Hitchhike: an I/O Scheduler Enabling Writeback for Small Synchronous Writes

Xing Liu[†‡], Song Jiang[†§], Yang Wang[†], and Chengzhong Xu[†§]
[†]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
[‡]Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences
[§]Department of Electrical and Computer Engineering, Wayne State University, USA
{xing.liu, yang.wang1, cz.xu}@siat.ac.cn

***Abstract*—Small and synchronous writes are pervasive in various environments and manifest in various levels of software stack, ranging from device drivers to application software. Given block interface, these writes can cause serious write amplifications, excess disk seeks or flash wear, and expensive flush operations, which, together, can substantially degrade the overall I/O performance. To address these issues, we present a novel block I/O scheduler, named *Hitchhike*, in this paper. *Hitchhike* is able to identify small writes, and embed them into other data blocks through data compression. With *Hitchhike*, we can complete a small write and another write in one atomic block operation, removing write amplification, and the overhead in excess disk seeks. We implemented *Hitchhike* based on the *Deadline* I/O schedulers in Linux 2.6.32, and evaluated it by running *Filebench* benchmark. Our results show that compared to traditional approaches, *Hitchhike* can significantly improve the performance of synchronous small writes.**

I. Introduction

Accesses to small data, or data that is much smaller than a block, are pervasive and often performance ciritical in modern computing environments, especially with the incoming era of big data-based applications where increasingly more and frequent on-disk meta-data is involved. For example, according to Miller et al. [1], the meta-data of file systems as a whole only occupies 0.1% to 1% of total storage capacity, but accounts for about 50% to 80% of the total access volume to the file systems [2]. On the other hand, for application-level scenarios, like those in the key-value stores[3] and object-based storage systems, small writes are also widespread in terms of access frequency and volume as evidenced by Facebook whose key-value items smaller than 500 bytes is reported to occupy approximately 90% capacity of its in-memory cache [4].

Additionally, the generated small-data writes in certain cases are also synchronous, which means the data written must be persisted immediately on the disks. This includes updates of file system meta-data, recording entries of lookup table for virtual disk devices, and so on. For example, when a block needs to be written to the disk, some associated bitmap information is also required to be recorded at the same time for data integrity. In this scenario, the modification to the *bitmap* as small data must follow the *write-through* policy to immediately persist on disks.

Currently for efficient access mainstream storage devices, including hard disks and solid-state disks, enforce an access interface with respect to block or page, that is much larger than access unit of byte-addressale memory As such, the mismatch between the block-based devices and the small data accesses could cause serious data write amplifications[5][6], excess disk seeks or flash wear, and expensive flush operations, drastically degrading the overall I/O performance.

To address this issue, current technologies typically resort to special hardware supports, such as NVRAM and PCM memory [7], [8], to enable writeback buffers that could accumulate multiple small writes in the buffer before committing them to the disks together in one go. However, these technologies either suffer from the physical limitations of the new media or are not effective in addressing the issues caused by the synchronous small writes, such as write amplification and write order enforcement. Unlike previous studies, in this paper we propose *Hitchhike*, a novel I/O scheduler that can identify and remove small writes among requests received from the file system. The essence of *Hitchhike* is introducing a new data embedding and hitchhiking technique to remove the overhead for accommodating small writes. Specifically, when a small write request is generated, a data block can be identified and its data is compressed to make room to accommodate the small data in the block. As such, we can complete a small write and other write in one atomic block operation, removing the associated potential write amplification, and the overhead in disk seek, and flush operations as well. An illustration of the technology as compared with traditional methods for small data persistence is shown in Figure 1. As shown, in the conventional layout, metadata is stored only at the original metadata block, while in *Hitchhike*, data in a block is compressed to make space for temporarily holding metadata so that two writes (one for data block and the other for metadata block) can be merged into one, dramatically improving the small write performance.

By opportunistically employing the data compression technique to piggyback semantically or temporally related small data on a data block, *Hitchhike* represents a major deviation from conventional use of the compression technique for reducing space and I/O volume. It is a disruptive technology to addressing the issue of deteriorating situation on the conflict between increasing use of synchronous small writes and persistent use of block storage device in today's data-intensive computing platforms. Our prototype-based evaluation demonstrates that *Hitchhike* can significantly improve the

978-1-5090-3555-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCBD.2016.55

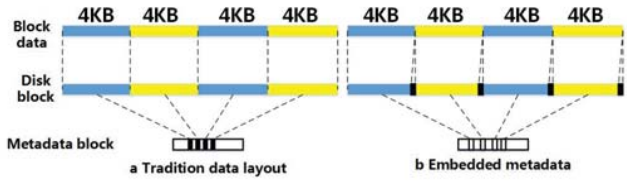

Fig. 1. Illustration of methods for persisting small data.

performance of synchronous small writes.

The reminder of this paper is organized as follows. We introduce the design of *Hitchhike* in Section II, and present its performance evaluation in Section III. After that, we survey and compare with some related works in Section IV, and finally conclude the paper in the last section.

II. Design of Hitchhike

As shown in Fig. 2, when the file system writes some data to a block device, it actually creates a stream of block requests, which are placed into a dispatch queue and scheduled by an I/O scheduler [9]. Among these requests, some could be small updates on existing data blocks, such as updates on file system's meta-data blocks and new entries appended at a log file. Such small writes, as we have shown, can drastically degrade the I/O performance. Our goal in this design is to leverage a data embedding and hitchhiking technique to have a new I/O scheduler, *Hitchhike*, that can remove the small writes as separate requests.

To reach our goal, we require a new writeback mechanism for *Hitchhike*, allowing it not only to retain the regular scheduling functionality, but also to have five new capabilities:

1) it can identify small writes hidden in the write requests by maintaining a write buffer and performing comparison of block contents;
2) it can search in the scheduler's dispatch queue for writes whose data can be compressed to host the small data;
3) it can adaptively determine whether a small write should be embedded and where it is embedded to maximize I/O performance;
4) it can quickly recover lost data after a failure by searching the embedded data; and
5) it does not need to access the embedded data during system's normal operations.

The design of *Hitchhike* centers around these capabilities.

a) Write-back buffer: As apposed to other I/O schedulers, *Hitchhike* maintains a *writeback buffer*. Any write request submitted from generic block layer to I/O scheduler will be first inserted or merged into the scheduler's dispatch queue, as existing schedulers do. Writeback buffer is responsible for caching *bio*, a core data structure of the write request and describes the I/O block device operation.

To find a *bio* in *writeback buffer* efficiently, the writeback buffer is organized as a *lookup tree*, which is a *red-black tree*[10] with *bio*s being linked according to their sector orders. The *lookup tree* is used to sort and dispatch the requests in the *Deadline* I/O scheduler[11]. Additionally, to maintain the *writeback buffer*, an LRU-based replacement algorithm is used to decide which *bio* should be evicted when the buffer is full. *Hitchhike* keeps track of the timing when a *bio* is used as its timestamp, and separates all *bio*s cached in the *writeback buffer* into two lists called *active list* and *inactive list*, where the former links those *bio*s that have been accessed written, while the latter tends to include the *bio*s that have not received writes recently. The *bio* in *inactive list* will be firstly evicted to make room for new *bio* when the writeback buffer is full.

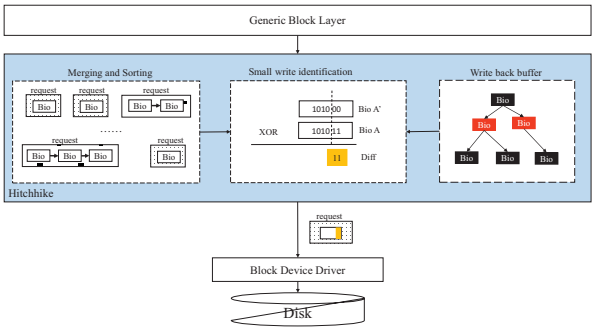

Fig. 2. Hitchhike I/O architecture in Linux.

b) Small write identification: When a write request containing one or more *bio*es is submitted to the I/O scheduler, *Hitchhike* will iterate the *bio*s. At each iteration, it will search the *lookup tree* according to the disk address of one *bio*. Let us denote the *bio* as `A`. If a *bio* with the same disk address (*bio* `A'`) as `A` has existed in the writeback buffer, a *diff* between the data of `A` and `A'` is computed using *Exclusive OR* (XOR). Then, the *diff* will be compressed by *Extremely Fast Compression* algorithm (LZ4), which is a fast lossless data compression algorithm. Besides, the compressed *diff* will be marked with a timestamp and the disk address of *bio* `A`. This timestamp is useful if a block has multiple *diffs*.

A correct order is required during reconstruction of the block should it be lost after a system failure. After calculated the *diff*, *bio* `A` will be written into the *writeback buffer* and replace *bio* `A'` for computing future *diffs*. The compressed *diff* is used to determine whether *bio* `A` can be considered as a small write. For example, when a program continuously appends small new entries at the end of a log file in a synchronous fashion, many small *diffs* (or small writes) can be detected on the last *bio* of the file. When there is a write request ready to be dispatched to the device, *Hitchhike* will determine whether there is a *diff*. If yes, the *bio* of the write request will also be compressed to make the room. If the room is sufficiently large, a compressed *diff* will be embedded to the room.

c) Data persistence: For write requests, *Hitchhike* uses two modes to persist data on the disk. One is the traditional full-block mode in which the whole block of data is written to the block's on-disk address. The other is the partial-block mode in which only *diff* data is persisted in host blocks.