contains the locations of all the node blocks. To address the cleaning overhead problem, F2FS supports a background cleaning process. In addition, it supports greedy and cost-benefit algorithms for victim selection policies. Furthermore, it supports multi-head logs for static or dynamic hot and cold data separation. Additionally, it introduces adaptive logging for efficient block allocation

C. User Level

In this section, we discuss about UnifyCR which is our target user level file system for applying GC optimization and adding persistence.

a) UnifyCR: UnifyCR is a user-level file-system which can use node-local flash storage devices as burst buffers for distributed and shared files. It is known as a unified file system that facilitates applications to write efficiently like a parallel file system. The main difference is normally UnifyCR is ephemeral and has lifetime equal to the application running on top of it.

According to Fig. 3, UnifyCR has two basic components, the UnifyCR Server and UnifyCR Library.

UnifyCR Server: The server in this ephemeral file system is responsible for dealing directly with the underlying node local SSD devices. In addition to managing the striping of data in different nodes in a HPC cluster, it

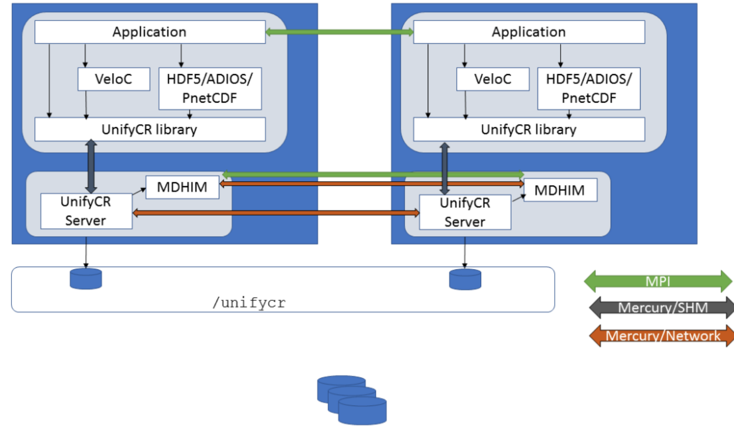


Fig. 3. UnifyCR System Architecture

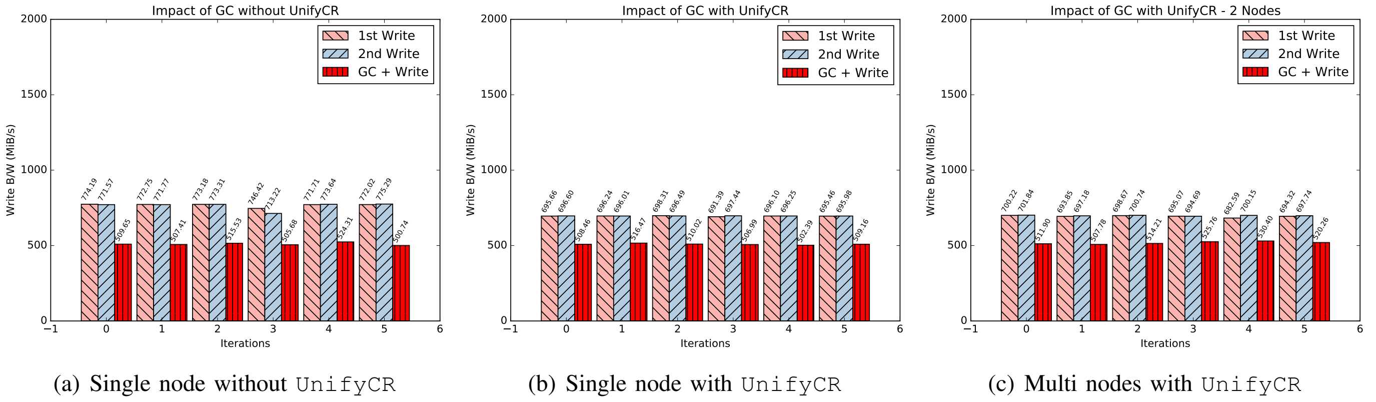
manages the striping of data among the nodes to ensure data parallelism in UnifyCR. The UnifyCR library can communicate with the hardware level through the server. If it is directly started without the intrusion of the library calls, it takes different configuration parameters from the different environment variables assigned for various purposes, like specification of the data mountpoint, metadata mountpoint etc. Moreover, it communicates with MDHIM, the metadata database manager for acquiring information about the metadata.

UnifyCR Library: The library can be considered as the client-side of this system as it exposes set of APIs to the application for using UnifyCR. For example, the application can mount and unmount UnifyCR through using simple `unifycr_mount` and `unifycr_unmount` methods. According to the theory, the data generated by scientific applications are supposed to be flushed into the underlysing persistent PFS, but in the current implementation of UnifyCR it is not present. Apart from that, application can communicate with the library through different data format specific I/O subsystems like HDF5, ADIOS, PnetCDF etc. and multi-level checkpoint restart systems like VeloC. In addition to mount and unmount operations, the library supports different methods to check the status of the file system in runtime by calling them from application.

IV. Motivation

As discussed in Sec. III-A0b, garbage collection is an unavoidable phenomenon in SSD devices. As UnifyCR is made to handle an array of distributed SSDs in HPC clusters, it can suffer I/O overhead because of frequently invoking GC through differnt I/O patterns posed by scientific applications. Hence it is very important to have a dedicated GC handling module in UnifyCR. If we try to perceive this idea in a bottom-up approach, we can first try to emperically deduce the problem caused by garbage collection. For invoking GC, we first write a 3 GB file on a 128 GB partition of an SSD. Then we modify the file again to generate the case of creating some invalid pages in the SSD blocks. Afterwards, when we try to write another file in the same partition, the SSD needs to have some clean blocks in order to write the new file. In this case, there is a definite activation of GC process. We run this experiment 6 times on a mountpoint of ext4 file system. According to Fig. 4(a), we can observe 33% bandwidth degradation due to GC on every third write. Later, we mounted UnifyCR on a ext4 mountpoint to see the impact of GC using the same set of experiments. As shown in Fig. 4(a) for single node and 4(c) for 2 nodes, bandwidth is slightly decreased because of overhead added by UnifyCR processes, but still it degrades the bandwidth to around 30% for the third run. These results clearly demonstrate the impact of GC on SSD devices and we also can deduce that, UnifyCR still do not have a module to handle GC efficiently in device level. This phenomenon motivates us to incorporate an efficient GC mechanism in UnifyCR by leveraging the in-built GC unit in F2FS file system when UnifyCR is mounted on a F2FS mountpoint.

Furthermore, we got motivated by the importance of data persistence in flash memory devices. Persistence can be explained as the continuance of an effect after its cause is removed. In the context of data storage in computer system, this means that the data survives after the process with which it was created has ended. In simple words, for a data store to be considered persistent,



(a) Single node without `UnifyCR` (b) Single node with `UnifyCR` (c) Multi nodes with `UnifyCR`

Fig. 4. Impact of GC on Single Node

it must write to non-volatile storage.

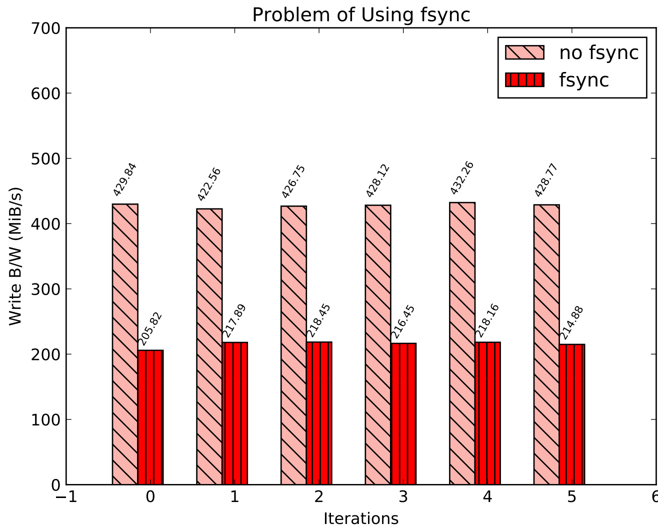


Fig. 5. Impact of Synchronous Persistence Mechanism

Data persistence is also required to make a system fault-tolerant. This actually refers to a systems ability to allow for failures or malfunctions, and this ability may be provided by software, hardware or a combination of both. In a software implementation point of view, OS provides an interface that allows a programmer to checkpoint critical data at predetermined points within a transaction. Hence, periodic checkpointing is necessary for making the application recoverable from a drastic situation. Another importance of data persistence is in facilitating data archiving for an organization. Data archiving is the process of moving data that is no longer actively used to a separate storage device for long-term retention. Archive data consists of older data that remains important to the organization or must be retained for future reference or regulatory compliance reasons. Data archives are indexed and have search capabilities, so files can be located and retrieved.

In order to equip UnifyCR with data persistence, we need to add a flushing mechanism at least on `unifycr_unmount` operation. For this, we keep the data generated by an application in a buffer and flushed the buffer to a persistent storage through executing `fsync` system call with that buffer. To check the opportunity cost for adding data persistence, we perform some baseline experiments using SysIO-ReadWrite application from the UnifyCR package by running it 6 times with synchronous flushing on `unifycr_unmount`. We observe almost 50% write bandwidth degradation in the I/O generated by the application after adding flushing from original bandwidth. Hence, we can clearly see that even though data persistence is important, it definitely comes with some opportunity cost. The second motivation of our design is to lessen this cost by adding techniques like asynchronous flushing of data.

V. Design and Implementation

In this section, we discuss the high level System Architecture of our flash-friendly and persistent implementation of `UnifyCR`. Afterwards, we depict the implementation techniques of two basic features in the system. We first discuss about the incorporation of flash-friendliness by means of using F2FS followed by that of asynchronous persistence in the system to lessen the time consumption due to adding persistence.

A. System Architecture

The architecture of the flash-friendly and persistent system consists of a persistent storage stack made of a Persistent Parallel File System (PPFS), e.g. Lustre, BeeGFS etc., with an ephemeral UnifyCR leveraging the F2FS mountpoints of SSD devices on top. There is a data pipeline to asynchronously move temporary data on

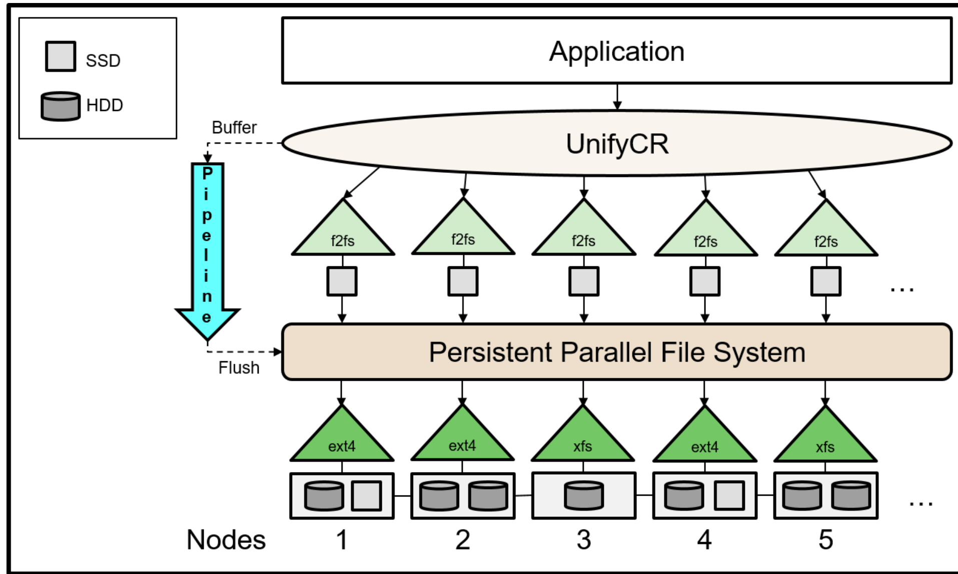


Fig. 6. A Sketch of the System Architecture

UnifyCR to the PPFS when UnifyCR is unmounted from the application. The PPFS manages the underlying array of distributed persistent storage devices, i.e. HDD and SSD, that can have any file system mounted on top that are treated as the management, metadata and object storage targets for the management, metadata and object storage services.

According to Fig. 6, a distributed HPC cluster can have numerous underlying nodes denoted by numbers from 1 to 5, which can be equipped with multiple HDD and SSD devices for persistent storage. These devices can be formatted by ext4, xfs, zfs or any type of file system and accessible to the PPFS through file system mountpoints. These mountpoints are exposed to the PPFS by being treated as the management, metadata and object storage targets. On the other hand, management, metadata and object storage services are configured in such a way that, those are configured to use these persistent targets. This PPFS will be mounted to a single point in the distributed system which is accessible from any node of the system. Modern HPC clusters usually have an array of on-node SSD, NVMe or any other fast flash devices that can also be referred to as a burst buffer. According to our design, these flash devices are formatted as F2FS and mounted on a location on each node. The UnifyCR setup to manage this array of flash devices is configured to leverage these mountpoints so that it can take advantage of the flash-friendly garbage collection mechanism as discussed in Section III-B. This UnifyCR system is mounted on a directory with the same URL in all the nodes. From the application using UnifyCR, this per node UnifyCR setup is mounted and utilized for the I/O accesses. This burst buffer file system can be leveraged to handle random bursty I/O patterns invoked by scientific applications running on HPC systems. Finally, we maintain a data pipeline to add persistence to the data that are generated by the applications. This pipeline asynchronously perform system calls to drain the data from UnifyCR mountpoint to the PPFS mountpoint.

B. Implementation

The implementation has to steps, i.e. incorporating F2FS in UnifyCR and implementing persistence. We discuss the techniques to implement the ideas in the following sections.

1) Incorporating F2FS in UnifyCR: The implementation of the flash-friendliness is achieved by incorporating F2FS in UnifyCR and using it from the application layer. Introducing effecient metadata management by maintaining two layers, one in F2FS another in UnfyCR, can be considered as one of the challenges posed by this implementation. We perform a deep dive into the codebase of UnifyCR to find a suitable place in the code to segregate at least the metadata and actual data in the UnifyCR system. We find that, the existing system internally manages the data and metadata in a modular manner. Hence, it is easy to segregate data and metadata management through different file systems through adding two configuration entries in the system for configuring the path to manage the data and metadata. These entries are managed by two environment variables, i.e. UNIFYCR_EXTERNAL_META_DIR and

UNIFYCR_EXTERNAL_DATA_DIR that are read by the UnifyCR implementation.

For enabling F2FS feature in the system, we build f2fs-tools from source. Unfortunately, the kernel we had in the nodes of the cluster did not support F2FS mounting. Hence, we perform a kernel update to Linux-4.19.6 in the cluster we use for the experiments. Afterwards, we format the devices with mkfs.f2fs and mount it on a directory on each node. Besides, we also keep another partition in the flash device formatted in ext4 and mounted on another location. Finally we specify the metadata and data mountpoints in UnifyCR through the environment variables mentioned and try different combinations of metadata and data mountpoints as discussed in Section VI-B2.

2) Implementing Persistence: During the first phase of the implementation of persistence in the ephemeral UnifyCR burst-buffer file system, we utilize the system call fsync defined in the unistd.h file. We simply maintain a char buffer through the application lifetime using pwrite sysmtem call and specify the location in the system that is mounted on a persistence storage system. In ideal case, it should be the mountpoint of a PPFS, but in our experiments we considered the ext4 mountpoint on a device as the persistent module of the design as we do not have a parallel file system installed in the in-house small cluster named impact. We assume that, if we use real PPFS mountpoint, the bandwidth will be better than the currently reported numbers. Later, we call fsync when the UnifyCR system is unmounted through the application. There is a basic problem with fsync, as it is a synchronous blocking call to the kernel. Hence, once this blocking method is called, it holds the execution until it is finished. So, if the buffer size is large, it will take a while to flush those data to the persistent storage system and hamper the total application execution time.

Listing 1. aiocb structure

```
struct aiocb
{
    int                    aio_fildes;
    off_t                  aio_offset;
    volatile void         *aio_buf;
    size_t                 aio_nbytes;
    int                    aio_reqprio;
    struct sigevent        aio_sigevent;
    int                    aio_lio_opcode;
};
```

For eradicating this problem with synchronous flushing, we add the feature of flushing the data asynchronously. By this way, the application do not need to wait for the flushing to be finished. The buffer is sent to the kernel space and kernel finally deals with the complete transfer of data from UnifyCR mountpoint to the underlying persistent system. For this reason, we use aio_fsync defined in aio.h file. As mentioned in Listing 1, aiocb has a aio_buf. We use this buffer to keep the data generated by the application using aio_write and update the aio_filedes with a file descriptor pointing to a file to be created in the underlying persistent storage system. Then during unmount, we call aio_fsync to invoke an asynchronous system call to drain the buffer data to the file pointed by the file descriptor. We finally check the checksum of the files created by fsync and aio_fsync to check the sanity of the file. We found that, both the versions of file are same for the same application.

VI. Experimental Evaluation

In this section, we describe our experimental setup and the evaluation results. We measure the I/O bandwidth of UnifyCR by mounting data and metadata modules on different file system setups. We also evaluate the impact of asynchronous flushing mechanism on UnifyCR and compare it with the contemporary approach.

A. Experimental Setup

We conduct all the experiments on our in-house *Impact cluster* located at the Computer Architecture and Systems Research Laboratory, Florida State University. The cluster is equipped with Intel(R) Core(TM) i5-4690 CPUs, 4 Cores per node. Each CPU has a base frequency of 3.5 GHz with a maximum clock speed of 3.9 GHz. Each of the nodes also comes with a 1 TB of hard disk drive and 256 GB of solid state drive. Innovation uses Intel Corporation Ethernet Connection I217-LM as interconnect which supports 1GbE.

For our experiments, we use 2 of its nodes configured with both HDD and SSD. The Linux kernel version of these two nodes have been updated to *Linux 4.19.6* for enabling the F2FS support. F2fS tools have been used for mounting the F2FS file system. We have also configured UnifyCR and varies the underlying file systems for comparison purposes. IOR benchmark is used to determine the write bandwidth of UnifyCR operations on SSD. We use mpiio provided by IOR and SysIO-ReadWrite application provided by UnifyCR for our evaluation. The garbage collection is invoked manually for measuring the garbage collection overhead and its

impact on UnifyCR performance. One file is written twice on the same UnifyCR mount point to make the SSD block occupied. Then a second file is written on the same mount point which enforces the SSD to invoke garbage collection for removing the stale data of the first file. We have empirically decided the file size as 3GB to satisfy our need for invoking garbage collection using the configuration as mentioned earlier. In the next part of our evaluation, we calculate the impact of data persistence in UnifyCR using synchronous and asynchronous flushing mechanism. We again collected the write bandwidth using IOR benchmark to determine the impact of different persistent mechanism for UnifyCR.

B. *Impact of Garbage Collection on UnifyCR*

Impact of garbage collection overhead has been evaluated by collecting the write bandwidth results for different scenarios. First, the default F2FS write bandwidth after invoking the garbage collection is compared after mounting the UnifyCR over F2FS on a single node. Then the impact of garbage collection is also evaluated using multiple nodes while setting different configurations for UnifyCR data and metadata mount points.

1) Impact of F2FS on GC - single node: Fig. 7 shows the overall impact of UnifyCR over default write bandwidth of F2FS. It also demonstrates the impact of garbage collection overhead before and after mounting the UnifyCR.

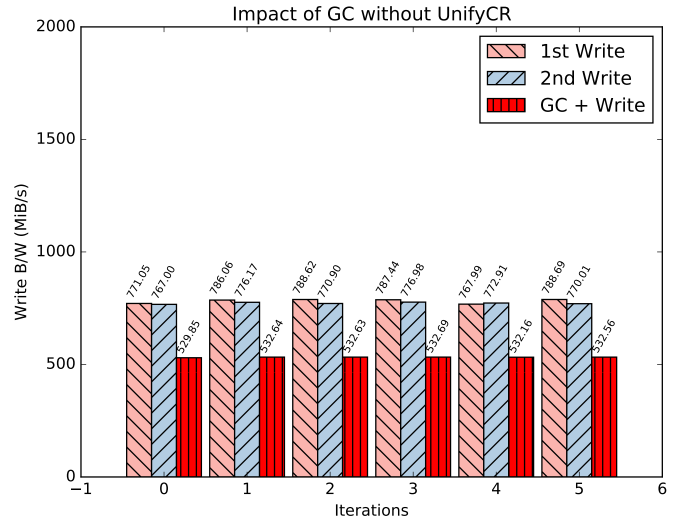
Fig. 7(a) shows a significant performance improvement in write bandwidth after invoking garbage collection. The write bandwidth improvement is around 4% over ext4. However, we can see an overall degradation in write bandwidth performance after mounting UnifyCR which is expected.

Fig. 7(b) shows the write bandwidth results after mounting UnifyCR over F2FS. The write bandwidth reported after invoking garbage collection is around 528 MiB/sec which was 507 MiB/sec in case of ext4.

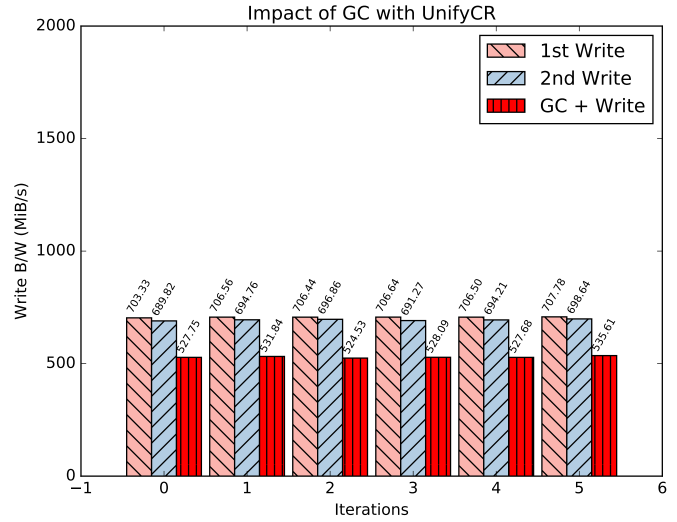
2) Impact of F2FS on GC - Multi node: The F2FS file system is mounted on two of the nodes for evaluating the performance of UnifyCR after the invocation of SSD garbage collection unit. From Fig 8, it is evitable that the F2FS helps to improve the performance of UnifyCR in case of garbage collection. The write bandwidth is around 528 MiB/sec in case of F2FS which was previously reported as 520 MiB/Sec in case of ext4.

3) Segregation of Data and Metadata on Different Filesystems: The impact of garbage collection overhead on UnifyCR after using F2FS as the underlying file system is analyzed further by setting different data and metadata mount points. The evaluation has been done in two phases. In the first phase, the F2FS is set as the metadata mount point and traditional ext4 as the data mount point. In this scenario, we get an improvement for garbage collection overhead over storing both the data and metadata in ext4. However, the improvement is lesser than storing all the files in F2FS. This result is obvious as the size of metadata stored in F2FS is considerably small than the original data which is stored in ext4. Fig. 9(a) shows that the write bandwidth after invoking garbage collection is around 515 Mib/sec which is better than storing all the files in ext4 but lesser than storing all the files in F2FS.

In the next phase, we store the original data in F2FS



(a) without `UnifyCR`



(b) with `UnifyCR`

Fig. 7. Impact of GC on Single Node

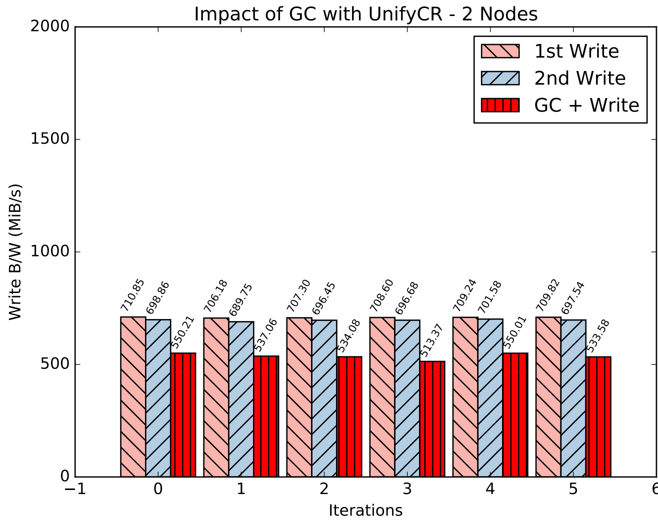
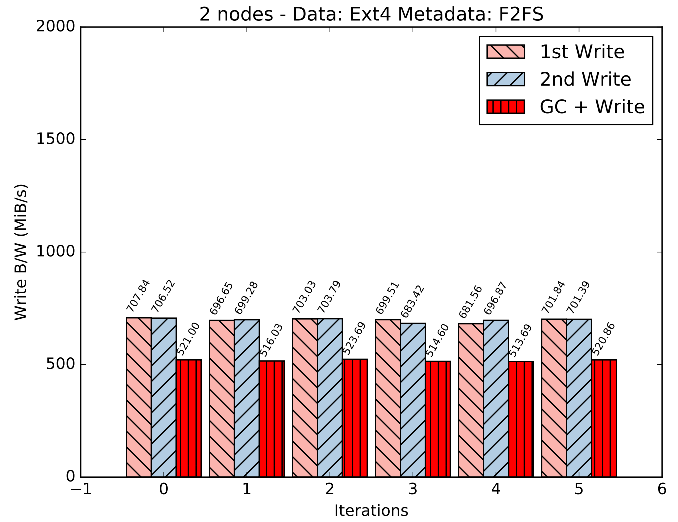
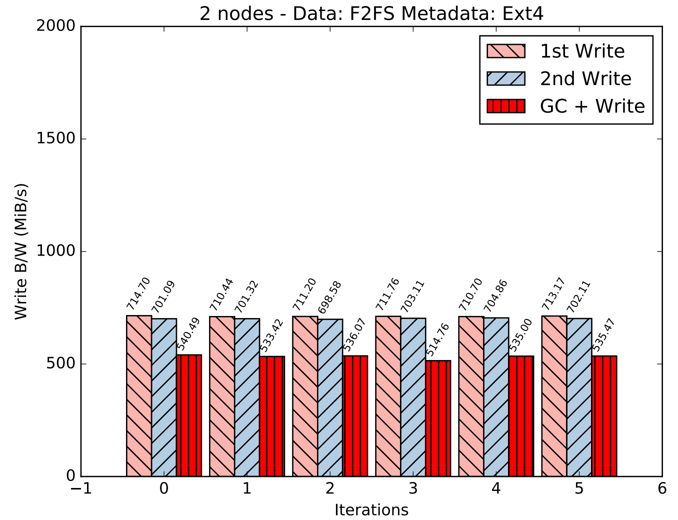


Fig. 8. Impact of GC on Multi Node Node



(a) without UnifyCR



(b) withUnifyCR

Fig. 9. Impact of GC with Different Data and Metadata Mountpoint

and metadata in ext4. Fig. 9(b) shows that a significant bandwidth improvement after garbage collection. The primary causes can be the better frequent read performance of ext4 in case of metadata fetching and efficient handling of the big amount of data files into SSD by F2FS. The bandwidth after invoking garbage collection in this scenario is around 535 MiB/sec which around 3% better than the results demonstrated in Fig. 9(a).

C. Impact of Asynchronous Persistence Mechanism

Addition of asynchronous persistence mechanism during UnifyCR unmounting results in a signification bandwidth improvement over the synchronous persistence mechanism. Fig. 10 compares the write bandwidth of UnifyCR without any persistence mechanism, with synchronous and asynchronous persistence. The evaluation results show that the write bandwidth of UnifyCR without any persistence mechanism is around 430 Mib/sec. However, the addition of synchronous persistence mechanism reduces the write bandwidth by a significant amount which is around half of the default bandwidth of UnifyCR. The main reason behind this behavior is synchronous flushing of data while unmounting UnifyCR blocks the completion of UnifyCR operation while increasing its total execution time. However, the asynchronous *aio_fsync()* call execute the flushing mechanism without blocking the real execution. The experiments show a significant performance improvement over synchronous flushing mechanism with a bandwidth improvement of 50%. Overall, the asynchronous flushing mechanism can manage to achieve near default bandwidth of UnifyCR.

VII. Related Works

F2FS [5], also known as Flash-Friendly File System is a log structure file system, specifically designed for modern flash storage devices. The main design ideas behind F2FS are flash friendly on-disk layout, cost-effective index structure, multi-head logging, adaptive logging and acceleration with roll-forward recovery. F2FS on-disk layout consists of three configurable units known as segment, section, and zone. It allocates storage blocks in the unit of segments from a number of individual zones and performs cleaning in the unit of sections. The flash -friendly layout of F2FS helps to align the filesystem garbage collection with FTL garbage collection unit. The cost-effective index structure of F2FS helps to restrain

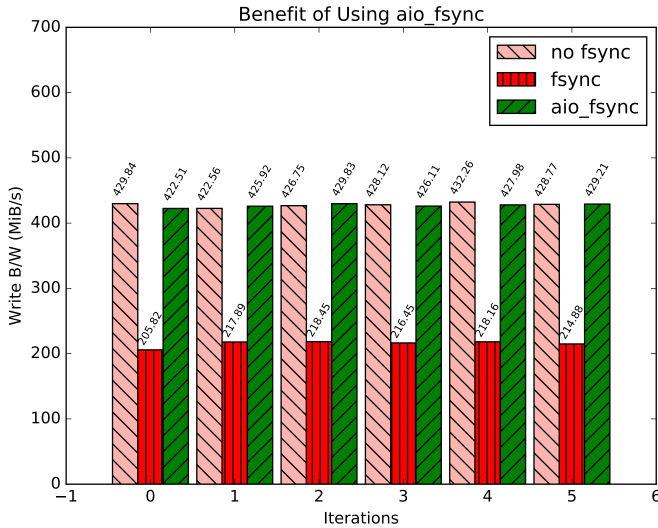


Fig. 10. Impact of Asynchronous Persistence Mechanism

write propagation overhead by using Node Allocation Table. Moreover, it also reduces the cleaning cost by using multi-head logging. The F2FS distribution is included in Linux mainline kernel since Linux 3.8.

In recent High Performance Computing (HPC) systems, hard disk drives (HDDs) are still used as prominent storage devices which renders the system under a huge I/O overhead while handling random data access in data intensive scientific applications. With a view to resolving the I/O problems, the characteristics of SSDs that involve no seek latency and high throughput are leveraged by incorporating an intermediate layer of distributed SSDs in the form of Burst Buffer to absorb bursty I/O requests and improve the write performance. In these sorts of hybrid storage systems, two of the most striking challenges are, the requirement of large and expensive SSD capacity and harmonious overlapping between computation and data flashing stage. In this research paper, a traffic aware SSD burst buffer system, SSDUP [9] (an SSD write buffer Using Pipeline) has been proposed and experimentally established to be improving the I/O performance by more than 50regarding capacity of the SSDs, this method is designed to buffer only the random writes to SSD buffer, and other writes are considered as sequential and directly flushed to the HDDs. Besides, a pipeline mechanism has been proposed to resolve the problem of perfectly overlapping the computation phase and flushing stage. In addition, the sequence information of the random data in the SSDs are stored in an AVL tree structure and the data are stored in a log-structure to sort the data sequentially before finally flushing to the HDDs and make the flushing of data faster. SSDUP is developed on top of OrangeFS and finally extensive experimental evaluations has been performed to gather a comparative performance information and the system is established as an improvement of the Burst Buffer system.

VIII. Future Work

The asynchronous persistence mechanism is introduced in our implementation to ensure the data generated during the operation should be flushed out to the HDD during unmount. However, the write pipelining mechanism can be improvised in many ways to make it more cost effective. One of the early work mentioned in [9] by Xuanhua et. al. shows how an efficient write pipelining mechanism can improve the overall performance of burst buffer file systems by flushing the random writes sequentially into SSD with the help of AVL tree type structure. Design to detect the I/O pattern of burst buffer file system using different application can be enabled to identify the requirements of flushing during execution by overlapping it with the computation phase. The flushing can also be handled using a background thread which will be responsible to write the sequential data directly to HDD and write the random data into a buffer before flushing them sequentially in HDD. Also, our implementation only demonstrates the improvement achieved by using F2FS as the underlying file system for UnifyCR. Whereas, the performance can be measured by using several other flash friendly file systems including btrfs [10] to gain a better exposure about the impact of different flash friendly file system over the garbage collection mechanism of SSD.

IX. Conclusion

We have proposed a design for incorporating F2FS in UnifyCR to take advantage of its efficient garbage collection unit which is specifically designed for solid state drives. Also, we have proposed an efficient asynchronous flushing mechanism for UnifyCR to enable fail-safety and checkpointing of the application status. Our implementation orchestrates a number of design ideas to address several critical challenges of UnifyCR efficiently. Moreover, We have evaluated our design extensively using different configurations and also compared it with the traditional ext4 file system. Our evaluation results show a visible performance improvement for UnifyCR by using the flash-friendly file system, even in case of garbage collection. We have also enabled the provision of both synchronous and asynchronous flushing

for data checkpointing. The measurements show a significant bandwidth gain for asynchronous mechanism over the synchronous one by achieving almost near default bandwidth of UnifyCR. We plan to investigate the performance by using other flash-friendly file systems as UnifyCR mount point. Furthermore, We also intend to improve our existing flushing mechanism by including a cost-effective write pipelining system.

REFERENCES

[1] "Github link to UnifyCR code." [Online]. Available: https://github.com/LLNL/UnifyCR

[2] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 69:1–69:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014997

[3] "Computer architecture and systems research lab (castl)." [Online]. Available: http://castl.cs.fsu.edu/doku.php

[4] T. Wang, A. Moody, Y. Zhu, K. Mohror, K. Sato, T. Islam, and W. Yu, "Metakv: A key-value store for metadata management of distributed burst buffers," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1174–1183.

[5] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 273–286. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee

[6] D. Woodhouse, "Jffs: The journalling flash file system," 12 2018.

[7] "Jffs2." [Online]. Available: https://en.wikipedia.org/wiki/JFFS2

[8] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992. [Online]. Available: http://doi.acm.org/10.1145/146941.146943

[9] X. Shi, M. Li, W. Liu, H. Jin, C. Yu, and Y. P. Chen, "Ssdup: a traffic-aware ssd burst buffer for hpc systems," in *ICS*, 2017.

[10] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *TOS*, vol. 9, pp. 9:1–9:32, 2013.