creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. DISSERTATION

Efficient Data Management Schemes for High-performance Storage Devices

고성능 저장장치를 위한 효율적인 데이터 관리 기법

AUGUST 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Nae Young Song

Ph.D. DISSERTATION

Efficient Data Management Schemes for High-performance Storage Devices

고성능 저장장치를 위한 효율적인 데이터 관리 기법

AUGUST 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Nae Young Song

Abstract

Recently, storage devices have developed with the advancement of hardware technology, and high-performance storage devices such as Solid-State Drives (SSDs) have appeared. High-performance storage devices offer high bandwidth, low latency and high I/O parallelism. They eliminate the mechanical overhead of traditional Hard Disk Drives (HDDs), resulting in data access from tens to hundreds of times faster. However, if these high-performance storage devices are used in the existing software layers, the performance cannot be maximized due to the overhead of the software layers.

In this dissertation, we optimize the existing software layer to exploit the high-performance storage devices. To this end, we propose data management schemes in memory management and VFS layer Because high-performance storage devices have very low-latency to access data, the overheads in existing software layers becomes more visible.

The first software overhead pointed out in this dissertation is page reclaim overhead when data are accessed with memory-mapped interface. This interface maps the physical pages into the process's virtual address space. The unmap overhead of the software layer is highlighted when reclaiming the memory-mapped pages based on a high-performance storage device. To reduce the unmap overhead, we propose a page recycling scheme and limit the unmap overhead to the corresponding application. With the scheme, we can increase the whole system performance.

The second is the metadata lookup operation. The metadata lookup operations that must be performed before path-based file access are redundant. The

overhead of these redundant lookup operations becomes more visible when the data access latency becomes low on high-performance storage devices. Therefore, we propose a backward finding mechanism for efficient metadata operation. In this way, we can reduce the number of metadata operations.

The last scheme is for the Log-Structured Merge (LSM) algorithm. The traditional LSM algorithm is constructed assumed that the storage device is enough to slow. So, it has complicated in-memory data structure to reduce the storage access for data management. Therefore, LSM algorithm suffers from side effects such as write amplification. When using the high-performance storage devices, we don't need to use complicated data structures because the storage latency is low. In this dissertation, we remove the software overhead in LSM algorithm by using the simplified data structure, and this data structure leads to reduce the write amplification.

Keywords: High-Performance Storage Devices, Data Management Scheme, Operating System, Memory Management

Student Number: 2010-23270

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern society is changing rapidly. Users often produce, share and require analysis or computation of data to understand and keep up with changing society. These changes have a great impact on modern computing systems.

One of the changes is that the data has grown up to a large scale. This may be due to the development of the Internet, such as Social Network System (SNS) and personal web pages, and the generalization of the cloud system. Users can produce, share, and reproduce data in global. In response to this change, the demanding for fast data processing is increasing. The users require more complex and accurate data analysis and faster processing of data. Therefore, computing systems have changed in many aspects to meet these requirements. To make it easy to analyze the vast amount of data, some researches make the distributed frameworks [1, 2] for large-scale data processing. Google [3] proposed a programming model, Map-Reduce [1] and it has been written in many programming languages. These frameworks enabled a tremendous amount of data processing in parallel and distributed form. Changes to the large-scale

data also affected the memory management system and file systems (FSs) in computing systems, and there are many related researches.Typically, there are systems [4, 5] for handling the amount and number of files that grow, and systems [6, 7, 8] to achieve high-level I/O performance for a large number of files.

Another change is the development of hardware. Techniques of storage devices have evolved rapidly and, finally have greatly reduced hardware latency in data access. This trend is expected to continue, and companies and researchers are developing more faster storage devices such as flash memory, PRAM or MRAM. These high-performance storage devices have difference characteristics from the traditional Hard Disk Drives (HDD) [9]. The high-performance storage devices are usually performed using electronic signal by eliminating the mechanical part of HDDs. Therefore, they have very low data access latency and resistant to physical shock. In addition, the host interface, which enables communicate hardware device to host systems, has evolved greatly so that the communication overhead is reduced when data is transmitted to/from the host systems. These developments have a great impact on storage systems and file systems, and even memory management systems. With the development of Solid-State-Drives (SSDs), specialized file systems [10, 11] have emerged to reflect their characteristics.

High-performance storage devices such as flash-based SSD are already prevalent in data center or modern computing systems because they can be replaced the traditional HDDs without any code changes (drop-in replacement). However, drop-in replacement of high-performance storage devices cannot exploit the full performance of the new devices. This is because that the software layers are designed to use the HDDs. Therefore, we should optimize the software layers for the characteristics of high-performance storage devices. In this dissertation,

we propose efficient data management mechanisms by reducing the software overhead based on high-performance storage devices.

1.1 Approaches and Contributions

In this dissertation, we examine the characteristic of high-performance storage devices, especially flash-based SSDs. With the examination results, we propose three optimizations for data management mechanism on high-performance storage device.

First, we optimize the memory-mapped I/O on high-performance storage devices. Because file I/O using memory-mapped scheme can be more efficient than existing file I/O using read/write systemcalls, data-intensive applications may use the memory-mapped I/O on high-performance storage devices. We profile memory-mapped I/O path and virtual memory subsystem of Linux. We also analyze the impact of '`mmap()`' on high-performance storage devices and find out the problems of performance degradation. To remedy the problem of 'mmap', we propose a new concept of page reclaim, *page recycling*, for an efficient memory-mapped I/O.

Second, we optimize the metadata-related operations on high-performance storage devices. We will look at the VFS layer which are in charge of metadata operations, and then consider the optimizations on high-performance storage devices. On large scale systems, the overhead of metadata lookup is increasing by the number of files that grows. This is because that the files are managed in path-based mechanism, and the system have to traverse the path in regular order one by one. This overhead is more visible on high-performance storage devices when the data access latency becomes very low. To reduce the overhead in metadata-related operations, we propose a new path finding schemes that

performs the hash table lookup operation *in a backward manner* to find the target dentry.

Third, we optimize the Log-Structured Merge (LSM) algorithm which manage the data as key-value pairs for the high-performance storage devices. LSM can deal with write-intensive workloads by cascading data from smaller, high-performance stores such as RAM to larger, less performant stores. Traditional LSM algorithm is devised assume that the storage device is very slow. So, it has very complicated in-memory data structures to make the data access more efficiently. However, with the complicated data structures, it suffers from heavy write amplification during the compaction. To mitigate the inefficiency of the original LSM algorithm, we propose the *Ranged Log-Structured Merge (RLSM) tree* by considering the high-performance storage devices. In RLSM, we conduct the compaction with append operations based on data range in on-storage files. This algorithm takes advantage of high-performance storage devices.

The contributions of this dissertation can be summarized as follows:

- We study and recognize the characteristics of high-performance storage devices. We also analyze the existing data management schemes in the application layer and kernel layer which prevent the high-performance storage devices from being exploited their full performance.

- We propose several optimization for data managements on high-performance storage devices and implemented them on linux kernel.

  - Efficient page reclaim, page recycling, for an efficient memory-mapped I/O.

  - Path finding scheme that performs the hash lookup operation in backward manner.

  - RLSM which is eliminated complicated data structures and conducts the compaction with append operations.

- Our experimental results show that our optimizations can increase the system performance in execution time and throughput when compared to the original Linux kernel. These results verify that our optimized system can manage the data more efficiently on high-performance storage devices.

1.2 Dissertation Structure

To support our researches and discussions, this dissertation is structured as following.:
In Chapter 2, we describe the background and motivations of our researches.
In Chapter 3, we will describe the our approaches of solutions, and we also describe the implementation techniques.
Chapter 4 evaluates our optimized data management schemes using various benchmarks.
Chapter 5 present related researches.
Chapter 6 concludes our works.

Chapter 2

Background and Motivation

2.1 Large scale systems

In modern society, the amount of data is increasing exponentially. Anyone can produce data and reproduce more of the data while sharing that on-line. Users can post some messages to the SNS to express their feelings or share information by talking to other users. Facebook [12], the global Social Networking Service, adds 3 billion new content per day, and another SNS, Twitter [13], says that they deliver 1.4 billion tweets a day. Since online commercial sites, such as Amazon and Taobao, are targeted at people all over the world, the sales volume of products during the day is very large. In addition, multimedia data have become popular and increased a lot as camera and audio devices are pervasive to people. Because multimedia data varies in size from a few Megabytes to a few Terabytes, and its structure is not standardized, its processing importance is also growing. The development of weather forecasting researches [14], particles researches [15], and genome research [16] has also increased the amount of

data that computers must process. In this way, the data of modern society has become enormous and diversified. With these vast amount of data, users require fast and accurate processing of such data. To meet these requirements, the computing systems environments must change.

As the amount of data that people produce and access has increased enormously, many research studies [2, 17, 18] have been conducted on data managing frameworks that can process such an enormous amounts of data. For example, Facebook has developed F4 [19] and haystack [20] as the data managing frameworks that efficiently manages large data. Because these systems must handle large data, they often form distributed systems [17, 21, 22, 23, 24, 25].

2.2 High-performance storage devices

The hardware technology of modern society is developing rapidly. In particular, the development in the storage devices is remarkable compared to others. For example, SSDs that appeared a few years ago have almost replaced the existing storage devices, HDDs, that have dominated the storage market. In recent years, the emergence of Storage Class Memory (SCM), which is superior to SSD in performance aspect, is expected to replace the SSD market.

The evolution of the host interface is also greatly reducing the data transfer overhead between hardware device and host software system. If SATA or SAS were suit to the block device such as HDDs, then the recently introduced PCI-e is an appropriate interface to the SSDs which have low-latency. To fully utilize the performance of these rapidly evolved storage devices, software changes are essential. This is because the hardware cannot be delivered to application through software due to the software overhead. Therefore, in order to use such high-performance hardware device, software changes are also necessary.

2.3 Exposed software overheads

2.3.1 Overhead of un-mapping in memory-mapped I/O

Paging is one of the important memory-management schemes in operating systems. It uses the main memory and the secondary storage to provide the illusion that the system has a larger memory than the physical main memory. When paging is used, frequently accessed data are located in the main memory and less frequently accessed data are located in secondary storage. Paging is generally used for implementing the virtual memory that allows very large address spaces and memory-mapped I/O (mmio). *mmio* is one of the ways to access data from files or file-like resources through load/store instructions of the CPU. Since *mmio* maps files to contiguous virtual addresses of applications, applications can access data by using in-memory variables. This feature makes *mmio* more advantageous since complex in-memory objects can easily be made persistent and the data in the main memory can be accessed faster without context switch. For these reasons, many software programs, such as NoSQL, and scientific applications use *mmio* to access data more efficiently, manage complex data structures, and permanently store data. In the HPC community, the Catalyst supercomputer co-developed by Cray and Intel adopts an optimized mmap system (DI-MMAP [26]) for flash-based SSDs. Many other researchers [27, 28] have studied about the impact of nonvolatile high-performance storage in data-intensive HPC environments. In Unix-like operating systems, the special system call, `mmap()`, is given for *mmio*.

Owing to the low and uniform latency for data access, demand for high-performance storage devices is increasing in many fields, such as OLTP and NoSQL, as well as applications with demand paging. Moreover, many research projects involving high-performance storage devices have been undertaken. Coburn

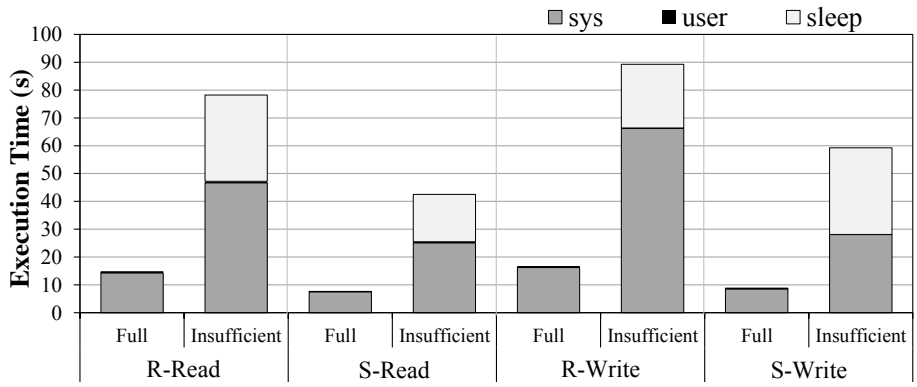


Figure 2.1 **Comparison of memory-mapped I/O performance between full and insufficient DRAM system with synthetic benchmark.** Full DRAM means 16 GB main memory and insufficient DRAM means 2 GB. The benchmark accesses the data set whose size is 8 GB with a single thread. In type of workloads, capital 'S' and 'R' means sequential and random workloads.

et al. [29] optimized a high-performance storage device and software stack for transaction rollback and recovery to provide high-performance transaction processing systems. Caulfield et al. [30] presented an efficient file access mechanism for high-performance storage devices. Yu et al. [31] proposed several optimization techniques for the device driver of a high-performance storage device and achieved peak-throughput of the device with the optimized device driver.

However, the performance of mmap can dramatically decrease if the working set for the mapped file with random access patterns is larger than the main memory. The actual performance drop is considerable as shown in Figure 2.1. We conduct the experiments on a high-performance storage device and used a synthetic benchmark. Our synthetic benchmark make the three processes share the data and access the whole data set three times with *mmap*.

The execution time of the application on a full DRAM system is much

shorter than that on an insufficient DRAM system, and the performance gap between full and insufficient DRAM is up to five times in the worst case. The graph bar titled 'sleep' shows how much the application sleeps while it is running. In the full DRAM experiment, the sleep portion is nearly zero in every workload type. However, with insufficient DRAM, the sleep portion is up to half of the total execution time in sequential write, which means that the application suffers from overheads. All the workloads with insufficient DRAM have a considerable portion of sleep in their execution time. This is mainly because page faults occur, and the OS kernel may put the process into the sleeping state.

To identify the main reason for this problem, we first analyze *mmio* and the virtual memory subsystem of Linux when the main memory is overcommitted. Once the `mmap` system call is executed to map data in a file to the address space of the application, the mapping is conducted in a lazy manner. When the application accesses an unmapped region of the address space, a page fault occurs and a free page is allocated for the data from storage. The virtual memory subsystem sets the mapping between the address space and the page with the data. As the application continues its computation, the number of free pages may decrease, and the system would suffer from a shortage of free pages. If the application fails to allocate free pages, the *page reclamation procedure* is triggered to raise the free page ratio of the system. The *page reclamation procedure* first selects victim pages to be reclaimed, invalidates the TLB entries and page table entries of selected pages, and unmaps the pages from the address space of the process.

When a page fault occurs, as shown in Figure 2.2, the system stops computation of the process and starts the page fault handler in the kernel mode by triggering the mode switch [32]. The page fault handler first tries to allocate a

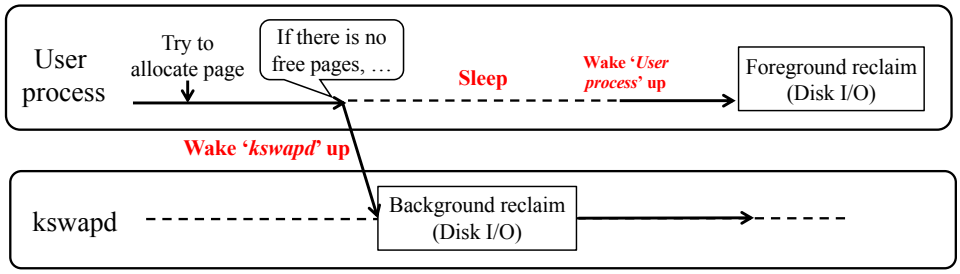


Figure 2.2 **Page reclamation procedure in the existing virtual memory subsystem of Linux.**

free page for data in a file. If the page allocation function returns a free page immediately, the system can continue its computation without further context switches.

However, if the system does not have free pages due to memory overcommitment, it executes the page reclamation procedure that is down to the `kswapd` process. To this end, the kernel puts the application process into the sleeping state and the `kswapd` process into the active state. The `kswapd` process then starts the page reclamation procedure. This asynchronous context switch from the application process to the `kswapd` process leads to a considerable portion of sleeping time.

On traditional HDD storage devices, the overhead of asynchronous context switches is negligible because the hardware latencies that include seek time and rotational delay are high. However, the overhead becomes critical to the application on low-latency storage devices based on flash-memory. Since NVM such as flash-based SSD has access latencies in the hundreds of microseconds, the time for accessing NVM is too short to hide the overhead of the asynchronous context switch. Thus, long reclaiming latencies become a burden to applications with *mmap*, and these behaviors hamper the delivery of the bare-metal

performance of high-performance storage to the applications.

If the `kswapd` process does not maintain pace for high demand for free pages, the page fault handler fails to allocate a free page even after the `kswapd` process finishes the page reclamation procedure. In this case, the page fault handler itself executes the *page reclamation* procedure, called *foreground reclaim*. The tasks of *foreground reclaim* are the same as those of *background reclaim*.

2.3.2 Overhead of redundant metadata operations

The local system in data processing frameworks, especially the local filesystem, is responsible for managing data as a 'file' and managing storage I/O. Recently, as the number of files in the local FS has increased considerably, interest in metadata and data content has increased as well; in particular, the overhead of metadata management has spiked [4, 33, 34, 35].

There are several metadata management mechanisms to handle filesystem objects such as files and directories. Unix and its variants have similar metadata management mechanisms. Especially in Linux, metadata is represented by a structure '*inode*' and a structure '*dentry*' in the local filesystem. Because these metadata are mapped with files and directories, the number of objects increases, and accordingly, the number of metadata is increased. Therefore, it is critical that such metadata is efficiently processed. To this end, existing systems use metadata caching techniques such as inode cache or dentry cache. Researchers in [33] are studying to reduce metadata overhead by breaking a 1:1 mapping between a set of metadata and a file. In [35], the authors propose a filesystem that manages metadata by using key-value store. Thanks to previous research projects on metadata for the local filesystem [4, 34], the metadata management overhead can be reduced.

However, we found that there are still inefficiencies in the metadata man-

agement mechanisms in existing Linux FSs; lookup operations in the hash table of the dentry cache are redundantly called. This overhead worsened as the directory depth became deeper and the number of files increased.

To find out the root cause of inefficiencies, we delve into the VFS layer in the Linux kernel to understand the path traversal mechanism. In order to access a file through its name, we should conduct a file path traversing. 'Traversing' means examining the file location along with the subdirectories in the directory hierarchy. The path traversal mechanism of the Linux kernel is described in Figure 2.3. Dentry structure describes the directory hierarchy of the filesystem by indicating upper- and lower-level dentries. In the dentry cache, each dentry is hashed with its respective hash value. The hash value is derived from the pair of the corresponding directory name and parent name. Assume that if we want to access the '`test`' file in '`/a1/b2/c3/d4`', we first find the dentry corresponding to '`/`', which is the starting point of the path. Then, we find subdirectories, such as '`a1`' and '`b2`', in a regular order. In this manner, we finally find the dentry for '`d4`,' which is the target dentry. Traversal latency depends on the path depth of the file because a deep path triggers more lookups. For example, when finding a file in '`/a1/b2`', we traverse only two levels. On the contrary, when finding the `test` file in '`/a1/b2/c3/d4`', we must traverse four levels. This mechanism is filesystem-specific and is implemented in the VFS layer; regardless of the filesystem, the path traverse is performed according to this mechanism. The path traversing usually occurs in the pre-processing phase before a read/write operation, such as `open()` system call.

Redundant lookup occurs when similar paths are looked up. Assume that we first find a file in '`/a1/b2/c3/d4`' and then find a file in '`/a1/b2`'. Although the dentries for '`a1`' and '`b2`' are cached because of the first path traversing, we should lookup the components of the second path from the start to check

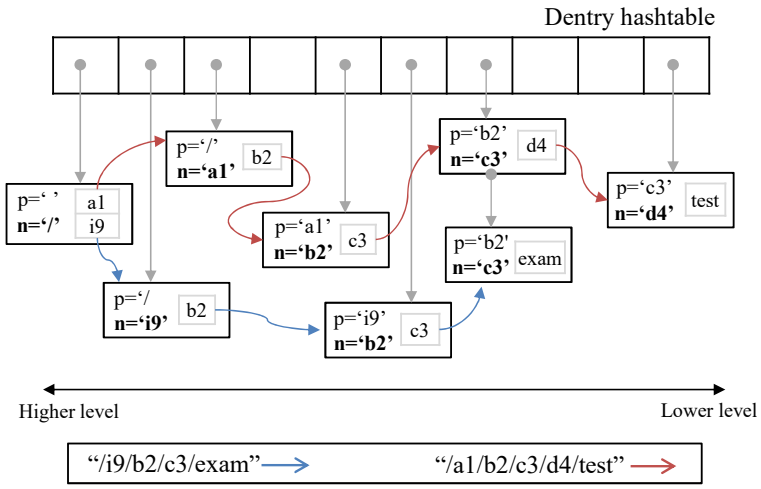


Figure 2.3 **The Linux path lookup mechanism.** The *‘block’* refers to a dentry structure and *‘p’* indicates the parent’s name and *‘n’* represents the dentry’s own name.

whether it is cached or not. In other words, even if the target dentry ‘`b2`’ is already cached, the kernel does not demonstrate this.

To access the overhead of redundant hash table lookups, we measure the execution time to `open()-read()-close()` a single file according to the path depth of the file. This preliminary experiment is presented in Figure 2.4. As shown in the figure, as the file path depth increases, only the time for the `open()` system call increases proportionally. This increased execution time in the `open()` system call derives from redundant hash table lookups. Many usually-used workloads such as *find, du, tar, git* commands in Linux, might suffer from this path traversing overhead. If the last cached component of the path among the cached dentries is found more efficiently, we can reduce the number of unnecessary and redundant dentry cache lookups.

This overhead of redundant lookup operations is more visible when the data

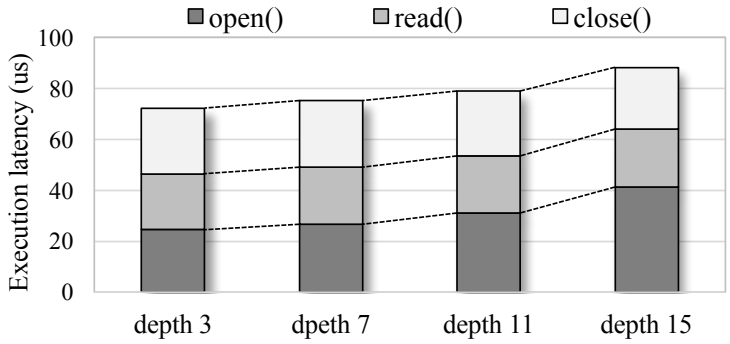


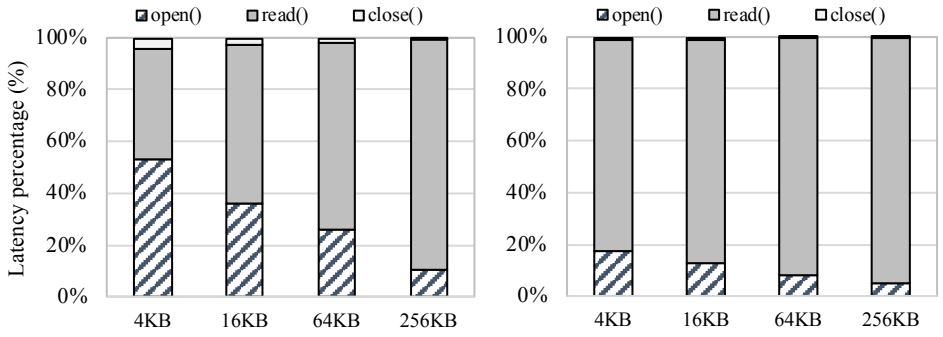
Figure 2.4 **The latency breakdown of one read request into `open()`, `read()`, and `close()`.** '*depth 3*' means that we open, read, and close the file in a directory of the level 3 (e.g., '`/a1/b2/c3`').

is stored in high-performance storage devices. This is because that data access latency becomes low on that kind of storage devices. To measure the effect of storage types, we conduct the same experiments with Figure 2.4 with different storage types, NVMe optane and SATA 850pro.

As can be seen in Figure 2.5, the percentage of path traversing in `open()` systemcall with high-performance storage type (NVMe Optane) is much larger than that with relatively slow storage (SATA 850pro). With 4KB file, the percentage of path traversing is about 50%. Therefore, we have to eliminate the path traversing overhead to exploit the full performance of high-performance storage devices.

2.3.3 Overhead of LSM algorithm in key-value store

The Log-Structured Merge(LSM) tree [36], one representative example of log-structured approaches, has been used as a key data structure of many key-value stores [37, 38, 39, 40, 41, 42]. LSM can deal with write-intensive workloads by cascading data from smaller, high-performance stores such as RAM to larger,



(a) NVMe Optane (b) SATA 850pro

Figure 2.5 **Percentage of path traversing in open() systemcall with different storage types.**

less performant stores. LSM-based storage systems typically insert and store data in the memory buffer, and the data in the memory buffer are synchronized or flushed to persistent storage devices in bulk when the memory buffer is full. For the synchronization, LSM-based systems create a file and write the data to the file in a log-structured manner. Write throughput in LSM-based systems is relatively high since data randomly inserted are sequentially written to persistent storage devices.

To take the advantages of LSM data structures, industry and academia have developed key-value stores based on LSM data structures. For example, Google developed levelDB [37], and many research groups proposed its variants such as rocksDB [38] or HyperLevelDB [40]. Cassandra [39] is one of the popular LSM-based key-value stores. In addition, Facebook message systems [43, 44] use HBase [41], and the recent version of mongoDB [45]uses the wiredtiger as the backend storage engine.

In the LSM tree, data are written to the storage sequentially even the data are requested in the random order. The complicated operations, such as re-

ordering of data, are deferred to elevate the insertion throughput. The LSM tree generally works as follows: first, the key-value pair is written in the memory buffer by the insert-order. When the memory buffer is full, the LSM tree sorts the key-value pairs in the memory buffer and synchronizes the buffer to storage in batch. During synchronization, the key-value pairs are sequentially written to a file. If the procedure is repeated, several files with overlapped ranges can be created in the storage, which leads to high read amplification. The LSM tree defers the complicated operations, such as re-ordering the key-value pairs in multiple files with overlapped ranges, for high write throughput. These deferred operations are called *compaction*. Therefore, the LSM tree can handle explosive insertion of data [36].

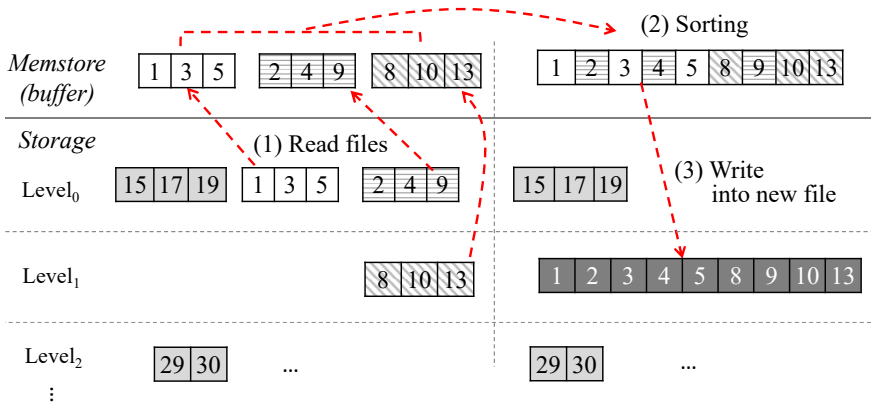
LSM algorithm is devised assume that underlying storage is very low so that the data should be sorted in memory buffer. Therefore, they usually use the complicated in-memory data structure to sort the data, such as Skip-list. Therefore, in memory buffer, the data is in sorted order with complicated data structure, and data in on-storage files is also sorted order.

With this complicated data structure, the LSM tree has a weakness: *write amplification* during the compaction. Basically, the compaction process merges several files whose ranges are not disjoint. To do this, selected files are loaded into the memory buffer, and the key-value pairs in loaded files are resorted. The sorted key-value pairs are then written into a new file sequentially. With the compaction process, a single key-value pair from an application is written into storage several times. As a result, the write volume to the storage is larger than the actual size of the data set. In addition, memory consumption is increased because all the key-value pairs of the files must be loaded into the memory buffer to be resorted.

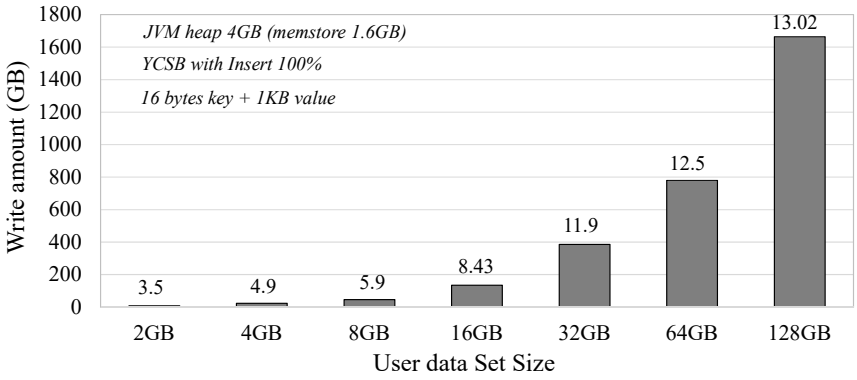
Figure 2.6 (a) shows the basic compaction procedure in LSM [36]. The com-

paction is conducted by merging the files with over-lapped ranges. Therefore, the files with data {1, 3, 5}, {2, 4, 9} at Level_0 and the file with {8, 10, 13} at Level_1 are included in compaction. They are loaded into the memory buffer, sorted with keys, and written into a new file at Level_1. Figure 2.6 (b) shows the write amplification ratio. The baseline experimental results are measured with HBase on SSD devices, and detailed experimental environments are explained in Section 4. As shown in the figure, the write amplification ratio grows as the data set increases. When a data set is 128 GB, the write amplification ratio is about 13. High write amplification can reduce the lifetime of the SSD storage and increase the storage costs. Therefore, an efficient compaction algorithm involving less I/O amplification is required.

When using the high-performance storage devices, we can use more simplified data structures because data access latency is low. Therefore, the optimized LSM algorithm with considering high-performance storage devices is needed.



(a) Original Compaction algorithm in LSM tree



(b) Write amplification result in HBase

Figure 2.6 **Basic compaction algorithm and write amplification results in the original LSM tree.** In (a), the array refers to the data set in a file, and the number in the square refers to the key. In (b), the value at the top of the bar is the write amplification ratio.

Chapter 3

Design and Implementation

3.1 Memory-mapped I/O optimization

In this section, we propose *Page Recycling*, an optimized mechanism to make page reclaiming more efficient. In the recycling path, the process searches its own address space to find victim pages for page recycling. We also add the per-core list for recycled pages to guarantee the allocation of recycled pages in a multi-threaded environment.

3.1.1 Design

When the page fault handler cannot allocate a free page for *mmap*'ed data, the kernel puts the application into the sleep state and wakes up the `kswapd` process to start page reclamation. As explained in the previous section, the context switch becomes a critical overhead when we use fast storage devices. Thus, it is necessary to devise a new solution that performs page reclamation with fewer context switches.

We add one more step to get a free page with reduced context switches, which is called *page recycling*, before the page fault handler starts the page reclamation procedure. In the recycling path, the page fault handler uses a *local scan* that considers only the address space of the *mmap*'ed file and chooses the LRU pages to recycle. Figure 3.1 describes the scope of the local scan. The page fault handler searches the victim pages only in the memory-mapped portion of the file in the virtual address space; it then frees and recycles the victim page. This can reduce the number of tasks compared to the existing page reclaim path.

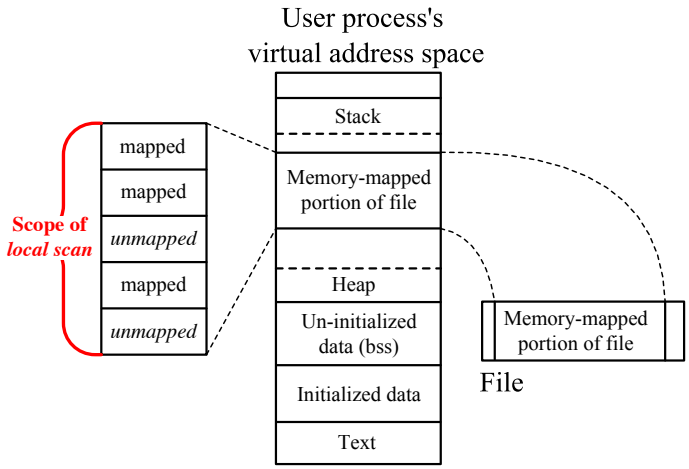


Figure 3.1 **The scope of the local scan for the page recycling.** If the user process accesses the *unmapped* region, it occurs page fault.

By recycling pages of *mmap*'ed regions, page recycling does not conduct irrelevant I/O and does not consider the free page ratio of the system. Thus, our recycling procedure can be executed without context switches, and it reduces the possibility that the page fault handler executes the original reclamation procedure with context switches. If the page fault handler fails to recycle the page

in its own address space, it performs the original reclaiming path to allocate a new free page.

Clean-Only Recycling (COR)

When the page fault handler selects the victim pages in the recycling procedure, we first select *clean* pages to simplify the recycling path. The page fault handler unmaps selected pages and releases pages to recycle *mmap*'ed pages. We call this scheme *Clean-Only Recycling* (COR). COR shows better performance than traditional *mmap* in a memory-overcommitted environment since it reduces context switches for waking the `kswapd` process. However, the performance of the Clean-Only Recycling scheme is still much worse than the performance of *mmap* with the full DRAM system, especially under a write-intensive workload. This is mainly because the Clean-Only Recycling scheme does not consider dirty pages but clean pages. To improve the performance of *mmap*, we consider dirty pages as well as clean pages.

Full Recycling (FR)

In the COR scheme, we target only clean pages to recycle because the page fault handler can throw away page content that is duplicated in the backing storage. Thus, COR does not require write operations. However, under the write-intensive workload, COR does not provide much benefit since it is highly probable that insufficient clean pages exist. When there are no clean pages in the *mmap*'ed region, the kernel wakes up the `kswapd` process, which leads to context switch.

To address this issue, we consider dirty pages when the page fault selects pages in the recycling path. If the selected page is dirty, an additional write operation is necessary since the in-memory page is up-to-date. We add the

page-out operation to the recycling path to write-back the content of dirty pages. Since the pages of the *mmap*'ed file are file-backed pages, the write-back requests modify the file contents. The page-out operations pass through the underlying file system by using the file system API, such as `ext2_writepage()`. We call this scheme *Full Recycling* (FR) because it targets every page of the local address space regardless of its state. It reduces context switches of the existing reclaiming procedure since FR can recycle enough pages.

Per-core list for recycled pages

The page fault handler reclaims a number of pages during the recycling procedure. We call the number of recycled pages *batch size*, and the default batch size is 32. The recycled pages are added to the global free page list that can be accessed by multiple threads. In multi-process environments, many processes can access the global free page list. If the process adds the recycled pages to the global free page list, the recycled pages may be used by other processes. In the worst case, the process that consumes its time quantum for the page recycling procedure still suffers from a lack of a free page. This leads the page fault handler to start the existing page reclamation procedure that requires context switches. This situation decreases the efficiency of page recycling.

To address this issue, we make a private page list per core, named the `recycling_list`. This list belongs to the per-core structure, and the page fault handler returns the recycled pages to the private list of the core that the process runs on.

The overall view of the page reclamation procedure with page recycling is described in Figure 3.2. In the modified page reclamation procedure, we add the recycling procedure before the original procedure to reduce context switches. We

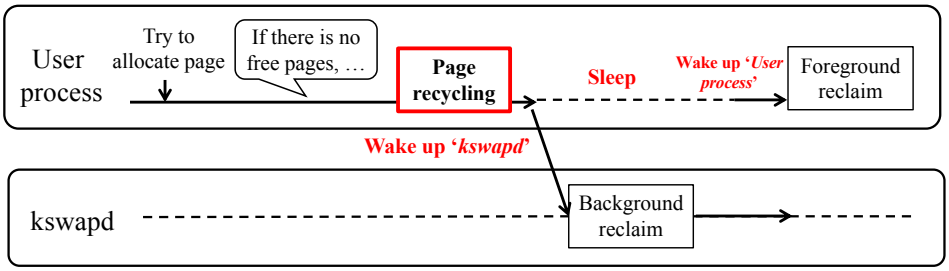


Figure 3.2 **Our page reclamation procedure.** We add the page recycling procedure with additional optimizations the existing reclaiming path.

increase the possibility of page allocation in the recycling path with the aforementioned schemes. This mechanism can reduce the sleep portion of execution time because the foreground process does not need to wake the background reclaim. In addition, the frequency of falling into the foreground reclaim path would be reduced.

3.1.2 Implementation

The `__alloc_pages_nodemask` function of the Linux kernel is the core part of the buddy allocator. To handle the page allocation request, the function first calls the `get_page_from_freelist` function, which tries to allocate pages from the free page list of the buddy system. If it fails to obtain a page from the `get_page_from_freelist`, it calls the `__alloc_pages_slowpath` function. The `__alloc_pages_slowpath` function invokes the original page reclamation procedure that incurs asynchronous context switches (sleep/resume).

To implement our page recycling scheme, we add our function (the `recycling` function), which is executed before the `__alloc_pages_slowpath` function to avoid asynchronous context switches. Our `recycling` function is largely divided into two parts; `select_pages` function and `unmap_list` function. The

`select_pages` function selects victim pages for recycling. In this function, we implemented COR and FR. In the `select_pages` function, we borrow lock usages of the `__alloc_pages_slowpath` function to avoid simultaneous accesses of shared data. In the original Linux kernel, the `__alloc_pages_slowpath` function uses the `lru_lock` mutex of `struct zone` to obtain candidate pages (32 pages) from existing page lists of the kernel. Our function obtains candidate pages from the `inactive_file_list` list in the same way and it selects victim pages among candidate pages [1]. The `unmap_list` function recycles selected victim pages and returns them to the per-core private list. By adding our function in the page allocation procedure, we largely reduce the number of asynchronous context switches.

To implement the private list, we slightly change the sequence of page allocation. When a page fault handler tries to allocate a free page, it first checks the private list of the core that it is running on. If a free page is available in the private list, the page can be returned to the page fault handler directly; otherwise, it starts the page recycling procedure for free pages. This procedure on the `recycling_list` is inserted in front of our `select_pages` function, and our `unmap_list` function adds reclaimed pages to the `recycling_list` (100 LOC). To implement the full recycling scheme and the per-core list, we modify the Linux kernel about 1000 LOC totally.

3.2 Metadata operation optimization

In this section, we describe the optimized mechanism of path finding, *backward* finding mechanism. We also present solutions of side-effect problems such as

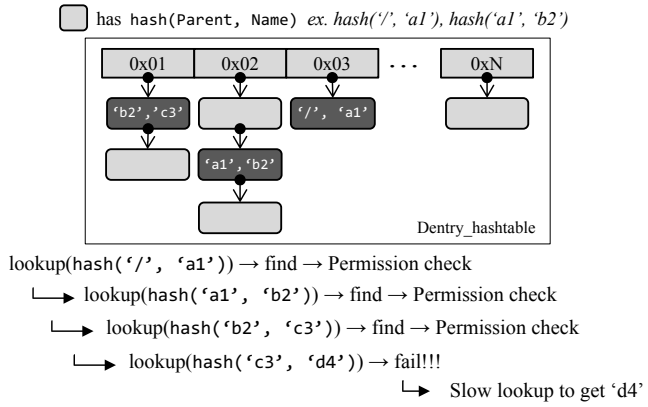
[1] Note that other page replacement policies such as CFLRU [46] can be adopted for page recycling. In this study we consider the default replacement policy of the Linux kernel since the main motivation of our study is the performance degradation from asynchronous context switches.

permission guarantee and the other operations related path finding.

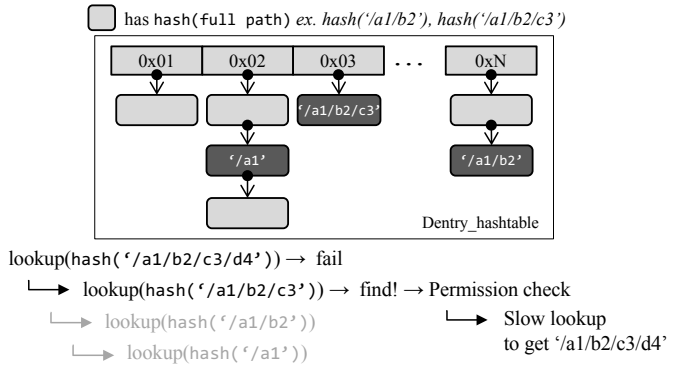
3.2.1 Design

To reduce the path traversal overhead, we propose the backward path traversal mechanism to find the last cached path component in dentry cache more efficiently. In our optimized mechanism, path traverse starts at the end component of the object path. If the dentry indexed with that path component is not found, the dentry is searched which is indexed with one higher path component. In this manner, we can find the dentry indexed with the lowest path component more easily among many cached dentries. This approach can reduce the number of dentry hash table lookups. However, with the current hashing mechanism of the dentry cache, finding the dentry in reverse order has a limitation, which is described in Figure 2.3. Assume that we now traverse the object path with the original finding mechanism to understand the limitation. When we want to traverse the object path '`/i9/b2/c3`' in a backward manner, we first try to find the '`c3`' dentry in dentry cache. In that case, we cannot distinguish the target dentry between the two cached '`c3`' dentries. (One is for `/a1/b2/c3` and the other is for `/i9/b2/c3`.) This is because that dentry is indexed by hash value with the pair of {*parent name, its own object name*}. Therefore, we cannot apply the backward path finding technique to the existing path-finding hashing mechanism. To solve this problem, we hashed dentries in the dentry hash table with the full path of the object. With this full path hashing, even though any two dentries could have the same parent name and its own name, we can recognize the target dentry because their full paths are always different.

The comparison between the original path finding mechanism and our optimized mechanism is shown in Figure 3.3. In this example, the dentry for '`d4`' is not cached yet and not stored in the dentry hash table. In the original lookup



(a) Original lookup



(b) *Backward* lookup

Figure 3.3 **Example of file path traverse.** The target directory is '/a1/b2/c3/d4'.

mechanism (Figure 3.3 (a)), we perform dentry hash table lookups in regular order to obtain the target dentry. When we fail to find the cached dentry, slow lookup, which might involve storage I/O, is conducted. However, in the backward finding mechanism (Figure 3.3 (b)), we hashed the dentry with the object's full path. In this mechanism, we start path traverse with the target object's full path '/a1/b2/c3/d4'. If it fails, the dentry is searched which is indexed with a shortened path whose last component is trimmed ('/a1/b2/c3'). When

the searching succeeds, we should conduct slow lookup to obtain the one lower-level dentry for '`d4`' based on the found dentry. By detecting the lowest dentry with this mechanism, we do not need to lookup the dentries for the higher level path components. This is the key contribution of the backward path finding mechanism.

Many studies suggest the full path indexing in their papers [47, 48]. However, the studies use the full path indexing only for finding the final target dentry. If they cannot find the final target dentry with full path indexing, they must turn back to the original mechanism. This shows that the previous studies cannot fully take advantage of full path indexing.

Permission check

In the Linux kernel, a filesystem object has three permission types (Read, Write, Execute) based on three users (Owner, Group, Others). In order to be granted to access to a filesystem object, the user must check the permission and user type of every path component. Since this permission check is a part of path traversing, it is done in regular order, that is from the beginning of the object path.

In the backward finding mechanism, we directly search the target dentry and omit searches of higher-level path components. This mechanism means that we also omit the permission check of the higher-level dentries. We, therefore, cannot guarantee whether accessing to the target dentry is allowed or not. This may trigger the permission violation. To guarantee the permission grant of the target dentry, we add *'permission-granted list'* to the dentry structure. This list stores the granted users' ID, presented in Linux as 'UID', to access the object.

The permission check is performed as following: 1) When a dentry is not cached yet, the original lookup mechanism is conducted. In the original lookup

mechanism, the permission is checked in forward for each path component. If the access to the dentry is granted, the corresponding ID is stored in the dentry's permission-granted list. 2) When a dentry is already cached, but the user cannot access the dentry because its ID is not in the permission-granted list of that dentry, the user must traverse the path components in backward manner until finding the dentry which it can access. And then, the user proceeds the permission check for lower dentries from that dentry.

If a user changes the permission of an object directly by using Linux command (ex. *chmod*), permission changing has to be conducted synchronously by traversing the subtree whose root is the changed object. This synchronous permission change may take relatively long execution time although it varies according to the size of the subtree. However, this kind of operation is not conducted frequently contrary to other metadata-related operations. Therefore, we focus on the more frequently conducted metadata-related operation, such as read operation, and reducing latency of the operation.

Algorithms 1 and 2 show the pseudocode of the dentry-finding algorithms with backward finding mechanism. In Algorithm 1, the ***else*** part (from line 8 to line 13) is the original dentry-finding algorithm in Linux. To be with backward finding mechanism, we added function `backward_ finding()` and ***if*** part (from line 1 to line 6). The function `lookup_fast()` (in line 9) means looking up the dentry hash table. Since this is an in-memory operation, it is named with *'fast'*. On the contrary, the function `lookup_slow()` (in line 5 and 11) means searching data from storage. This may be *slow* because of storage I/O.

In Algorithm 1, the target dentry's full path is passed to the function `backward_finding()` in Algorithm 2. The *hash_value* is created by using the passed full path and it points out the offset of the dentry hash table. We lookup

Algorithm 1: Dentry-finding algorithm with backward finding

```
1  n ← full name of the path;
2  dentry ← backward_finding(n);
3  if dentry ≠ the target dentry && perm_fail_flag not set then
4  |   for dentry == the target dentry do
5  |   |   dentry ← lookup_slow();
6  |   end
7  else
8  |   for dentry == the target dentry do
9  |   |   dentry ← lookup_fast();
10 |   |   if dentry == null then
11 |   |   |   dentry ← lookup_slow();
12 |   |   end
13 |   end
14 end
15 return dentry;
```

the dentry hash table to find the name-matching dentry. If the name-matching dentry is found, the permission check follows. When permission check succeeds, Algorithm 2 returns the dentry and back to Algorithm 1's line 8. If the name-matching dentry is not found, the last component of the path is cutoff (in Algorithm 2 line 14). The ***while*** loop (in Algorithm 2's line 2) is conducted again using the shorten path. When we found the name-matching dentry but fail at permission check, we set *perm_fail_flag* and repeats the ***while*** loop after cutting off the last component of path. This loop is repeated until we found name-matching and permission-granted dentry. And then, we return back to Algorithm 1's line 7 and, for permission checking, look up the dentry hash table again. We also slightly changed the original path because we have to update the permission list. By checking the dentry with regular order, the permission is examined and the permission list is updated when permission granted.

Algorithm 2: *backward_finding()* algorithm

```
1  f_name ← full name of the path;
2  while true do
3  |  hash_value ← hash(f_name);
4  |  for matching dentries in hash table do
5  |  |  if dentry's full name == f_name then
6  |  |  |  if the UID is in the dentry's pg_list then
7  |  |  |  |  return dentry ;
8  |  |  |  else
9  |  |  |  |  perm_fail_flag ← 1;
10 |  |  |  |  break;      // go to line 14
11 |  |  |  end
12 |  |  end
13 |  end
14 |  f_name ← trim the last component of f_name ;
15 |  if f_name == '/' then
16 |  |  return null;
17 |  end
18 end
```

Path-finding-related operations with backward finding

The backward finding mechanism, described in the previous subsection, only requires the modification of dentry structure. Therefore, this mechanism can be easily integrated with other path-finding-related operations. As an example of that operations, we now present the operations with *relative path*, *symbolic/hard links* and *mount*.

1. *Relative path* : Users may find an object with relative path; for example `/a1/b2/../bb/c3` which actually means `/a1/bb/c3`. These relative paths have different hash values from the actual absolute path. Therefore, we should pre-parse the relative path before the passing it to the function, `backward _finding()`. With the pre-parsed path, we can smoothly conduct backward finding mechanism.

2. *Symbolic and Hard links* : In the case of symbolic/hard links, dentry aliasing is conducted to link two different dentries. Dentry aliasing of symbolic links creates a dentry for the aliased name which points the object's original inode. In the case of hard links, we create a dentry for the aliased name and also make the dentry which points the duplicated inode of the original object.

3. *Mount point* : When mounting a device as a certain filesystem on the existing system, we consider the mount point as a 'prefix'. The objects in a new mount point are indexed with their full path without a 'prefix'. In contrast, we just conduct the backward searching and concatenate the 'prefix' to the name-matching dentry.

3.2.2 Implementation

To demonstrate that our proposed mechanisms have benefits for dentry lookup performance, we implemented that into the Linux kernel.

In order to modify the Linux's lookup mechanism of a dentry cache, we modified several functions in VFS layer, which are functions that perform path traversing. In particular, we modified `path_init()` function and `link_path_walk()` function. These functions lookup hash table of dentry cache by iterating each path component. To adopt the backward finding mechanism, we reorganize the order of path traversing to lookup the dentry in a backward manner. Also we added some auxiliary functions to support other path-finding-related operations.

For permission, we implemented permission checking procedure by adding a *'pg_list'* (permission-granted list) to the dentry structure which is an integer array. With this array, we could store the list of permission-granted users' IDs. (Users's ID is represented as an integer in Linux kernel.) If the number of

granted users exceeds the size of the list, randomly selected user ID is evicted. The size of permission-granted list is adjustable, and can be parameterized with experimental results. In our following experiments, we set the size of the list 5.

We also added a character array to make a room for storing the full path of the dentry. To do this, we modified the `fs/dcache.h` where the dentry structure defined. In our prototype system, the dentry structure is 384 bytes, including all the necessary space, which is twice the size of the existing dentry structure.

All of modified lines of code reaches at about 1500 lines that signify the simple code change and considerable performance improvement.

3.3 LSM algorithm optimization

In this section, we describe the Ranged LSM (RLSM) tree and its compaction procedure to reduce write amplification. We also describe the partitioning of files to decrease overhead of read operations, including scan operations.

3.3.1 Design

RLSM does not require data to be sorted strictly to reduce the sorting overhead during the compaction process. To this end, we use the hash-based data structure for storing data. We hash the key and store the data in the hash table according to the key's hash value. Whenever the memory buffer is full, the data is flushed into a file with the hashed order in a hash table, not the key order. These files are classified as *flushed files* in the RLSM, and the data ranges of flushed files are naturally overlapped.

If the data can be flushed in the unsorted order, one burden of the compaction process is relieved since the data is just appended to the existing file without sorting. We divide the files participating in the compaction process into *victims* and *targets*. In RLSM's compaction, only data in victim files are loaded

to the memory and rewritten (appended) to the target files. Compared to the original LSM, in which all victim and target files must be loaded to memory in order to sort and rewrite all of them to a file, we reduce memory consumption and the number of rewrites.

This mechanism takes advantage of fast storage device, especially fast random access. The appending data to the target files can arise huge amount of random access to the storage devices. On the fast storage devices, the random access also has the low-latency. Therefore, the appending data with random locations can have improved performance.

However, overlapped files with flushed data can cause a read amplification. Because we cannot assure the locations of the desired data, we must check all the existing files to find the entry. We address this problem by conducting the compaction according to the data range of each file. In RLSM, each existing file has its key range, which indicates the minimum and maximum key-value pair in the file. During the compaction, by using hash, the key-value pairs in victim files are appended at the end of a target file whose key range includes the key-value pair. With this mechanism, the files' ranges are not overlapped after compaction. The data is arranged roughly at the file level, even though the data in the files are not sorted. The files created during our compaction process are called *compacted files*. In the subsequent compaction process, these compacted files become the target files.

We classify the logical layout of storage into just two levels: Level_0 for the flushed files and Level_1 for the compacted files. The simplified logical level of the storage can reduce the I/O amplification by maintaining multiple levels in other LSM variants.

Our proposed RLSM tree functions as follows:

1. First, key-value pairs written from user requests are stored in the memory

buffer with a hash-based data structure. Therefore, key-value pairs are not kept in the sorted order.

2. If the memory buffer is full, the key-value pairs are flushed into a file in permanent storage in a log structure. This implies that key-value pairs are written into a file sequentially, not in key-order, but in hash-order. These flushed files are created whenever the memory buffer is full, and they are located at Level_0. The key ranges of files at Level_0 can be overlapped because they are created by merely flushing from the memory buffer.

3. When the number of files in Level_0 exceeds the threshold, compaction algorithm is triggered. At first, RLSM checks whether any files are located in Level_1. If there is no file, RLSM selects the victim file and just move them down to Level_1. By doing this, RLSM sets the key range in Level_1.

   If there are some files in Level_1, RLSM selects the victim files in Level_0, and then, appends each key-value pair in the victim file to the files in Level_1 according to their key range. When the appending is finished, the files in Level_1 updates their metadata and the victim file can be eliminated. We set the following criteria for selection of victim files at Level_0:

   - Union Set : If there is a file at Level_0 with a small range that can be included in a larger range of a file at Level_1, RLSM selects that file as a victim. This is because that compaction can be completed by just concatenating the files without changing the target file's range. This can reduce the amount of I/O required when combined with the implementation technique described in Section 3.3.2.

$$Range\_of\_the\_Victim \subseteq Range\_of\_the\_Target \tag{3.1}$$

- Density : If there are no union sets, the next criterion is density. RLSM selects the file with the heavy density as a victim to be compacted. This is because the key-value pairs in the file with the heavy density are clustered, so there is a strong likelihood they will be appended to the same target file. We can reduce the amount of I/O by creating one append operation with the clustered key-value pairs.

$$Density = Total\_number\_of\_KVs/(Max - Min) \tag{3.2}$$

4. Although there are files at Level_1, the data to be compacted cannot find the proper range to be appended. If there are several key-values for which the range cannot be found, RLSM creates a new file with a new range. Otherwise, RLSM appends them to the existing files by changing their key range.

5. When read operations are requested, the file with the corresponding key range is first searched. If we find the file where the required key-value pair belongs to its range, we load the file into the memory buffer. After loading the file into the memory, we find the data in that file with the hash function.

Figure 3.4 shows the compaction process of the RLSM. In this figure, each key-value pair is managed in hashed order. We write the only key in the figure. When the number of files at Level_0 exceeds the threshold (the threshold is 3 in this example), RLSM starts the compaction process. (a) At this time, since there is no file at Level_1, RLSM selects two victim files and appends one to the other to move them down to Level_1. In this example, RLSM selects the first and second files at Level_0 as victims because they are denser than others (the first file's density is 0.5 and the second file's density is 0.75.). By appending

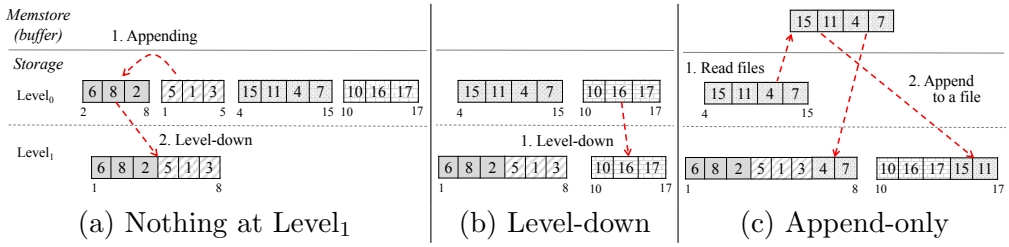


Figure 3.4 **Compaction process using Ranged LSM trees.** The numbers in the squares indicate the key, and the array of keys indicates the file stored in the storage. The small numbers below the arrays indicate the data range.

key-value pairs in the second file to the first file, the key range of the file is changed into the range [1–8]. (b) Because there is now a file at Level_1, we first search for the file with the range that is a union set of the Level_1 file. However, there is no suitable file. Therefore, the file with range [10–17] whose density is 0.429 is selected as the next victim to be compacted. The file is just leveled down to Level_1 and the new range is created because the key-value pairs in the file cannot find the range to be included. (c) When compacting the last file, with range [4–15], RLSM first finds the target file with the correct range for each key-value pair. The first two data items {15, 11} are appended to the file with the range [10–17], and the last two items {4, 7} are appended to the file with the range [1–8]. With this mechanism, the files at Level_1 are not overlapped and have distinct ranges, even though the key-value pairs in the files are in random order. Now assume that data {9, 23, 18} must be compacted, but none of them can find their range among the existing files at Level_1. In this case, the key-value pair {9} is appended to the file with the range [10–17] because it is the only key-value pair in that range. And the items {23, 18} create the new file with a new range.

Load balancing for files

Because we compact the files with their key ranges in RLSM, the file size can be skewed. In that case, read performance may be decreased, given the uncertainty of data location. Therefore, we need load balancing which indicates adjusting the amount of stored data in the file.

In addition, this load balancing is needed for processing the update or delete operations. In the LSM, the update and delete operations are considered to be insertion operations. In particular, the delete operation is a insertion operation with a tombstone flag. In our RLSM, we compacted files as if every operation is an insertion, and the update and delete operations are handled at load balancing time.

Partitioning To partition the files, we track the size of each file. When file size exceeds the threshold, load balancing occurs and partition the file into two small files. To partition the file, we use the *linear selection algorithm* [49] that re-orders the data in the array to find the k^{th} smallest element. In RLSM, we find the *(the number of key-value pairs in the file/2)*th element to keep the balance between key range of a file and the size of a file. The chosen pivot is used to partition data in the file, and the worst time complexity of our partition procedure is *O(n)*.

Figure 3.5 shows the load balancing of RLSM with selection algorithm. When we want to split the file with key range [1-29], we load the file to the memory buffer and apply the linear selection algorithm. We find the 5^{th} item because the number of key-value pairs is 10 in the array. In the example, the pivot (5^{th} item) is '5', and we can partition the array with the selected pivot. After the partitioning, we can get files with range [1-5] and [6-29]. The files' ranges are not divided equally, but the size of the file is equal to each other.

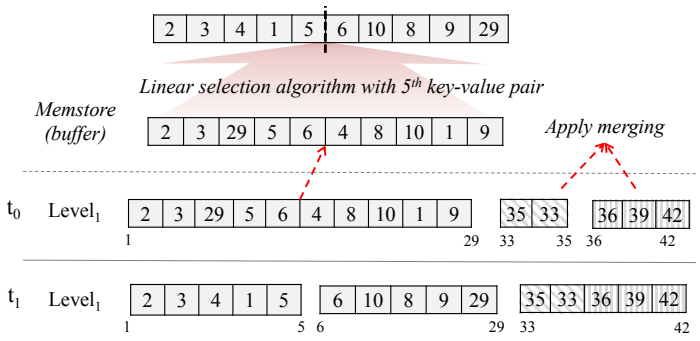


Figure 3.5 **Load balancing process in Level$_1$.**

Finally, we write the two parts to each file in Level$_1$.

Merging Merging refers to concatenating the files with small key ranges. During load balancing, the files with very small ranges are examined. When we select a file with a small range, we also select a target file with a smaller range or larger range. The two files are then merged, and their range is updated. In Figure 3.5, the file with a range of [33-35] is selected as a victim file to be merged. Furthermore, for the target file we can choose between the file with the smaller range [1-29] and the file with the larger range [36-42]. Because the file with a range of [36-42] has less data, we select the file as a target. Finally, two files with a small key range [33-35] and [36-42] are merged into one file.

3.3.2 Implementation

Our implementation is based on HBase 1.2.6 and Ubuntu 14.04.

Appending data

When the data resides in the memory buffer, the hash value of the key is used to return the corresponding value. When the memory buffer is flushed into a file in the storage, the file offset of the data is recorded into the hash table of keys.

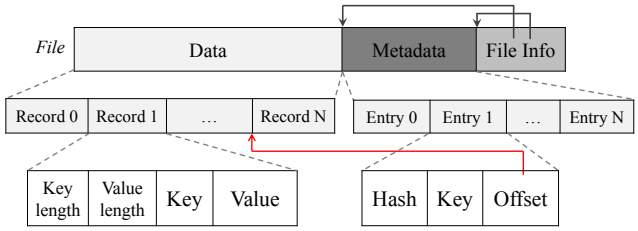


Figure 3.6 **File layout in RLSM.**

After the data is written, the metadata, including the hash table, is written to the file. As shown in Figure 3.6, the *'File info'* section is written behind the two sections (data/meata section).The last section maintains the offset of the data section and the metadata section.

When we append data to the target file, we first load the *'File info'* data. Then, RLSM reads the metadata of the target file by using the *'File info'* data. It writes the new data at the last position of the stored data. At that time, the hash table for appended data is merged to the existing hash table in the memory buffer. After the data appending process is finished, the merged hash set and *'File info'* data is written at the end of the file.

When appending *updated* (*deleted*) data to the existing file, the updated (deleted) data causes a hash collision because it is inserted with the same key. Since the Java `HashMap` API does not permit key duplication, the new data set overwrites the old data set. With this mechanism, we obtain the information of the old data set, and manage it with a separate list as invalid data. The invalid data is handled during the load balancing process.

Victim selection

As mentioned above, the first criterion is a union set when we select the victim files to be compacted. If the range of the victim file at Level_0 is a subset of the range of the file at Level_1, we do not conduct actual I/O for appending

the data. In this case, we can perform the compaction by simply linking these two files and merging their metadata. Therefore, only the first file's metadata is updated and rewritten. The hash set of metadata now includes file information, where the data is stored as well as the file offset. With this technique, we can reduce the amount of I/O even further. However, as the number of linked files increases, it may cause random reads in the storage device. A periodic cleanup is therefore needed, which is conducted during the load balancing process.

Data indexing

The original HBase uses a block-level Bloom filter to index the stored data block. We still maintain the Bloom filter, given the expectation of a high hit ratio in the read block cache. When the operation is missed in the read block cache, the files stored in permanent storage should be investigated. In RLSM, the data of the files at Level_0 are indexed with a key-level Bloom filter. This is because the key ranges of files at Level_0 are overlapped. The memory consumption for that Bloom filter is not big, because the amount of data at Level_0 are limited.

In addition to the Bloom filters for the files at Level_0, we use the indexing tree for files at Level_1. Each node of the indexing tree indicates a file in Level_1, and the nodes are indexed by the minimum key of the corresponding file. With the nodes, we create a binary search tree. When we search the indexing tree, we try to find the node with the largest key among the nodes with keys smaller than required. With these nodes, we also check whether the required data reside in the data range of the node. If the data are in that range, we then find the exact file offset of the required data with the hash value.

Chapter 4

Evaluation

We implemented our data management mechanisms in Linux kernel 4.1.30 and Ubuntu 14.04. In following subsections, we describe the results of our optimized mechanisms with various benchmarks. The machine used in this study has 32 cores with 24 GB of main memory. Detailed information is described in Table 4.1. In following experiments, we resize the system memory size to simulated the data-intensive environments. We indicate the system memory size and data set size for each experiments.

Table 4.1 **Host machine Specification**

	Description
Cores	32 cores of Intel Xeon CPU E5-4620 2.60GHz
Main Memory	24GB
Storages	HDD 1TB & NVMe DC4800 (Optane)
Linux kernel	4.1.30
OS	Ubuntu 14.04

4.1 Memory-mapped I/O performance

We evaluated our *mmap* with page recycling by using multiple benchmarks that include a synthetic benchmark and HavoqGT [26]. Abbreviations for these systems are summarized in Table 4.2. 'Full DRAM' represents the full DRAM system, and the others represent the insufficient DRAM case. 'mmap' is the system with the original page reclamation procedure. To evaluate the impacts of our schemes, we built three systems that have different policies of page recycling.

Table 4.2 **Abbreviations for memory-mapped I/O results**

Abbreviation	Description
Full DRAM	The system has sufficient DRAM to keep all data in the main memory.
mmap	The system has insufficient DRAM, so the page reclamation procedure can occur. The process conducts the *original page reclamation procedure* in memory-mapped I/O.
C	The system has insufficient DRAM, and the process conducts the page reclaim procedure with *Clean-Only Recycling*.
F	The system has insufficient DRAM, and the process conducts the page reclaim procedure with *Full Recycling*.
FP	The system has insufficient DRAM, and the process conducts *Full Recycling*. In the Full Recycling path, recycled pages are kept in the *private list* of the core.

4.1.1 Synthetic benchmark results

We measured the performance of our *mmap* by using the synthetic benchmark used in subsection 2.3.1. It is an application that accesses a large file via *mmap* interface in a sequential, random pattern and repeats accessing the whole file three times iteratively. In this experiment, systems with the original *mmap* and *mmap* variants had 2GB main memory. Figure 4.1 shows the execution time of the synthetic application with each scheme. Our systems shows better performance than the system with the original *mmap* in every workload. The

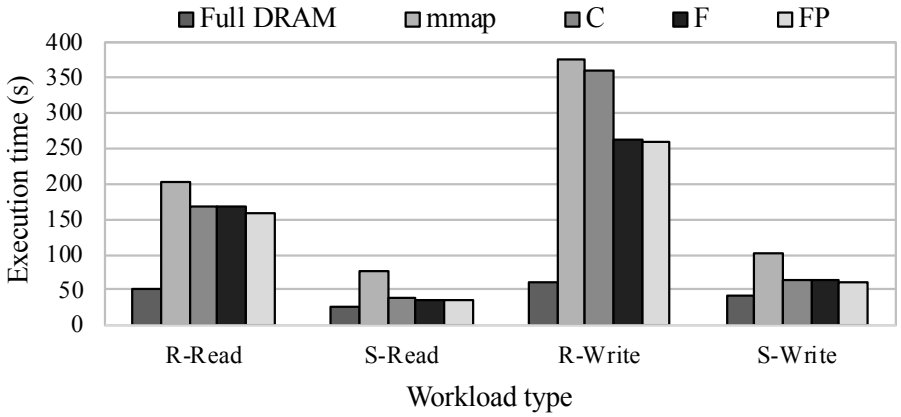


Figure 4.1 **Performance comparison with the synthetic benchmark.** The application access the data set whose size is 8GB with 2GB main memory.

"C" system shows enough performance improvement in read-intensive workloads, and the "F" system does likewise in write-intensive workloads. "FP" has similar performance with "F" because the application was single-threaded. So, recycled pages were not used by the other threads.

The batch size (the degree of recycling) can be an important parameter that determines performance improvement. The optimal batch size is dependent on workload characteristics, and our result indicates that recycling maximum 32 pages at once was the best policy for the synthetic benchmark. Recycling more than 64 pages at a time cannot provide seminal performance gain. We therefore use a batch size of 32 in the following experiments.

The salient purpose of page recycling is to reduce the possibility of context switches between the application and the *kswapd* process. The system with the original *mmap* includes considerable portions of sleep, which can be identified by using the Linux command "*time*". The output of the time command consists of *user*, *sys*, and *real* times. *User* and *sys* means the amount of CPU time in user-mode and kernel-mode, respectively. *Real* means all elapsed time to

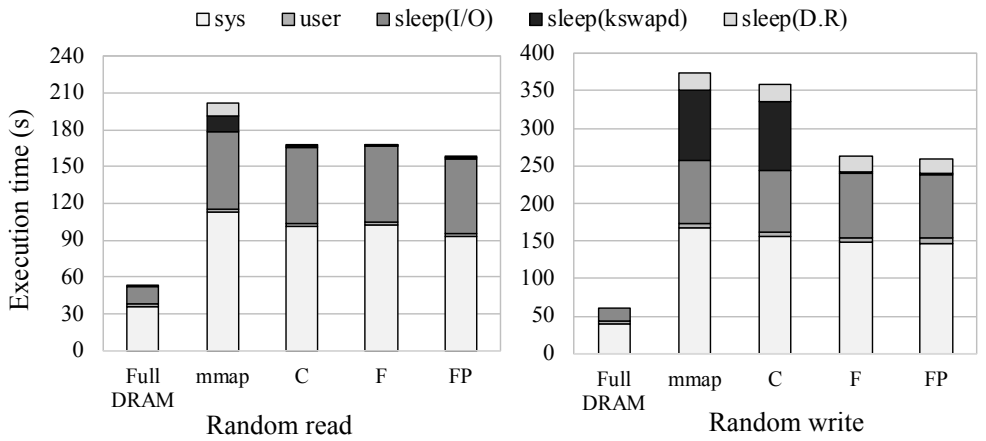


Figure 4.2 **Portion of sleep in total execution time.**

execute a program. Figure 4.2 shows the results of the *time* command with the synthetic benchmark in random workloads. In both (read/write) random workloads, when the synthetic application is executed with the full DRAM system, the summation of user and sys nearly equals to real.

On the contrary, when using the original *mmap*, representative as "mmap", user and sys summation is at most half of real. The empty part is the portion of sleeping. However, as each of our schemes is added, sleep time decreases. In the random read workload, we see near-zero sleep time in the system "C", indicating that recycling clean pages is enough to satisfy demands for free pages in the read-intensive workload. In the random write workload, the system with the "C" scheme still has some portion of sleep but the "F" policy reduces the sleep time significantly. From these results, we see that page recycling can substantially reduce the portion of sleep. This means that the advantages of *mmap* and fast storage devices can directly affect the performance of the application.

Reducing sleep time can be translated to better *predictability* as an important aspect of applications. Since the sleep time would vary according to

situations, execution time of the traditional *mmap* varies. The standard deviation in execution time of the original *mmap* is around 10%, which indicates poor predictability. However, our optimized *mmap* shows a lower standard deviation of about 6%. In this case, we can more accurately predict the response time of the application because *mmap* with page recycling always produces similar results.

4.1.2 Macro benchmark results

HavoqGT The Highly Asynchronous Visitor Queue Graph Toolkit (HavoqGT) is developed at Lawrence Livemore National Laboratory(LLNL); it implements a parallel-level asynchronous, breadth-first search traversal well suited to large scale-free graphs. It is a framework for expressing an asynchronous vertex-centric graph algorithm. HavoqGT makes the graph file and maps it through the *mmap* system call. We used HavoqGT version 0.1 at a scale of 24. The total graph size was 16GB, and the number of worker threads was 8. The performance of HavoqGT is represented by traversed edge per second (TEPS) and a higher graph bar means better performance. Figure 4.3 shows the execution time and TEPS performance according to each proposed scheme.

In the "Full DRAM" system, the main memory is larger than the graph file so that all data can reside in the main memory. Thus, the performance of "Full DRAM" is much greater than the others. However, insufficient DRAM systems with various *mmap* implementations induce performance degradation. To evaluate *mmap* performance with insufficient memory, we set the main memory to 4GB. The original *mmap* can traverse only 4.05E+06 edges per second which is 10% of Full DRAM performance. If our schemes are used, we can get considerable performance gains, especially for "FP", which shows 7x performance improvement over the original *mmap*. The performance of the "FP" system is

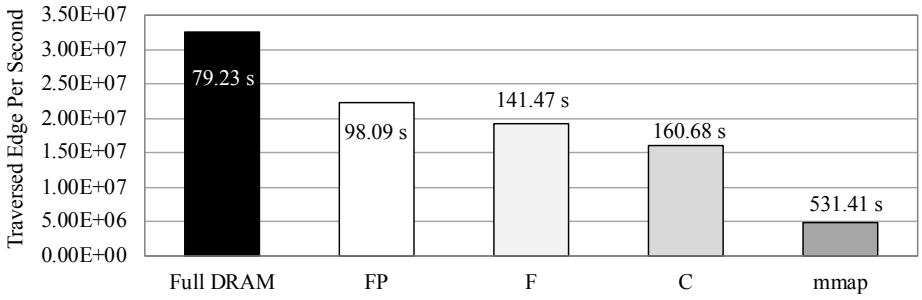


Figure 4.3 **HavoqGT performance.** Total graph size is 16GB, and the application runs 8 worker threads. Full DRAM means that system has 24GB main memory and others have 4GB main memory. The bar graph shows throughput and line graph shows execution time.

67% of Full DRAM. The line graph of Figure 4.3 shows the execution time of different schemes; the performance trends are very similar to that of TEPS.

More detailed analysis of the HavoqGT experiments is shown in Table 4.3. We count special events of Linux kernel such as page fault and entering the foreground/background reclaim by using the self-produced kernel module in each scheme. From the table, we see that more page faults occur in systems with our *mmap* than the system with the original *mmap*. This can be expected because unmapping of the process's address space is more demanded due to local reclaim. When our proposed schemes are used, the page fault handler rarely executes the background reclaim. This means that the page fault handler can recycle application's pages effectively. If it fails to get free pages in recycling path, the process wakes up the kswapd to reclaim pages in background mode (refer to Figure 3.2.) This is represented as "Background Reclaim" in the table. Its number also decreases and this turns into the reduced sleep portion in execution time. When the second trial to get free page ends up as failure, it enters the foreground reclaim which is represented as "Foreground Reclaim". As mentioned above, our important goal is keeping the process from entering

Table 4.3 **An analysis of HavoqGT performance**

Events	Full DRAM	FP	F	C	mmap
The number of Page fault	2,546,310	18,153,780	18,035,425	18,262,531	17,812,961
The number of Page Recycling	-	1,683,485	2,438,935	2,741,853	-
Background Reclaim	-	26	53	93	76,494
Foreground Reclaim	-	502	823	1,493	10,439,205

Note : This table is based on Figure 4.3.

foreground reclaim by satisfying the demand for free page in page recycling. In Table 4.3, the number of entering foreground reclaim decreases as the proposed schemes are developed, even zero in 'FP'. The reduced number of entering foreground reclaim directly connects to reduced execution time and increased TEPS.

4.2 Metadata operation performance

We conducted experiments with several benchmarks to measure performance of the optimized metadata management techniques. The storage device is HDD (Seagate 1TB) with Ext4 filesystem which is base filesystem of Ubuntu 14.04. Because our backward finding mechanism is implemented in VFS layer, the type of filesystem is not the big deal.

4.2.1 Microbenchmarks

In this section, we present the results with several microbenchmarks to explore advantages of the backward finding mechanism. With these microbenchmarks, we focused on specific latency of operations, or the number of dentry cache

lookups.

Synthetic benchmark

In Section 2.3.2, we observed that the latency of the `open()` system call increases in proportion to the depth of the file path. In this section, the `open()` system call is further broken down into parts related to and not-related to path traverse. The non-related parts include permission check, flag setting, and more. Figure 4.4 shows the breakdown result of the single latency when `open()` is applied with varying the file path depth in the warm cache environment.

In the case of the original Linux, the latency increases in the path traversing part, represented as *'Path walk'* in Figure 4.4, as the path depth increases. The non-related part, represented as *'ETC'.* in Figure 4.4, is constant regardless of the path depth. This indicates the path traversing overhead according to the path depth. The backward finding mechanism we proposed eliminates this overhead. As shown in the figure, when the backward mechanism is applied, it maintain almost constant path traverse time irrespective of the path depth. This is because we index dentry with the full path of the object and find the dentry with a single lookup.

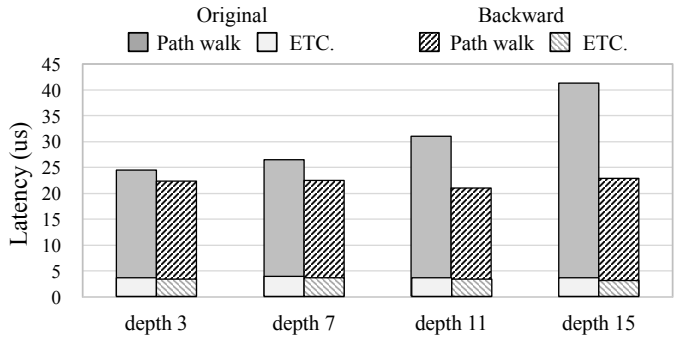


Figure 4.4 **Breakdown of the `open()` system call based on a warm cache environment.**

LMBenchmark

LMBench [50] is a suite of a simple, portable benchmark to measure the UNIX system performance. We used this benchmark to measure the single latency of the path-finding-related system calls while changing the path depth, such as performing the `stat()` and the `open()` system call, in a warm cache environment. The results are described in Figures 4.5, 4.6. In both case of the `stat()` and the `open()` system calls, the latency increases proportionally as the path depth increases in the original Linux kernel. When the path depth is 7, the `stat()` system call latency of the original Linux kernel has the 1.5x latency when compared to the path depth 1. However, when the backward finding mechanism is applied, latency is almost constant under all conditions. As a result, in `stat()` system call with depth 7, backward mechanism reduced latency by up to 60%.

Contrary to warm cache environment, in the cold cache environment, a single latency yields the same result between the original and backward finding. (The read-ahead effect is ignored for an accurate comparison of techniques.) Because there are no cached dentries, the backward finding mechanism must perform the same number of dentry cache lookups as the original. For brevity's sake, we do not include the results of the cold cache environment in this paper.

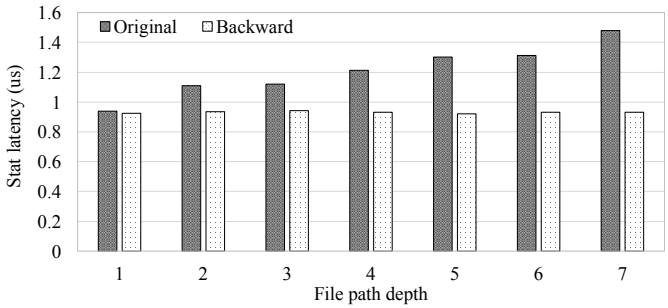


Figure 4.5 `stat()` **system call latency (warm cache).**

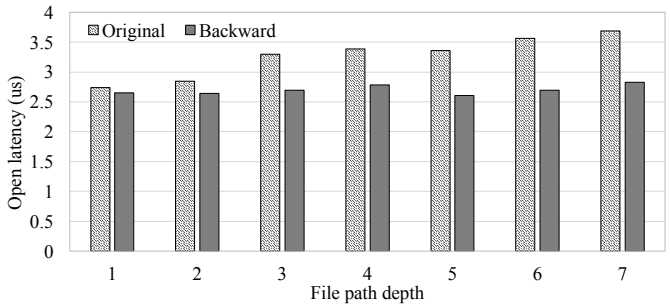
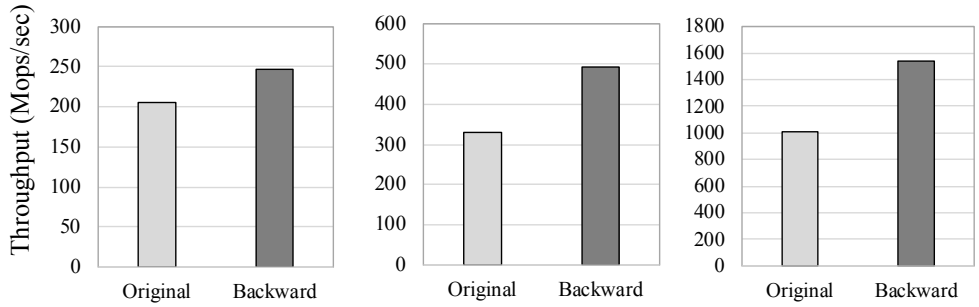


Figure 4.6 `open()` **system call latency (warm cache).**

4.2.2 Real-world workload

We experimented with real-world workload to demonstrate how our proposed techniques increase the performance of the system in this section. We obtained the trace from Kyujanggak website [51] about the accessed files for two weeks. Kyujanggak was the royal library of the Joseon Dynasty. Today, Kyujanggak is controlled by Seoul National University Kyujanggak Institute of Korean Studies [51]. The collection includes seven different types of texts, numbering 7,125 documents in total. It also includes national treasures, such as the Annals of the Joseon Dynasty. Because the Kyujanggak is a type of museum, its data has the property of BLOBs (written once, read often, never update, and rarely deleted) and the trace is structured as read-only pattern. We decide the Kyujanggak system trace to experiment because their properties would be suitable for our proposed mechanisms. The average path depth of accessed files is about $8 \sim 10$ and the accessed file size is mostly under 100KB. The average file size is 64KB and the total dataset is 6.8GB. The number of trace lines is about 3,000,000 and the total number of files is 100,000. We measure the throughput during the replaying of the trace.

Figure 4.7 shows a comparison of throughput on different storage environ-



(a) Cold - HDD (b) Cold - SSD (c) Warm

Figure 4.7 **Throughput results** (real-world workload).

ments and cache environment. When we apply the backward path finding mechanism, the throughput improves by up to 1.5x in cold cache environment with SSD. We analyze the experimental results and trace property to breakdown the main point of performance improvement. Based on our analysis, we found that this Kyujanggak trace shows the high spatial locality. That means the files in the same directories are accessed repeatedly. When spatial locality is high, the backward mechanism can have big advantages. Therefore, this trace can take advantage of the backward finding mechanism.

Table 4.4 shows the number of dentry cache lookups during the replaying of the Kyujanggak trace. This measurement is conducted on cold cache environment. When dentry cache lookup is failed in memory, slow lookup starts and finds inodes through storage I/O. Therefore, the number of slow lookup is same as the number of failure of dentry cache lookup. In Table 4.4, 'Slow lookup' means looking up through storage I/O when dentry cache lookup is failed. The dentry cache lookup count is reduced by about 10x when the backward mechanism is adopted. The number of slow lookup is rarely changed. That is, the path traversing is completed with only small number of dentry cache lookups. This reduced dentry cache lookup is the main contribution of reduced execution

time when backward finding mechanism.

Table 4.4 **Comparison of the number of dentry cache lookups.**

	Original	**Backward**
Dentry cache lookup	2,122,644	294,961
Slow lookup	29,685	29,881

4.3 RLSM performance

As mentioned above, we implemented RLSM with HBase. HBase usually uses the Hadoop Distributed File System (HDFS) as an underlying storage system. In this study, we execute HBase in the standalone mode to exclude the effect of network settings among HDFS data nodes. In the following experiments, we use openJDK1.8.0 and the JVM heap size is 4GB.

HBase divides the heap into three spaces: the first for the write cache (memstore), the second for the read block cache, and the third for system usage such as metadata and indexing data. The ratio of division is 4(memstore) : 4(read block cache) : 2(system usage); however, this ratio is flexible and can be adjusted according to workload patterns. We use the only 100MB write buffer of the 1.6GB memstore, and the rest of memstore is used for the snapshot of the write buffer. When the 100MB write buffer is full of key-value pairs, the RLSM snapshots the write buffer, and write the key-value pairs down to a file in a log-structured manner. Therefore, the basic file size is about 100MB in Level_0. The threshold size which triggers the partitioning is 120MB. As a result, the average size of files in Level_1 is about 60MB.

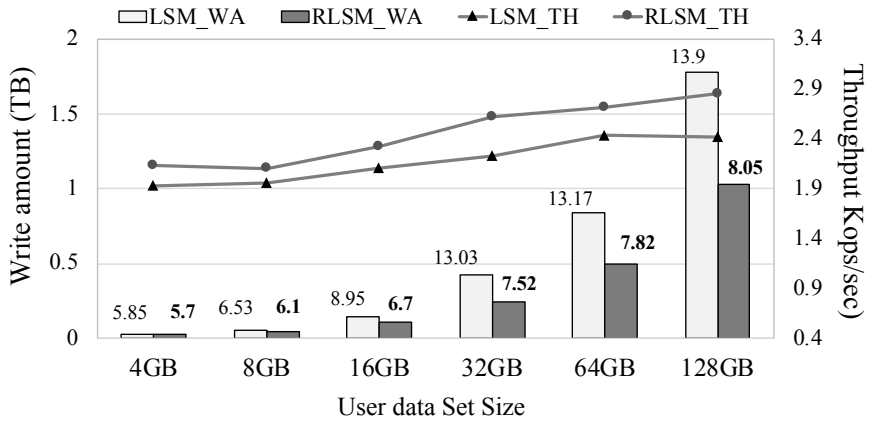


Figure 4.8 **Comparison of write performance in HBase.** YCSB experiments with insert-only workload. In the legend, WA means total write amount and TH means throughput. The value at the top of the bar is the Write Amplification Ratio.

4.3.1 Write performance

First, we measure the improvement of write performance, including write amplification. We can measure the amount of write to the storage device by using the Linux `vmstat` command during the execution of HBase. We perform the YCSB experiments with the insert-only workload and measure the write performance with varying the size of the user data set. The results are shown in Figure 4.8. In original HBase, the write amplification ratio increases as the user data set increases, and finally reaches about 13 with 128GB user data set. However, RLSM-based HBase can reduce the write amplification significantly. Even with the 128GB data set, the write amplification ratio is 5.29. And with the 64GB data set, we can reduce write amplification by a factor of about 3. We can see that RLSM-based HBase achieves 33% better write throughput than original HBase in a 64GB user data set.

Reducing the I/O caused by the compaction process also can reduce the

execution time of compaction. We measure the compaction time in the previous YCSB experiment with data set size of 64GB. Figure 4.9 shows the execution time of compaction during the experiment. In original HBase, the compaction time increases over time. This means that the burden for compaction increases because data to be sorted increases and the sorted data should be rewritten to the storage over time. However, our RLSM shows relatively constant compaction time over time, because compaction in RLSM simply appends the data at Level0 to the target files at Level_1. One variable parameter in compaction time may be load balancing (e.g., partitioning). However, it does not significantly affect the compaction time because the load balancing does not occur often. Even though the load balancing occurs, the number of target files whose size is relatively large or small would not be large.

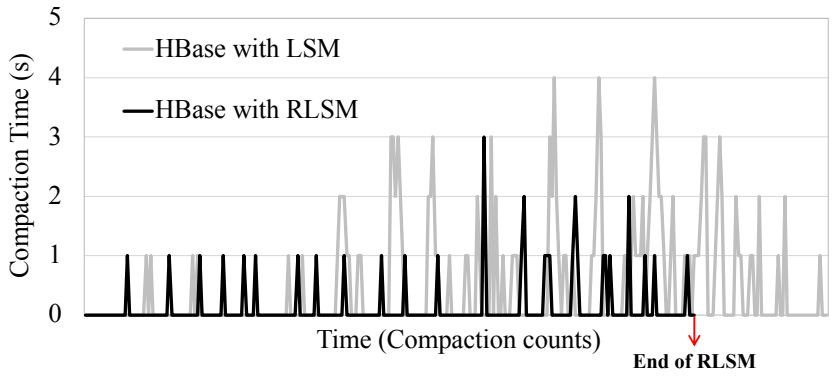


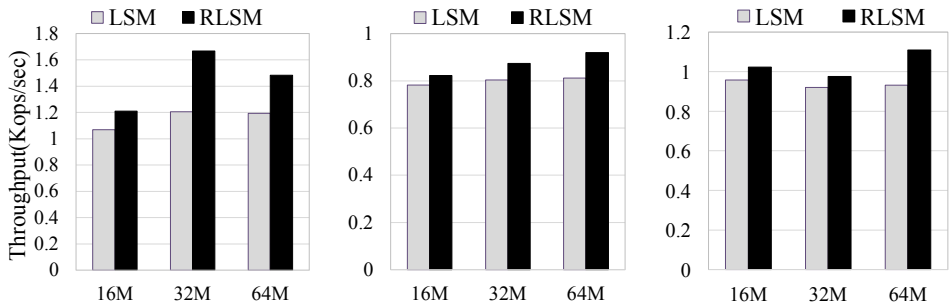
Figure 4.9 **Comparison of compaction time.** YCSB experiment with the insert-only workload and 64 GB data set.

4.3.2 Performance under the mixed workload

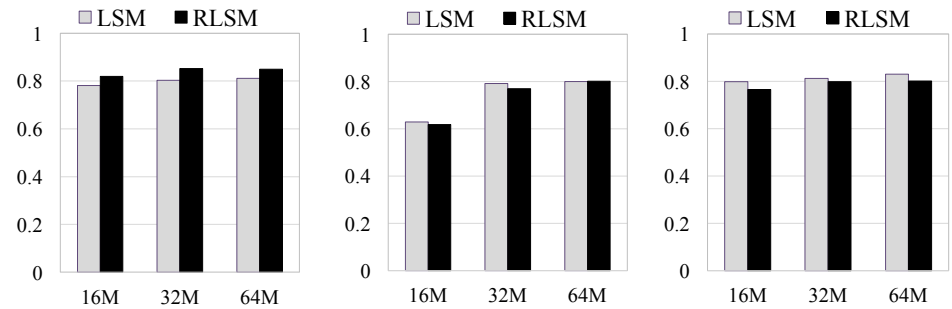
Figure 4.10 shows the experimental results with the mixed workload. For the evaluation, the base data set size is 64GB, and we performance experiment with varying workload type. When the ratio of Read and Update is 5:5, the

performance gain is the largest. When the count of operations is 32M, the throughput reaches at 1700ops/sec. As seen in Figure 4.10 (b) and Figure 4.10 (d), the performance improvement is about 8% when we use read-intensive workloads. With the read-only workload, the throughput is increased by up-to 10% since RLSM uses hash-based metadata.

In our RLSM, file sizes are adjusted to be relatively small by the load balancing process, in response to the read operations, especially scan operations. The measured scan performances are shown in Figure 4.10 (e) and 4.10 (f). In the figures, S_S indicates that the scan range is smaller than the file size, and S_L indicates the scan range is larger than the file size. In Figure 4.10 (e), RLSM-based HBase achieves 95% performance of the original HBase. However, the overhead is slightly increased when the scan range increases. We can see the decreased scan performance of about 10% in Figure 4.10 (f). The performance degradation arises since RLSM loads the files with corresponding ranges and sort the loaded data in memory to response to the scan operation. Note that other LSM variants based on hash-based data structures (LSM-trie [52]) or un-sorted data (Wisckey [53]) do not cover the scan query.



(a) R : U = 0.5 : 0.5 (b) R : U = 0.95 : 0.05 (c) R = 1



(d) R : I = 0.95 : 0.05 (e) S_S : I = 0.95 : 0.05 (f) S_L : I = 0.95 : 0.05

Figure 4.10 **Compaction of throughput in various workloads.** Base data set size is 64GB. X-axis means the count of operations. 'R' = 'Read', 'U' = 'Update' and 'I' = 'Insert'. In (e) and (f), S_S means that the scan range is smaller than the file size and S_L means that the scan range is larger than the file size.

Chapter 5

Related Work

5.1 Effort to adopt the changes

To handle this massive amount of data, many research studies [2, 17, 18, 54] have been conducted on data managing frameworks or programming models that can process such an enormous amounts of data. For example, Facebook has developed F4 [19] and haystack [20] as the data managing frameworks that efficiently manages large data.

Apache project developed the Hadoop framework [55], which is open-source software for reliable, scalable, distributed computing. It allows processing of large data sets across cluster of computers using simple programming models. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failure. Hadoop systems includes these modules: Hadoop Common, MapReduce [1], HDFS [2], YARN [56]. Other hadoop-related projects at apache

include Cassandra [57], Hive [58], Pig [59], Spark [60], Zookeeper [61], etc,. These systems comprise of Hadoop echo system.

MapReduce [1] is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a `Map`() procedure that performs filtering and sorting and a `Reduce`() procedure that performs a summary operation. The MapReduce system orchestrates the processing by marshallings the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the systems, and providing for redundancy and fault tolerance.

CUDA [62] is a parallel computing platform and application programming model created by NVIDIA [63]. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing (an approach termed GPGPU (General Purpose computing on GPU)). The CUDA platform is designed to work with programming languages such as C, C++, and Fortran. This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like Direct3D and OpenGL, which required advanced skills in graphics programming. Also, CUDA supports programming frameworks such as OpenACC and OpenCL. When it was first introduced by Nvidia, the name CUDA was an acronym for Compute Unified Device Architecture, but Nvidia subsequently dropped the use of the acronym.

Many systems have been introduced to handle growing unstructured data, such as multimedia data. NoSQL [64] databases are emerged to process these data type. They are next-generation database mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable. The original intention has been modern web-scale databases. The movement

began early 2009 and is growing rapidly. Often more characteristics apply such as: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), a huge amount of data and more. So the misleading term "nosql" (the community now translates it mostly with "not only sql") should be seen as an alias to something like the definition above.

MongoDB [65] is a representative NoSQL database. It is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas. MongoDB is developed by MongoDB Inc. and is free and open-source, published under a combination of the GNU Affero General Public License and the Apache License. MongoDB scales horizontally using sharding. The user chooses a shard key, which determines how the data in a collection will be distributed. The data is split into ranges (based on the shard key) and distributed across multiple shards. (A shard is a master with one or more slaves.). Alternatively, the shard key can be hashed to map to a shard – enabling an even data distribution. MongoDB can run over multiple servers, balancing the load or duplicating data to keep the system up and running in case of hardware failure. MongoDB can be used as a file system with load balancing and data replication features over multiple machines for storing files. This function, called Grid File System [66] is included with MongoDB drivers. MongoDB exposes functions for file manipulation and content to developers.

Since a large amount of data can not be processed in one local server, many clustered structures computing systems [17, 21, 22, 23, 24, 25, 67] are used to process data in parallel or in distributed.

IndexFS [23] adds support to existing file systems such as PVFS [68], Lustre [17], and HDFS [2] for scalable high-performance operations on metadata and small files. That is, it is middleware inserted into existing deployments of

cluster file systems to improve metadata efficiency while maintaining high I/O bandwidth for data transfers. IndexFS uses a table-based architecture that incrementally partitions the namespace on a per-directory basis, preserving server and disk locality for small directories. An optimized log-structured layout is used to store metadata and small files efficiently.

Ceph [54], a free-software storage platform, implements object storage on a single distributed computer cluster, and provides interfaces for object-, block- and file-level storage. Ceph aims primarily for completely distributed operation without a single point of failure, scalable to the exabyte level, and freely available. Ceph replicates data and makes it fault-tolerant, using commodity hardware and requiring no specific hardware support. As a result of its design, the system is both self-healing and self-managing, aiming to minimize administration time and other costs. Ceph does striping of individual files across multiple nodes to achieve higher throughput, similarly to how RAID-0 stripes partitions across multiple hard drives. Adaptive load balancing is supported whereby frequently accessed objects are replicated over more nodes. As of December 2014, XFS [5] is the recommended underlying filesystem type for production environments, while Btrfs [69] is recommended for non-production environments. ext4 filesystems are not recommended because of resulting limitations on the maximum RADOS objects length.

Lustre [17] is similar to ceph, but it is a type of parallel distributed file systems, not object storage. Lustre is generally used for large-scale cluster computing. Lustre file system software provides high performance file systems for computer clusters ranging in size from small workgroup clusters to large-scale, multi-site clusters. The Lustre file system also uses inodes, but inodes on MDTs point to one or more OST objects associated with the file rather than to data blocks. These objects are implemented as files on the OSTs. When a client opens

a file, the file open operation transfers a set of object identifiers and their layout from the MDS to the client, so that the client can directly interact with the OSS node where the object is stored. This allows the client to perform I/O in parallel across all of the OST objects in the file without further communication with the MDS.

In the G-HBA [21], the authors presents a scalable and adaptive decentralized metadata lookup scheme for ultra large-scale file systems ($\geq$ Petabytes or even Exabytes). Their scheme logically organizes metadata servers (MDS) into a multi-layered query hierarchy and exploits grouped Bloom filters to efficiently route metadata requests to desired MDS through the hierarchy. This metadata lookup scheme can be executed at the network or memory speed, without being bounded by the performance of slow disks.

The local system in distributed data processing frameworks, especially the local filesystem, is responsible for managing data as a 'file' and managing storage I/O. Modern file systems deliver scalable performance for large files, but not for large numbers of files [70, 71] Therefore, researchers try to optimize the local filesystems with a number of files [4, 34]. Recently, as the number of files in the local FS has increased considerably, interest in metadata and data content has increased as well; in particular, the overhead of metadata management has spiked [33, 35].

In the GIGA+ filesystem [4], the authors examine the problem of scalable file system directories, motivated by data-intensive applications requiring millions to billions of small files to be ingested in a single directory at rates of hundreds of thousands of file creates every second. We introduce a POSIX-compliant scalable directory design, GIGA+, that distributes directory entries over a cluster of server nodes. For scalability, each server makes only local, independent decisions about migration for load balancing.

File systems that manage magnetic disks have long recognized the importance of sequential allocation and large transfer sizes for file data. Fast random access has dominated metadata lookup data structures with increasing use of B-trees on-disk. Yet our experiments with workloads dominated by metadata and small file access indicate that even sophisticated local disk file systems like Ext4, XFS and Btrfs leave a lot of opportunity for performance improvement in workloads dominated by metadata and small files. Therefore, tableFS [35] uses another local file system as an object store. It organizes all metadata into a single sparse table backed on disk using a Log-Structured Merge (LSM) tree [36], LevelDB.

CFFS [33] also points out the metadata overhead. Traditional file system optimizations typically use a one-to-one mapping of logical files to their physical metadata representations. Such mapping is desirable because metadata constructs are deep-rooted data structures, and many storage components and mechanisms—such as VFS API, prefetching, and metadata caching—rely on such constructs. However, this mapping results in missed opportunities for a class of optimizations in which such coupling is removed. CFFS allows many-to-one mappings of files to metadata.

In large scale systems, applications frequently request file system operations that traverse the file system directory tree, such as opening a file or reading a file's metadata. As a result, caching file system directory structure and metadata in memory is an important performance optimization for an OS kernel. The paper [48] identifies several design principles that can substantially improve hit rate and reduce hit cost transparently to applications and file systems. Specifically, their directory cache design can look up a directory in a constant number of hash table operations, separates finding paths from permission checking, memoizes the results of access control checks, uses signatures to accelerate

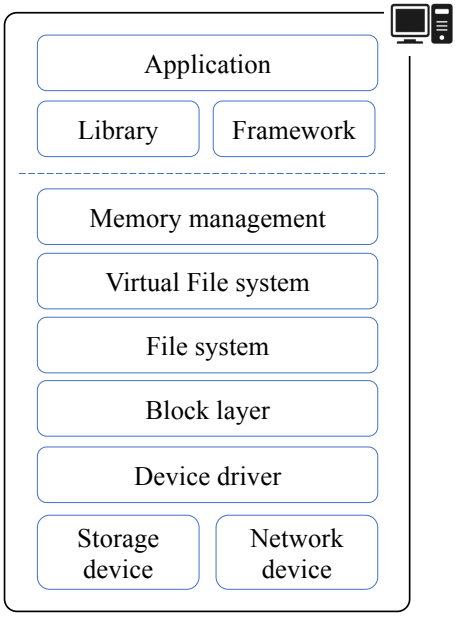


Figure 5.1 **I/O layers in local systems.**

lookup, and reduces miss rates through caching directory completeness.

In computing systems, the IO path from application to storage is long and complex. The I/O operations have to through the multiple layers, such as VFS, file systems, block layer, etc,. Figure 5.1 shows the system layers that I/O operation should go through. In this architecture, storage systems means file systems, block layer and device driver.

Emerging non-volatile memory technologies such as phase change memory (PCM) promise to increase storage system performance by a wide margin relative to both conventional disks and flash-based SSDs. Realizing this potential will require significant changes to the way systems interact with storage devices as well as a rethinking of the storage devices themselves. Many researches [30, 31, 72, 73, 74, 75, 76] optimized the storage system to fast storages.

Moneta [72] describes the architecture of a prototype PCIe-attached stor-

age array built from emulated PCM storage. Moneta provides a carefully designed hardware/software interface that makes issuing and completing accesses atomic. The atomic management interface, combined with hardware scheduling optimizations, and an optimized storage stack increases performance for small, random accesses. The authors have developed the system by delegating kernel's task to hardware. This system is called Moneta-D [30].

In the papers [31, 73] for optimizing block layer for fast storage, the authors propose six optimizations that enables an OS to fully exploit the performance characteristics of fast storage devices. With the support of new hardware interfaces, their optimizations minimize per-request latency by streamlining the I/O path and amortize per-request latency by maximizing parallelism inside the device. Also, they find that eliminating context switches in the I/O path decreases the software overhead of an I/O request from 20 us to 5 us and a new request merge scheme called Temporal Merge enables the OS to achieve 87% to 100% of peak device performance, regardless of request access patterns or types.

As mentioned in the paper [31], the context switch can have harmful effect on I/O latency. With existing storage device which has very high latency such as HDD, this context switch overhead can be hided. However, with fast storage, this overhead increase the I/O latency. Therefore, the authors in the paper [77] said that polling scheme is better than interrupt when using fast storage devices. This paper thus argues for the synchronous completion of block I/O first by presenting strong empirical evidence showing a stack latency advantage, second by delineating limits with the current interrupt-driven path, and third by proving that synchronous completion is indeed safe and correct.

We now focus on non-volatile memory which can be attached directly to the memory bus and is also byte addressable. Such nonvolatile memory can be used

in the computer system for the main memory as well as for persistent storage of files. Thus, it reduces the latencies to access persistent storage. There are many related works [78, 79, 80, 81, 82] to use SCM or NVM more efficient in memory management layer and VFS layer.

In SCMFS [81], the authors propose a new file system - SCMFS, which is specifically designed for SCM. With consideration of compatibility, this file system exports the identical interfaces as the regular file systems do, in order that all the existing applications can work on it. In this file system, they aim to minimize the CPU overhead of file system operations. The authors build SCMFS on virtual memory space and utilize the memory management unit (MMU) to map the file system address to physical addresses on SCM. The layouts in both physical and virtual address spaces are very simple. They also keep the space contiguous for each file in SCMFS to simplify the process of handling read/write requests in the file system.

SSDAlloc [82] is a hybrid main memory management system that allows developers to treat solid-state disk (SSD) as an extension of the RAM in a system. SSDAlloc moves the SSD upward in the memory hierarchy, usable as a larger, slower form of RAM instead of just a cache for the hard drive. Using SSDAlloc, applications can nearly transparently extend their memory footprints to hundreds of gigabytes and beyond without restructuring, well beyond the RAM capacities of most servers.

In the paper [83], the authors propose, crafted from a fundamental understanding of PCM technology parameters, area-neutral architectural enhancements that address these limitations and make PCM competitive with DRAM. To exploit PCM's scalability as a DRAM alternative, PCM must be architected to address relatively long latencies, high energy writes, and finite endurance. A baseline PCM system is 1.6x slower and requires 2.2x more energy than a

DRAM system. Buffer reorganizations reduce this delay and energy gap to 1.2x and 1.0x, using narrow rows to mitigate write energy and multiple rows to improve locality and write coalescing. Partial writes enhance memory endurance, providing 5.6 years of lifetime. Process scaling will further reduce PCM energy costs and improve endurance.

To reflect the characteristic of new fast storages, many kind of file systems [10, 11] are emerged.

F2FS [10] is a Linux file system designed to perform well on modern flash storage devices. The file system builds on append-only logging and its key design decisions were made with the characteristics of flash storage in mind. This paper describes the main design ideas, data structures, algorithms and the resulting performance of F2FS. It has five contributions.: Flash-friendly on-disk layout, Cost-effective index structure, Multi-head logging, Adaptive logging, fsync acceleration with roll-forward recovery. F2FS builds on the concept of LFS [84] but deviates significantly from the original LFS proposal with new design considerations

Chapter 6

Conclusion

In this dissertation, we research the impact of society changes in computing systems and the tries to adopt the changes. We especially focus on the memory management layer and VFS layer. Based on these observations, we suggest the optimized schemes of data management on high-performance storage devices.

First of all, we propose the page recycling which is efficient page reclamation scheme. In page recycling, we reduce the page un-mapping overhead by recycling only processes' own pages. Page recycling can reduce the unnecessary context switches, then reduce the overhead of page reclamation. This scheme is very useful when memory-mapped I/O is used with high-performance storage devices.

Also, as modern computer systems face the challenge of large data, filesystems must deal with a large number of files. This leads to amplified concerns of metadata and data operations. In large-scale filesystems, the path-based file access can suffer from the redundant metadata lookups. Moreover, the high-performance storage devices make the data-access latency low, so the metadata

lookup overhead can be more visible. To reduce the redundant hash table lookup to find the metadata, we propose the backward finding mechanism. With backward finding scheme, we can find the target dentry with reduced number of lookups.

Finally, we optimized the LSM tree because it contains complicated in-memory data structures which are used well on traditional slow storage devices. Therefore, we suggest the Ranged LSM algorithm which conduct the compaction process with only appending operation by using more simplified data structures. RLSM can reduce the side effect of the original LSM such as read/write amplification and takes advantage of the high-performance storage devices.

With these optimizations, we evaluated the system performance and prove the improved performance. Therefore, we can optimize the data management mechanism on the fast storage devices.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pp. 1–10, IEEE, 2010.

[3] "Google inc.." http://www.google.com/.

[4] S. Patil and G. A. Gibson, "Scale and concurrency of giga+: File system directories with millions of files.," in *FAST*, vol. 11, pp. 13–13, 2011.

[5] R. Y. Wang and T. E. Anderson, "xfs: A wide area mass storage file system," in *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pp. 71–78, IEEE, 1993.

[6] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, M. Bender, *et al.*, "Optimizing every operation in a write-optimized file system," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 1–14, 2016.

[7] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, *et al.*, "Betrfs: Write-optimization in a kernel file system," *ACM Transactions on Storage (TOS)*, vol. 11, no. 4, p. 18, 2015.

[8] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "The tokufs streaming file system.," in *HotStorage*, 2012.

[9] A. Al Mamun, G. Guo, and C. Bi, *Hard disk drive: mechatronics and control*. CRC press, 2017.

[10] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2fs: A new file system for flash storage.," in *FAST*, pp. 273–286, 2015.

[11] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "Xtreemfs: a case for object-based storage in grid data management," in *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB*, vol. 2007, 2007.

[12] "Facebook." http://www.facebook.com/.

[13] "Twitter." http://www.twitter.com/.

[14] J. Shi, W.-J. Lee, Y. Liu, Y. Yang, and P. Wang, "Forecasting power output of photovoltaic systems based on weather classification and support vector machines," *IEEE Transactions on Industry Applications*, vol. 48, no. 3, pp. 1064–1069, 2012.

[15] D. Malhotra, A. Gholami, and G. Biros, "A volume integral equation stokes solver for problems with variable coefficients," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 92–102, IEEE Press, 2014.

[16] K. Mahadik, S. Chaterji, B. Zhou, M. Kulkarni, and S. Bagchi, "Orion: Scaling genomic sequence matching with fine-grained parallelization," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pp. 449–460, IEEE, 2014.

[17] "Lustre file system." http://lustre.org/.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, pp. 29–43, ACM, 2003.

[19] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, *et al.*, "f4: Facebook's warm blob storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 383–398, 2014.

[20] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, *et al.*, "Finding a needle in haystack: Facebook's photo storage.," in *OSDI*, vol. 10, pp. 1–8, 2010.

[21] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and adaptive metadata management in ultra large-scale file systems," in *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, pp. 403–410, IEEE, 2008.

[22] P. H. Lensing, T. Cortes, J. Hughes, and A. Brinkmann, "File system scalability with highly decentralized metadata on independent storage devices," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pp. 366–375, IEEE, 2016.

[23] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," in *High*

Performance Computing, Networking, Storage and Analysis, SC14: International Conference for, pp. 237–248, IEEE, 2014.

[24] L. Pineda-Morales, A. Costan, and G. Antoniu, "Towards multi-site metadata management for geographically distributed cloud workflows," in *2015 IEEE International Conference on Cluster Computing*, pp. 294–303, Sept 2015.

[25] D. Dai, Y. Chen, P. Carns, J. Jenkins, W. Zhang, and R. Ross, "Graphmeta: A graph-based engine for managing large-scale hpc rich metadata," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 298–307, Sept 2016.

[26] B. Van Essen, H. Hsieh, S. Ames, and M. Gokhale, "Di-mmap: A high performance memory-map runtime for data-intensive applications," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pp. 731–735, Nov 2012.

[27] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[28] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of nvram in data-intensive architectures: An evaluation," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 703–714, May 2012.

[29] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From aries to mars: Transaction support for next-generation, solid-state drives," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 197–212, ACM, 2013.

[30] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pp. 387–400, ACM, 2012.

[31] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, "Optimizing the block i/o subsystem for fast storage devices," *ACM Trans. Comput. Syst.*, vol. 32, pp. 6:1–6:48, June 2014.

[32] U. Vahalia, *UNIX internals: the new frontiers*. Pearson Education India, 1996.

[33] S. Zhang, H. Catanese, and A. A.-I. Wang, "The composite-file file system: decoupling the one-to-one mapping of files and metadata for better performance," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 15–22, 2016.

[34] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 4, IEEE Computer Society, 2004.

[35] K. Ren and G. A. Gibson, "Tablefs: Enhancing metadata efficiency in the local file system.," in *USENIX Annual Technical Conference*, pp. 145–156, 2013.

[36] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[37] S. Ghemawat and J. Dean, "Leveldb," 2011.

[38] Facebook, "Rocksdb," may 2012.

[39] A. Cassandra, "Apache cassandra," Nov. 2010.

[40] "Hyperleveldb," Nov. 2012.

[41] "Apache hbase." https://hbase.apache.org.

[42] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[43] Z. Fong and R. Shroff, "Hydrabase – the evolution of hbase@facebook," June 2014.

[44] "Facebook's new real-time messaging system: Hbase to store 135+ billion messages a month," Nov. 2010.

[45] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc.", 2013.

[46] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: A replacement algorithm for flash memory," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, (New York, NY, USA), pp. 234–241, ACM, 2006.

[47] P. H. Lensing, T. Cortes, and A. Brinkmann, "Direct lookup and hash-based metadata placement for local file systems," in *Proceedings of the 6th International Systems and Storage Conference*, p. 5, ACM, 2013.

[48] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter, "How to get more value from your file system directory cache," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 441–456, ACM, 2015.

[49] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Journal of computer and system sciences*, vol. 7, no. 4, pp. 448–461, 1973.

[50] L. W. McVoy, C. Staelin, *et al.*, "lmbench: Portable tools for performance analysis.," in *USENIX annual technical conference*, pp. 279–294, San Diego, CA, USA, 1996.

[51] "Kyujanggak institute for korean studies." http://kyujanggak.snu.ac.kr/LANG/en/main/main.jsp.

[52] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: an lsm-tree-based ultra-large key-value store for small data," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pp. 71–82, USENIX Association, 2015.

[53] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, p. 5, 2017.

[54] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the*

7th symposium on Operating systems design and implementation, pp. 307–320, USENIX Association, 2006.

[55] "Apache hadoop." http://hadoop.apache.org/.

[56] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

[57] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[58] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[59] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1099–1110, ACM, 2008.

[60] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[61] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX annual technical conference*, vol. 8, p. 9, 2010.

[62] C. Nvidia, "Programming guide," 2010.

[63] "Nvidia." http://www.nvidia.com/.

[64] "No-sql database." http://nosql-database.org/.

[65] "Mongodb." http://www.mongodb.com/Atlas/.

[66] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 1, pp. 38–71, 1984.

[67] J. Byers, J. Considine ˆ, and M. Mitzenmacher, "Simple load balancing for distributed hash tables," *Peer-to-peer systems II*, pp. 80–87, 2003.

[68] I. F. Haddad, "Pvfs: A parallel virtual file system for linux clusters," *Linux Journal*, vol. 2000, no. 80es, p. 5, 2000.

[69] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[70] R. Wheeler, "One billion files: Scalability limits in linux file systems," *Presentation at LinuxCon*, vol. 10, 2010.

[71] A. Fikes, "Storage architecture and challenges," *Talk at the Google Faculty Summit*, 2010.

[72] A. M. Caulfield and J. Coburn, "Moneta : A High-performance Storage Array Architecture for Next-generation , Non-volatile Memories," *Micro*, 2010.

[73] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, H. Eom, and H. Y. Yeom, "Exploiting peak device throughput from random access workload," in

Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems, HotStorage'12, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2012.

[74] Y. Son, H. Han, and H. Y. Yeom, "Optimizing file systems for fast storage devices," in *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, 2015.

[75] Y. Son, N. Song, H. Han, H. Eom, and H. Yeom, "Design and evaluation of a user-level file system for fast storage devices," *Cluster Computing*, vol. 18, no. 3, pp. 1075–1086, 2015.

[76] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: a software-defined storage architecture," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 182–196, ACM, 2013.

[77] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt.," in *FAST*, vol. 12, pp. 3–3, 2012.

[78] J. Coburn, A. M. Caulfield, L. M. Grupp, R. K. Gupta, and S. Swanson, "NV-Heaps : Making Persistent Objects Fast and Safe with Next-Generation , Non-Volatile Memories," *ASPLOS*, pp. 105–117, 2011.

[79] M. Wu and W. Zwaenepoel, "envy: A non-volatile, main memory storage system," in *ACM SigPlan Notices*, vol. 29, pp. 86–97, ACM, 1994.

[80] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Io-lite: a unified i/o buffering and caching system," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 1, pp. 37–66, 2000.

[81] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 39:1–39:11, ACM, 2011.

[82] A. Badam and V. S. Pai, "Ssdalloc: hybrid ssd/ram memory management made easy," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2011.

[83] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 2–13, ACM, 2009.

[84] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

요약

최근 하드웨어 기술의 발전으로 저장 장치가 발전함에 따라 Solid State Drive (SSD) 와 같은 고성능 저장 장치가 등장하였다. 고성능 저장 장치들은 높은 대역폭, 낮은 지연시간, 높은 입출력 및 병렬성을 제공하며, 기존 Hard Disk Drive (HDD) 의 기계적 오버헤드를 없앴기 때문에 데이터 접근을 수 십에서 수 백 배 빠르게 한다. 하지만, 이러한 고성능 저장 장치들을 기존의 소프트웨어 계층에서 그대로 사용하게 된다면 소프트웨어 계층의 오버헤드 때문에 고성능 저장 장치의 성능을 최대로 사용할 수없다.

본 논문에서는 고성능 저장 장치의 특성에 맞게 데이터 관리 기법들을 최적화 한다. 고성능 저장 장치는 지연시간이 낮기 때문에 기존 소프트웨어 계층에서의 오버헤드가 더 많이 드러난다. 본 논문에서 지적한 첫 번째 소프트웨어 오버헤드는 페이지 회수 오버헤드이다. 고성능 저장 장치 기반의 시스템에서 매핑된 페이지를 회수 할 때 소프트웨어 계층의 unmap 오버헤드가 부각된다. 이를 줄이기 위해서 본 논문에서는 page recycling 기법을 제안하여 unmap overhead를 해당 응용 프로그램으로 국한시킴으로써 전체 시스템의 성능을 높일 수 있었다.

두 번째는 metadata lookup operation이다. 기존 리눅스 시스템에서는 파일들을 path 기반으로 관리하고 있다. 이러한 path 기반 파일들을 접근 하기 전에 반드시 수행되어야 하는 metadata operation은 hash table lookup이 중복됨으로써 파일 접근 시에 오버헤드를 유발한다. 이러한 오버헤드는 고성능 저장 장치에서 더 크게부각 되는데 상대적으로 데이터 접근의 오버헤드가 적어지기 때문이다. 따라서 효율적인 metadata lookup operation을 위해 본 논문에서는 hash table 을 접근할 때 검색 방향을 거꾸로 하는 backward finding 을 제안한다. 이와 같은 방법으로 metadata lookup operation의 횟수를 줄이고 전체 파일 접근 시간을 줄일 수 있었다.

기존의 Log-Structured Merge (LSM) 알고리즘은 기존 저장 장치의 지연 시간이 길다는 것을 고려하여서 알고리즘 자체를 복잡한 data structure를 사용 하여 구현하였다. 하지만 이러한 복잡한 data structure 때문에 오히려 부작용으로써 read/write amplification 이늘어난다. 고성능 저장 장치를 활용한다면 굳이 복잡한 data structure 를 사용하지 않아도 data 의 접근을 빠르게 할 수있다. 따라서 본 논문에서는 간단한 data structure 를 사용하여서 기존LSM 알고리즘을 수정하고 또한 write amplification 을 유발하는 compaction 과정도 data의 범위에 맞춰서 파일에 append만 사용하여서 효율적으로바꾸었다. 이러한 알고리즘을 HBase에 구현하여서 실험한 결과 write throughput 은 향상되었고 read/write amplification 은 줄어들었다.

주요어: 고성능 저장장치, 데이터 관리 기법, 운영체제, 메모리 관리
학번: 2010-23270