Towards High Performance Big Data Processing by Making Use of Non-volatile Memory

Shuichi Oikawa
University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, Japan

Keywords: Non-volatile Memory, Operating Systems, Storage, Big Data Processing.

Abstract: Cloud computing environments for big data processing require high performance storage. There are emerging high performance memory storage technologies, such as next generation non-volatile (NV) memory and battery backed NV-DIMM. While their performance is much higher than the current block storage devices, such as SSDs and HDDs, they provides only limited capacity. Such limited capacity makes it difficult for memory storage to be adapted as mass storage, and their uses in cloud computing environments have been severely limited. This paper proposes a method that combines memory storage with block storage. It makes use of memory storage as cache of block storage in order to remove the capacity limitation of memory storage. The proposed method inherits the high performance of memory storage and also the large capacity of block storage. Therefore, memory storage can be transparently used as a part of mass storage while its low overhead access can accelerate storage performance. The proposed method was implemented as a device driver of the Linux kernel. Its performance evaluation shows that it outperforms a bare SSD drive and achieves better performance on the Hadoop and database environments.

1 INTRODUCTION

The importance of big data processing increases more than ever before, and it is convincing that its importance will continue increasing in the future as well. Cloud computing environments are currently the only solution that can provide the scalability required by big data processing since they can scale out their storage capacity along with necessary computing resources. There is no doubt that cloud computing environments for big data processing require high performance storage; thus, SSDs were quickly adapted in such environments, and they are sometimes combined with HDDs to transparently enhance the performance and capacity of storage.

Now, high performance memory storage technologies, such as next generation non-volatile (NV) memory and battery backed NV-DIMM, are emerging. These new kinds of storage provide both high performance and persistency, and they are byte addressable. Since their byte addressability enables them to be accessed as memory, we call them *memory storage*. While they provide much higher performance than the current block storage devices, such as SSDs and HDDs, their capacities are limited. Such capacity limitation makes it difficult for memory storage to be adapted as mass storage, and their uses in cloud computing environments have been severely limited.

This paper proposes a method that combines memory storage with block storage. It makes use of memory storage as cache of block storage in order to remove the capacity limitation of memory storage. Combining block storage with another faster block storage, which is typically an SSD, for higher access performance is a well known technique (Kgil and Mudge, 2006; Koller et al., 2013; Saxena et al., 2012). The technique utilizes faster block storage as cache and stores frequently accessed data in it in order to improve the average time to access data. Its open source implementation is widely available (Facebook, 2014). The existing technique employs a software layer that combines two block storage devices. Since it is possible for memory storage to emulate block storage and to use the software layer for combining block storage devices, the emulation sacrifices its performance advantage for the compatibility with the block storage interface.

The proposed method directly manages memory storage in order to make use of its high performance and byte addressability. The byte addressability of memory storage enables its direct management without a device driver; thus, the memory storage man-

Oikawa S..
Towards High Performance Big Data Processing by Making Use of Non-volatile Memory.
DOI: 10.5220/0005463605290534
In *Proceedings of the 5th International Conference on Cloud Computing and Services Science* (CLOSER-2015), pages 529-534
ISBN: 978-989-758-104-5

agement can be integrated in a device driver that combines memory storage with block storage without an additional software layer as required by the existing method. It can effectively utilize the high performance of memory storage and also provides the large capacity of block storage. Therefore, memory storage can be transparently used as a part of mass storage while its low overhead access can accelerate storage performance.

The proposed method was implemented as a device driver of the Linux kernel, and its performance evaluation was performed by measuring the file access performance on the Hadoop distributed processing environment and also a typical benchmark performance on the MySQL database environment. Hadoop and MySQL were employed for the measurements in order to evaluate the effectiveness of the proposed method in realistic environments. The measurements were performed on a virtualized environment. The evaluation results show that the proposed method considerably outperforms a bare SSD drive and achieves better performance on the Hadoop and database environments.

The rest of this paper is organized as follows. Section 2 describes the background of the work. Section 3 describes the detailed design and implementation of the proposed method. Section 4 shows the result of the experiments. Section 5 describes the related work. Section 6 summarizes the paper.

2 BACKGROUND

This section describes the background of this work, which includes the overview of the block device driver layer of the operating system (OS) kernel and the existing method to combine block storage devices.

2.1 Block Device Driver Layer

The current storage devices, such as SSDs and HDDs, are block devices, and they are not byte addressable; thus, CPUs cannot directly access the data on these devices. A certain size of data, which is typically multiples of 512 byte, needs to be transferred between memory and a block device for CPUs to access the data on the device. Such a unit to transfer data is called a block.

The OS kernel employs a file system to store data in a block device. A file system is constructed on a block device, and files are stored in it. In order to read the data in a file, the data first needs to be read from a block device to memory. If the data on memory was modified, it is written back to a block device. A memory region used to store the data of a block device is called a page cache. Therefore, CPUs access a page cache on behalf of a block device.

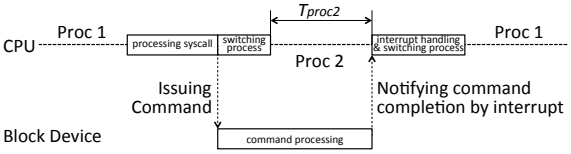

Figure 1: The asynchronous access command processing and process context switches.

Since HDDs are orders of magnitude slower than memory to access data on them, various techniques were devised to amortize the slow access time. The asynchronous access command processing is one of commonly used techniques. Its basic idea is that a CPU executes another process while a device processes a command. Figure 1 depicts how it works. Process 1 issues a system call to access data on a block device. The kernel processes the system call and issues an access command to the corresponding device. The kernel then looks for the next process to execute and perform context switching to Process 2. Meanwhile, the device processes the command, and sends an interrupt to notify its completion. The kernel handles the interrupt, processes command completion, and performs context switching back to Process 1. T_{proc2} is a time left for Process 2 to run. Because HDDs are slow and thus their command processing time is long, T_{proc2} is long enough for Process 2 to proceed its execution.

The I/O request queueing mechanism that implements the asynchronous access command processing has been a right choice for the block devices. It poses high processing cost, but the cost pays off by creating additional processing times made available for other processes. Such justification for the I/O request queueing mechanism and he asynchronous access command processing is, however, no longer true when storage becomes much faster.

2.2 Problems to Combine Memory Storage with Block Storage

The existing method combines block storage with another faster block storage, which is typically an SSD, for higher access performance (Kgil and Mudge, 2006; Koller et al., 2013; Saxena et al., 2012). It utilizes faster block storage as cache and stores frequently accessed data in it in order to improve the average time to access data. Its open source implementation is widely available (Facebook, 2014), and the current Linux kernel includes several implementations, such as dm-cache and bcache.

The implementations of the existing method in the Linux kernel employ the device mapper mechanism to constitutes a single storage device. The device mapper is implemented as a software layer in the kernel, and provides the mechanism to transfer access requests for the constituted device to appropriate underlying devices. The policy part defines how it transfers requests. There can be multiple policy implementations, and some of them combines block storage with faster storage as cache. When an SSD is used as cache storage by combining it with a HDD, it is straightforward that the combined storage provides the block storage interface and is accessed asynchronously since both of them are block storage. As its extension, it is possible for memory storage to emulate block storage and to have the device mapper to combine block storage with memory storage.

The use of the device mapper requires memory storage to emulate block storage interface since the device mapper expects it as an interface. While such emulation enables the use of the device mapper, it causes significant software overhead. The device mapper is basically a block device driver; thus, it receives access requests from the upper generic block device driver framework. It then transfers the received requests to another block storage device. The transferred requests are processed again by the generic block device driver framework, and finally the target block storage device receives them (Ueda et al., 2007). Therefore, processing in the generic block device driver framework occurs multiple times, and such processing causes a software overhead that can be hidden in the long access latency of block storage devices but becomes apparent for memory storage.

3 DESIGN AND IMPLEMENTATION OF THE PROPOSED METHOD

This section first describes the design of the proposed method. It next describes the implementation in the Linux kernel.

3.1 Design Overview

The most considerable advantage of memory storage is its performance. In order to make use of it as much as possible and not to sacrifice it, the software overhead to access it must be minimum. The existing method to combine block storage devices is, however, inappropriate in this sense because of its inefficiency that is inherent in its use of the block storage interface and its asynchronous access. As described in Section 2.1, the overhead of the block storage interface and its asynchronous access is significant, and it must be avoided.

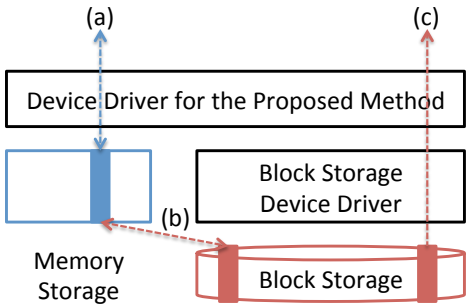

Figure 2: The design overview of the proposed architecture.

The proposed method keeps its access overhead to memory storage minimum by making use of the direct and synchronous access to memory storage. Memory storage provides the memory interface, which means that there is no need to use a device driver to access it; thus, the device driver of the proposed method can directly access memory storage. Such direct access allows the least access overhead to memory storage. Moreover, because memory storage allows synchronous access, of which software overhead is much less than asynchronous access, the proposed method aggressively makes use of synchronous access.

Figure 2 depicts the design overview of the architecture of the proposed method. First, we consider reading data. There are three access paths, which are shown as (a), (b), and (c) in the figure. When data is available on memory storage, which is shown as (a), the device driver of the proposed method provides direct and synchronous access to memory storage. Such access enables the least overhead; thus it should be utilized as much as possible. In order to make it possible, data needs to be read ahead from block storage to memory storage, which is shown as (b). When reading ahead is successful, data can be continuously read from memory storage.

When data is not available on memory storage, a straightforward way is to read in the requested data from block storage to memory storage. This way, however, unnecessarily pollutes memory storage because the data that was read in to memory storage becomes useless. Therefore, the proposed method reads the data from block storage bypassing memory storage, which is shown as (c).

Second, we consider writing data. There are also three data paths, (a), (b), and (c), which are shown in the figure. Unlike reading, there is no need to read ahead into memory storage for writing since

valid data is written onto memory storage. Therefore, data can be written onto memory storage whenever free spaces are available on memory storage, which is shown as (a). The free spaces can contain valid data for reading. Unavailable spaces of memory storage are those where dirty data resides. The unavailable spaces that contain dirty data become free spaces when the dirty data is written back to block storage, which is shown as (b). When there is no free space available, data can be written to block storage, which is shown as (c). The path (c) is, however, considered to be rarely used since writing to memory storage and writing back to block storage can be processed in parallel.

The device driver of the proposed method manages memory storage and also interacts with a block storage device driver. A block storage device driver is not a part of the driver of the proposed method. By separating the management of memory storage and block storage, there is no restriction of a choice of block storage, and arbitrary block storage can be combined with memory storage.

3.2 Implementation in the Linux Kernel

The Linux kernel provides the device mapper mechanism, which can be used to combine multiple block storage devices. The existing method uses this mechanism as described in Section 2.2. The proposed method, however, does not use the device mapper mechanism in order to avoid the overhead of itself and also the overhead incurred by having memory storage emulate block storage.

The proposed method implements its own function that can provide the synchronous access interface depending upon the location of requested data.

```
void memory_make_request(
    struct request_queue *q,
    struct bio *bio)
```

This interface is typically used by the device driver of ramdisk, which provides synchronous access. The proposed method makes use of this interface and provides synchronous access when data is available on memory storage for reading or when a free space is available on memory storage for writing. In this case, memory storage is considered to be working just as ramdisk. When data is unavailable on memory storage for reading or when no free space is available on memory storage for writing, however, the access request is transferred to the device driver of block storage. Then, the block storage device driver asynchronously processes the request.

The device driver of the proposed method also implements the functions for reading ahead and writing back data between memory storage and block storage. Because they need to be invoked in parallel with reading and writing data from/to memory storage, the dedicated kernel threads process them. They are invoked at appropriate timings in order to improve the efficiency of the proposed method.

4 EXPERIMENT RESULTS

First, file I/O throughput was measured by using the Hadoop TestDFSIO benchmark program to see performance impact on big data processing. The measured costs are compared with a sole SSD drive. Second, the performance of the database processing was measured by the TPCC-MySQL benchmark program to see performance impact on database processing.

4.1 Experiment Environment

Since there is no publicly available system that equips memory storage, we used DRAM to emulate it. Since MRAM, which is considered to be the best match for the proposed method, performs comparably to DRAM, the differences of results must be negligible. All measurements described below were performed on the Linux kernel 3.14.12 that includes the implementation of the proposed method. Execution times were measured using the TSC (Time Stamp Counter) register.

The system used for this experiment is a PC system equipped with the Intel Core i7-4930K 3.4GHz and 64GB of DRAM. The KVM virtualization software of Linux is employed to construct experiment environments that consist of virtual machines. Each virtual machine is configured with two CPUs, the main memory, and a dedicated block storage device. The sizes of the main memory differ to match their functionality, they are described below. The CFD S6TNHG6Q 128GB SATA SSD is used for a dedicated block storage device, and a whole SSD is assigned to a single virtual machine. When the proposed method is used for an experiment, memory storage consists of 1GB of memory.

4.2 Results of Hadoop TestDFSIO

This section shows the measurement results of the Hadoop TestDFSIO benchmark program. For this experiment, four virtual machines were configured to be a Hadoop cluster. One virtual machine becomes the master node, and the others are slave nodes. The main memory size of the master node is 8GB, and that of

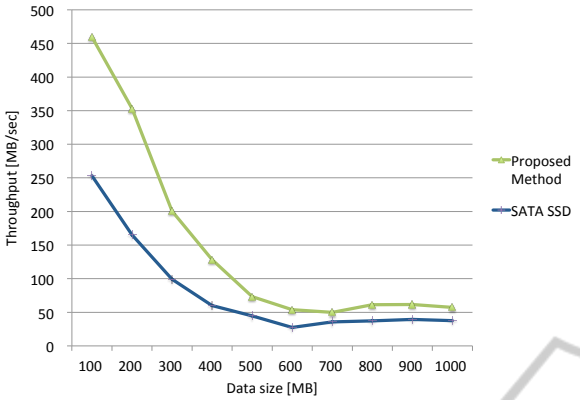

Figure 3: Comparison of read performance by Hadoop TestDFSIO.

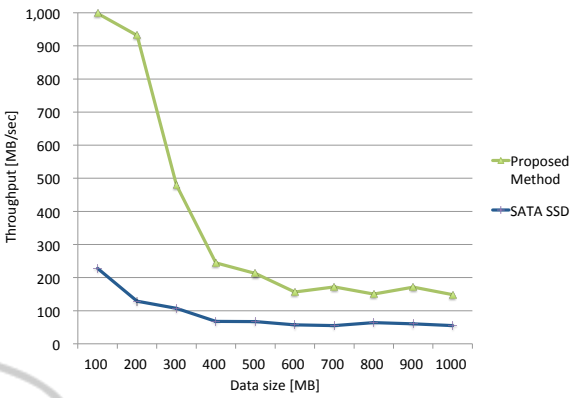

Figure 4: Comparison of read performance by Hadoop TestDFSIO with SCR enabled.

slave nodes is 1GB. Hadoop employs Hadoop Distributed File System (HDFS) for the file service of its applications (Shvachko et al., 2010). The HDFS servers consist of the name node and data nodes, which are executed as user processes. The master node runs the name node, which locates on which data node requested data files reside upon access requests from clients. The data nodes of slave nodes manage data files.

We measured the file I/O throughput of reading files of various data sizes by using the Hadoop TestDFSIO benchmark program. Larger numbers are better as I/O throughput. The size of each file created for measurements was fixed to 100MB, and the number of files was changed from one to ten in order to change the total data sizes from 100MB to 1GB. We first executed the writing program of TestDFSIO to create files for reading. After flushing the page cache of the data nodes, we executed the reading program of TestDFSIO, and measured the costs. HDFS provides two methods for reading. One receives data from a data node through remote procedure calls (RPCs), and the other directly interacts with a local file system. The latter one is called short circuit read (SCR). Both methods were used for measurements. Figure 3 and 4 show the results without and with SCR enabled, respectively.

The measurement results show a significant performance advantage of the proposed method for the Hadoop TestDFSIO. For reading from 100MB to 1GB data sizes without SCR, it performs approximately 39.21% to 114.36% better than SSD. For reading with SCR enabled, it performs approximately 135.32% to 624.10% better than SSD. On average, the proposed method performs 78.45% better without SCR and 266.08% better with SCR than SSD.

A realistic evaluation with Hadoop shows that the proposed method provides a significant boost with the

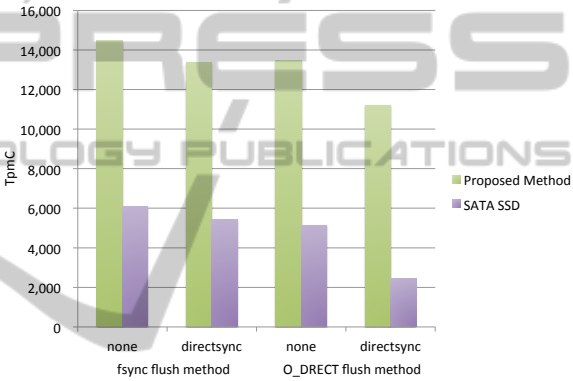

Figure 5: Comparison of TPCC-MySQL performance.

file access throughput of Hadoop. Therefore, it is certain that a wide range of applications, which involves a large amount of file access, can benefit from it.

4.3 Results of TPCC-MySQL

This section shows the measurement results of the TPCC-MySQL benchmark program. For this experiment, a single virtual machines with 8GB main memory was configured. The number of warehouses is 40, which constitute approximately 4GB of a database. The buffer pool size of the InnoDB storage engine is 4GB. The measurements were performed with the two flush methods of InnoDB and the two storage cache modes of KVM; thus, there are the four combinations of them. The Innodb flush methods used for the measurement are fsync and O_DIRECT, and the KVM storage cache modes are none and directsync. The none cache mode provides the write buffer while the directsync cache mode does not. Figure 5 shows the results.

The performance improvement enabled by the

proposed method is significant. The proposed method executes the benchmark from 2.39x to 4.60x faster than SSD. The difference between the proposed method and SSD is the largest when the combination of the O_DIRECT Innodb flush method and the KVM directsync cache mode is used. Since this combination provides no buffering of data transfer in the OS kernel and the KVM virtualization software, the cost to write data in storage becomes the maximum among the combinations used for the experiments. The other combinations provide buffering somewhere in the OS kernel and the KVM virtualization software; thus, the differences are closer but still large, which are from 2.39x to 2.62x.

5 RELATED WORK

A technique to combine block storage with another block storage for higher access performance existed before SSDs become widely available and popular. DCD (Hu and Yang, 1996) first stores data in cache storage, so that it can make use of sequential access, of which performance is typically much better than random access, so that the write performance can be improved. The emergence of SSDs stimulated the research and development of various caching techniques (Kgil and Mudge, 2006; Koller et al., 2013; Saxena et al., 2012; Facebook, 2014) in order to make use of their high performance. Because SSDs are block storage, all of them combine block storage with another block storage, and provide the block storage interface. The proposed method is different from them since it combines memory storage with block storage. Because memory storage allows synchronous access, the proposed method aggressively makes use of it in order to reduce the access cost in total.

The Linux kernel provides the device mapper as the software layer to combine multiple storage devices and to constitutes a single storage device. When the device mapper is used to combine memory storage with block storage, it requires memory storage to emulate block storage since the device mapper can interact only with the block storage interface. It also causes significant software overhead since the access requests can be processed by the generic block device driver framework multiple times(Ueda et al., 2007). The proposed method does not use the device mapper mechanism in order to avoid such overheads, and implements its own function that can provide the direct and synchronous access interface to memory storage.

6 SUMMARY

Memory storage technologies are emerging, and they should be effectively utilized in cloud computing environments in order accelerate storage performance for big data processing. This paper proposed a method that combines block storage with memory storage and makes use of memory storage as cache of block storage in order to remove such limitation. The proposed method effectively utilizes the high performance of memory storage and also provides the large capacity of block storage. Therefore, memory storage can be transparently used as a part of mass storage while its low overhead access can accelerate storage performance. The proposed method was implemented as a device driver of the Linux kernel. Its performance evaluation shows that it outperforms a bare SSD drive and provides better performance on the Hadoop and database environments.

REFERENCES

Facebook (2014). Flashcache. https://github.com/facebook/flashcache.

Hu, Y. and Yang, Q. (1996). Dcd – disk caching disk: A new approach for boosting i/o performance. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 169–178.

Kgil, T. and Mudge, T. (2006). Flashcache: A nand flash memory file cache for low power web servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 103–112, New York, NY, USA. ACM.

Koller, R., Marmol, L., Rangaswami, R., Sundararaman, S., Talagala, N., and Zhao, M. (2013). Write policies for host-side flash caches. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 45–58, Berkeley, CA, USA. USENIX Association.

Saxena, M., Swift, M. M., and Zhang, Y. (2012). Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA. ACM.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA. IEEE Computer Society.

Ueda, K., Nomura, J., and Christie, M. (2007). Request-based device-mapper multipath and dynamic load balancing. In *Proceedings of the Linux Symposium*, volume 2, pages 235–243.