Information Systems 91 (2020) 101488

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Compatible byte-addressable direct I/O for peripheral memory devices in Linux

Sung Hoon Baek [a], Ki-Woong Park [b,*]

[a] Department of Computer Engineering, Jungwon University, South Korea
[b] Department of Computer and Information Security, Sejong University, South Korea

ARTICLE INFO

Article history:
Received 6 March 2018
Received in revised form 16 May 2019
Accepted 17 December 2019
Available online 2 January 2020
Recommended by Jens Teubner

Keywords:
Operating system
Input output
Memory

ABSTRACT

Memory devices can be used as storage systems to provide a lower latency that can be achieved by disk and flash storage. However, traditional buffered input/output (I/O) and direct I/O are not optimized for memory-based storages. Traditional buffered I/O includes a redundant memory copy with a disk cache. Traditional direct I/O does not support byte addressing. Memory-mapped direct I/O, which optimizes file operations for byte-addressable persistent memory and appears to the CPU as a main memory. However, it has an interface that is not always compatible with existing applications. In addition, it cannot be used for peripheral memory devices (e.g., networked memory devices and hardware RAM drives) that are not interfaced with the memory bus. This paper presents a new Linux I/O layer, byte direct I/O (BDIO), that can process byte-addressable direct I/O using the standard application programming interface. It requires no modification of existing application programs and can be used not only for the memory but also for the peripheral memory devices that are not addressable by a memory management unit. The proposed BDIO layer allows file systems and device drivers to easily support BDIO. The new I/O achieved 18% to 102% performance improvements in the evaluation experiments conducted with online transaction processing, file server, and desktop virtualization storage.

1. Introduction

DRAM takes on the important function of the main memory in computers, and it is also used to cache and buffer data to improve disk performance. Recently, DRAM has been used as a storage system to provide a fast response time that cannot be provided by disk and flash systems. In-memory computing is used for intensive random accesses in various fields such as large-scale caching systems [1,2], in-memory databases [3], cloud computing [4–6], virtual desktop infrastructure [7,8], and web search engines [9].

A disk cache improves the read performance when the hit rate is high. Even a 1% miss ratio for a DRAM cache can lead to a tenfold reduction in performance. A caching approach could lead to the faulty assumption that "a few cache misses are okay" [10]. An alternative choice to disk caching is a ramdisk.

A ramdisk is a software program that takes up a portion of the main memory, i.e., DRAM chips on DIMM modules. The computer uses this block of memory as a block device that can be formatted to a file system format, then mounts the file system on it. It is sometimes referred to as a software RAM drive to distinguish it from a hardware RAM drive, which is provided by a type of solid-state drive (SSD).

The performance of a ramdisk, in general, is orders of magnitude faster than other forms of storage media, such as an SSD (up to 100 times faster) or a hard drive (up to 200 times faster) [11]. It is the fastest type of storage media available, but it cannot retain data without power. To address this problem, many studies have focused on various types of ramdisk systems, a single system to cluster systems [4,12,13].

Thanks to practical developments that have overcome the volatility of RAM, RAM has become a storage medium in more systems. The ramdisk has the same interface as the conventional hard disk, but the traditional block interface is not optimized for RAM as a storage medium. Prefetching, disk scheduling, and the disk cache are designed for hard disk drives, and these degrade the performance of the ramdisk. Prefetching and disk scheduling can be easily and transparently turned off for applications, but the disk cache cannot.

Block devices, such as hard disk drives, transfer data in block units. The disk cache allows for applications to process input/output (I/O) in byte units and it improves the I/O performance. The disk cache, referred to as the page cache in Linux, is configured as some part of the main memory. However, the

* Corresponding author.
E-mail addresses: shbaek@jwu.ac.kr (S.H. Baek), woongbak@sejong.ac.kr (K.-W. Park).

https://doi.org/10.1016/j.is.2019.101488
0306-4379/

page cache is useless for memory devices such as ramdisk and persistent memory (PMEM).

1.1. Related work

The advent of PMEM and the evolution of large-scale memory have led to many challenges in the field of data storage. Direct Access (DAX), a new optimized I/O interface for PMEM is now available for Linux. RAM-based file systems were designed to optimize themselves for temporary files.

1.1.1. Persistent memory

PMEM such as phase-change RAM, magnetoresistive RAM, and ferroelectric RAM, leads to challenges for file system designers.

PMEM is byte-addressable and directly accessible from CPU via the memory bus. It offers performance within an order of magnitude of that of DRAM [14].

The byte-addressability of non-volatile memory can make file systems more reliable and simpler. Strongly reliability file systems using PMEM have been proposed in several studies [15,16]. For instance, Dulloor et al. implemented the Persistent Memory File System (PMFS), a light weight POSIX file system, that exploits PMEM's byte addressability and offers direct access with memory-mapped I/O [14,16].

1.1.2. DAX

Direct access (DAX) is a mechanism defined by the Storage Networking Industry Association (SNIA) as part of the non-volatile memory (NVM) programming model that provides byte-addressable loads and stores when working with a PMEM-aware file system through a memory-mapped file interface [17].

Both the ext4 and XFS file systems are capable of DAX. DAX-enabled file systems support the legacy interface, but the direct access is only achieved by the new memory mapping programming model of DAX. Many existing applications that use the traditional I/O application programming interface (API) cannot utilize the features of DAX.

Moreover, DAX requires that PMEM can be accessed as the main memory by the memory management unit (MMU). DAX is not suitable for peripheral memory devices that are not accessible by the MMU. A peripheral memory device can be a clustered storage using RAM [4], an SSD-backed RAM disk [18], or a hardware RAM drive.

1.1.3. RAM-based file system

A RAM-based file system appears as a mounted file system without a block device such as the ramdisk. The temporary file system tmpfs is a typical RAM-based file system that appears in various UNIX-like operating systems. It creates a file system using the shared memory of the kernel, provides a byte interface without the page cache, and directly transfers data between shared memory and user memory. On reboot, everything in tmpfs is lost, so it is used for temporary file storage. It has no recovery scheme after a reboot even if a nonvolatile memory is used.

Tmpfs provides the best performance as a file system to store temporary files. It uses only the main memory, that means it cannot be used with the other durable storage device such as RAMCloud [4], Apache Hadoop [19–21], SSD-backed RAM disk, and persistent memory.

1.2. Motivation

A new interface must support peripheral memory devices, byte accessibility, and compatibility with the existing applications. The traditional direct I/O and latest memory-mapped direct I/O interfaces do not support all of these needs at once.

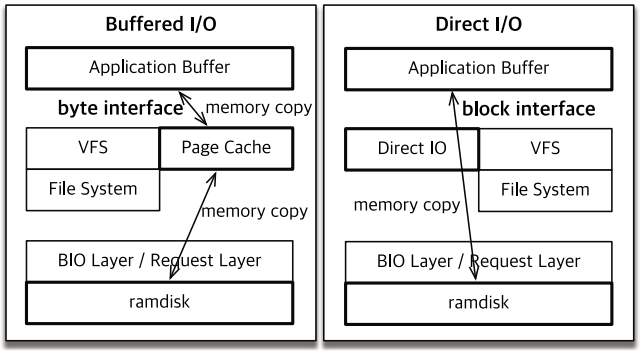

Fig. 1. Kernel structure and data path for buffered I/O and direct I/O. In buffered I/O, there exists an additional memory copy with the page cache. Direct I/O requires one memory copy between the application buffer and the ramdisk.

1.2.1. Constraints of direct I/O

Buffered I/O utilizes the page cache, so it aggregates small amounts of data into an integer multiple of the block size. Consider, for example, a process that requests one character from the kernel. Then, the kernel loads the corresponding block into the page cache. If the process reads the next single character, the request immediately responds with the already loaded block. For a write example, a process sequentially writes one byte of data for each write call. The kernel buffers them in a page and flushes it at a later time.

Fig. 1 shows the kernel structures and data paths for buffered I/O and direct I/O. In buffered I/O, there exist two memory copies; from the ramdisk to the page cache and from the page cache to the application buffer. When direct I/O is used for a file, data is transferred directly from the disk to the application buffer, bypassing the page cache.

Direct I/O bypasses the page cache, but it has several constraints, so applications that use buffered I/O for byte-range operations cannot be easily changed to use direct I/O. With direct I/O, application programs must obey the constraints of the block interface. The user memory and the file position used in read() and write() calls must be aligned in the logical block size. That is, the user memory address, the request size, and request location must be an integer multiple of the logical block size, which is typically 4096 bytes. User applications can obtain the logical block size of the file system with the BLKSZGET ioctl() call. Direct I/O is enabled by calling the open() call with the O_DIRECT flag.

1.2.2. Compatibility

Memory-mapped direct I/O [14,22,23] enables byte-addressable direct access without a system call after establishing a memory mapping, thus providing a significantly low latency after the mapping. DAX uses a new programming interface using persistent memory development kit (PMDK). Also other studies use their own user library [22,23].

DAX-enabled file systems support the legacy interface, but which does not provide the direct feature of DAX. Many traditional existing applications use the standard file API and do not use the direct feature that is provided by the memory-mapping. For such applications, we need a new I/O layer that makes the standard file API support byte-addressability and direct accessibility without needing any changes in the existing applications.

1.2.3. Support for peripheral memory device

A peripheral memory device is a memory-based storage that is interfaced with the peripheral I/O bus and not accessed by memory mapping. Such devices cannot use memory-mapped direct I/O such as DAX. Thus, we need another new approach for them.

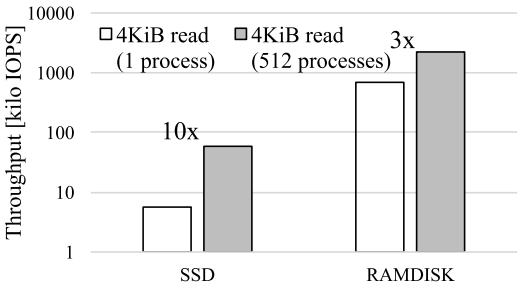

Fig. 2. Comparison of the 4 KiB random read performance for one process and 512 processes in an SSD and a ramdisk. The *y*-axis is in logarithmic scale. The ramdisk exhibits a smaller performance gap for the different numbers of processes.

Networked ramdisks (RAMCloud and Apache Hadoop with ramdisk), SSD-backed ramdisks, and hardware ramdisks interfaced with the I/O bus are types of the peripheral memory device.

RAMCloud utilizes remote ramdisks of clustered nodes to provide durability to DRAM [4,13]. It aggregates the main memories in thousands of servers, which keep the information entirely in DRAM. RAMCloud maintains redundant data copies across multiple nodes; thus, it can recover from crashes providing durable and available storage.

Apache Hadoop with ramdisk is a popular open-source software for reliable and scalable distributed computing. Apache Hadoop uses its own Hadoop File System (HDFS), which is a distributed file system that can scale form a single cluster to hundreds of nodes. HDFS [19] stores multiple replicas of a block on different data nodes, thereby providing availability and robustness. It also supports ramdisks on data nodes [20,21]. The data nodes will flush in-memory data to the disk asynchronously. The Hadoop cluster system has its own filesystem on the ramdisk or a persistent storage.

SSD-backed ramdisk provides strict durability in a local node that uses a ramdisk. This is similar to a mirrored disk array that is composed of a ramdisk and a flash-based SSD. Write requests are delivered to both the ramdisk and the SSD, but read requests are served by the ramdisk only [18]. This storage device is implemented as a typical block device, so it cannot use tmpfs and DAX.

1.2.4. Flash memory cannot replace RAM storages

Flash-based SSDs can potentially utilize up to the full bandwidth of the I/O bus by maximizing the parallelism of multiple flash chips, which have a lower latency than a disk. Flash cannot replace RAM storages for the following reasons:

- Tenfold slower latency: To transfer a 4 KiB block, flash needs 50 us and DRAM needs 5 us. Flash has additional latency in its device driver, the host bus adaptor, and the controller in the SSD.
- Limited lifetime: An SSD becomes unreliable beyond a limited number of program/erase (P/E) cycles. A write causes a programming cycle and may cause an erasure cycle; these cycles wear out the tunnel oxide layer of the transistors. A 2-bit multi-level cell (MLC) flash memory fabricated using the 2x nm process has a maximum lifetime of 3,000 program/erase cycles. A 3-bit MLC flash memory has a lifetime of only a few hundred cycles. Write-intensive workloads may make the lifetime much shorter than the warranted lifetime. Thus, SSD reduces (throttles) write performance by adding throttling delays to write requests, so as to guarantee the required SSD lifetime [24].
- Poor performance for small I/O with low concurrency. An SSD can perform using the full bus bandwidth using tens or hundreds of independent flash memory chips, but a large number of concurrent requests are required to utilize all independent chips. A small request consisting of a single process just utilizes only a single chip, thereby leading to low performance. Fig. 2 compares the 4 KiB random read performances for one process and 512 processes in an SSD and a ramdisk. The *y*-axis is in logarithm scale. Here, the SSD exhibits a 10 times performance gap for the different numbers of processes, but the ramdisk shows a three times performance gap.

1.3. Our contributions

The traditional block device suffers from an additional memory copy with the page cache. However, direct I/O cannot process byte-range requests. The conventional RAM-based file system cannot be used with peripheral memory devices such as RAMCloud, HDFS, or SSD-backed ramdisks, and etc. Flash cannot be a complete replacement of RAM. DAX requires a new programming interface that is given by persistent memory development kit (PMDK) [25]. This paper presents a new compatible I/O layer that is called byte-addressable direct I/O (BDIO) in Linux for RAM-based storages. BDIO has the following characteristics.

- Compatibility: BDIO is transparent to applications. No changes to applications are required for them to support it. The proposed scheme utilizes the standard file API.
- Page cache bypass: The application bypasses the page cache even if the buffered I/O interface is used.
- Byte-range I/O: Unlike direct I/O, which has a block interface, the proposed I/O has a byte interface. Therefore, an application program using byte-range buffered I/O can use BDIO without modification.
- Peripheral memory devices: BDIO can support peripheral memory devices that cannot be accessed by the MMU.
- Consistency with buffered write: The proposed scheme provides data consistency even if buffered I/O is mixed with BDIO. This is useful for the SSD-backed ramdisk, which must use buffered writes to SSD but can allow byte direct read (BDR) from RAM to improve read performance.

BDIO was implemented in a Linux kernel. The block device and the file system that supports BDIO need an additional interface. We implemented a BDIO-capable ramdisk and a BDIO-capable SSD-backed ramdisk, and revised XFS and ext4 to support BDIO.

2. Design and implementation of byte direct I/O

BDIO transfers data directly between a ramdisk and a user application buffer, where the application performs byte-range I/O with the same interface for the buffered I/O without any modification of applications.

2.1. Structure for cache-bypass

The ramdisk that supports BDIO has both the block interface and the BDIO interface as shown in Fig. 3. Thus, the BDIO ramdisk can be appeared as a conventional block device but has additional BDIO capability. The new kernel supports the BDIO layer, which directly transfers data between the application buffer and the BDIO ramdisk.

Applications use the traditional system-call interface for BDIO. Whenever an I/O request is delivered, the kernel checks whether the storage device related to the request is capable of BDIO. If

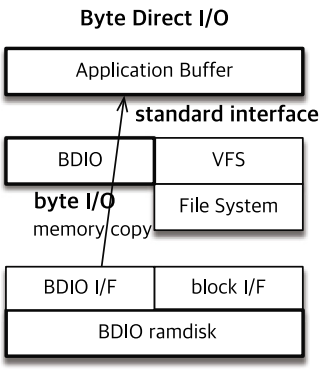

Fig. 3. Ramdisk supporting BDIO with both the block interface and the BDIO interface. It can appear as a traditional block device and has the additional BDIO capability. The proposed kernel supports the BDIO layer, which directly transfers data between the application buffer and the BDIO ramdisk with the standard system call interface.

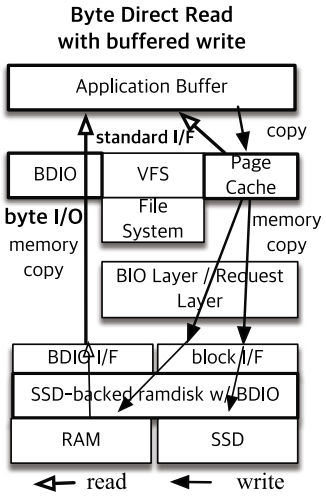

Fig. 4. Data paths of the BDR with buffered write: the SSD-backed ramdisk uses both BDR and the buffered write. The BDIO layer maintains data consistency even though BDRs are mixed with buffered writes.

BDIO is enabled, the request is forwarded to the BDIO layer; otherwise, the request is processed in the traditional path.

The BDIO ramdisk has the traditional block interface and an additional new interface with the BDIO layer. I/O requests processed by BDIO do not pass through the page cache and the block layer. The BDIO ramdisk can turn off or on its BDIO capability at run time using an ioctl call. When the BDIO capability is turned off, the traditional I/O layer is used.

2.2. Byte direct read with buffered write

The SSD-backed ramdisk shown in Fig. 4 uses the BDR from RAM and the buffered write to SSD and RAM. All data written using the buffered write are synchronously updated to both SSD and RAM as in a mirrored disk array. An SSD-backed ramdisk is used in systems that require low read latency and data durability.

An SSD-backed ramdisk cannot process a byte direct write to SSD because SSD has no byte interface. The buffered write is used so that SSD can gathers data in the page cache and transfers them to multiple flash chips in parallel.

The proposed scheme allows data consistency even though the BDRs are mixed with buffered writes. The BDR can also coexist with the page cache that is used by buffered I/O.

The byte direct read places the page cache at the highest priority in the read process. If the requested block is found in the page cache, the found data is sent from the page cache to the application buffer. Otherwise, the read request is delivered to the BDIO layer.

In terms of memory copy, BDR is similar to a traditional memory-mapped file. However, there are many applications that use the legacy interface that is available in BDR. In addition, the memory-mapped file causes more I/Os for a request that is unaligned to the page size. For example, A 1 KB read request requires a 4 KB page read and a 5 KB read leads to two 4 KB page reads.

2.3. Call path for byte-range I/O

The main idea is that the byte-range arguments and memory address of a user request are just passed to a BDIO-enabled device driver, where the BDIO layer just translates the logical location of the user request to the physical location in aid of a file system and splits a physically discontiguous user request into multiple contiguous device requests. The BDIO layer allows file systems and device drivers to easily support BDIO.

Fig. 5 shows the system call path for a read request inside the kernel. Read system calls such as aio_read(), read(), and readv() go through the generic_file_buffered_read() in the Linux kernel. The generic_file_buffered_read() function searches for the cached page in the requested range (using find_get_page). To support data consistency with the buffered IO, the BDR should search the page cache. Because the page cache is searched first, buffered writes and BDRs can be mixed.

If the page is found, the data in the page are copied to the application buffer. If no page is found, the requested position and length of the file and the user buffer address is just delivered to the fs_bd_read() function of the target file system.

A contiguous region in a file may be discontiguous in the physical storage. Thus, the fs_bd_read() splits the user request into multiple lower requests (fops->bd_read()) for each physically contiguous sectors in aid of the get_block() function of the target file system. Each of the contiguous lower requests are finally delivered to the bd_read () of the ramdisk, which has a new byte read interface that allows byte-range direct copy from the ramdisk to the user memory.

The file system passes the function address of its get_block() to the BDIO layer (blockdev_bd_read). Here, the get_block() function translates the file position into the physical position and finds physically discontiguous sectors.

Fig. 6 shows the write system call path inside the kernel. Write calls such as write(), aio_write(), and writev() go through the kernel function generic_file_write_iter(). If the file system and its storage device support BDIO, a write request is processed using BDIO.

First, cache pages that are in the requested range are flushed and invalidated, and then the write request is sent to the byte direct write interface of the file system (fs_bd_write()), which passes the function address of get_block() of the file system to the BDIO layer (blockdev_bd_write()), which in turn translates the requested file positions into physical positions. Function blockdev_bd_write() splits the write request into multiple BDIO requests and passes them to the BDIO device. The BDIO devices have a new byte write interface along with the conventional block interface, so byte-range data is transferred from the user memory to the RAM of the BDIO ramdisk.

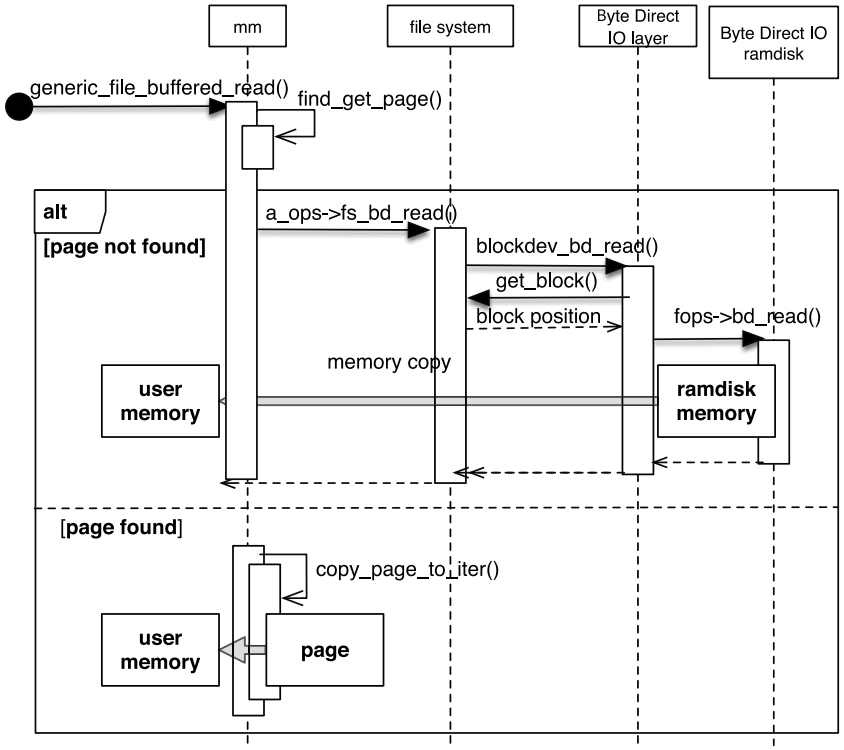

Fig. 5. UML sequence diagram of the BDR: If a cached page is not found, the target file system and the byte direct I/O layer translate the requested position of the file into physical parameters and pass them to the byte direct I/O ramdisk. For data consistency when buffered I/O is mixed with BDIO, each read request should search the page cache first.

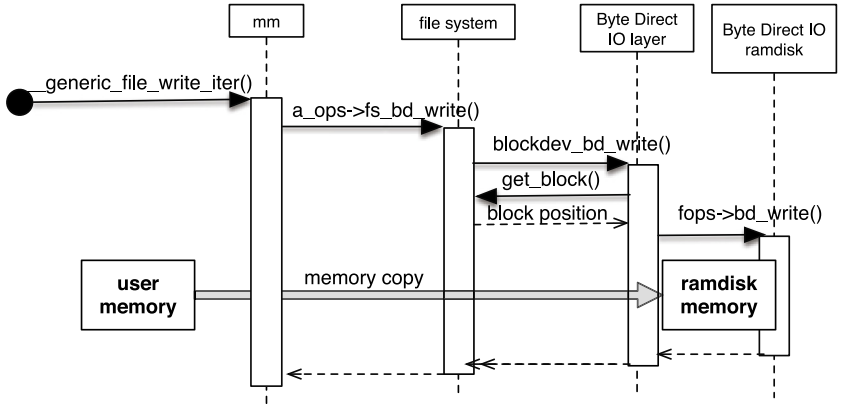

Fig. 6. UML sequence diagram of the byte direct write: The write request bypasses the file system and the byte direct I/O layer and is finally transferred to the byte direct I/O ramdisk. The target file system and BDIO layer translate the requested position of the file into physical positions and pass them to the byte direct I/O ramdisk.

2.4. Transparent interface for applications

BDIO is transparent to applications because it uses the standard file I/O interface that can be used for the peripheral memory devices that are not addressable by a memory management unit. The standard interface requires for applications not to be changed to use BDIO, but BDIO requires for file systems and block devices to have additional interfaces.

2.4.1. File system interface for BDIO

For a file system to support BDIO, it must support one more function than the conventional file system. This function is similar to that of traditional direct I/O, except that BDIO can process byte-range arguments.

The file system uses the structure address_space_operation to register with the virtual file system (VFS). For BDIO, function pointer fs_bd_read() and fs_bd_write() are added to the structure. These function pointers are created as the follows.

```
struct address_space_operations {
 ...
 ssize_t (*fs_bd_read)(struct kiocb *iocb,
      struct address_space *mapping,
      struct iov_iter *iter, ssize_t len);
  ssize_t (*fs_bd_write)(struct kiocb *iocb,
      struct address_space *mapping,
      struct iov_iter *iter, ssize_t len);
};
```

The simplest implementation of these functions is to just call blockdev_bd_read() or blockdev_bd_write(), which are provided by the BDIO layer. An example of this function is shown below.

```
static ssize_t ext4_bd_read(struct kiocb *iocb,
   struct address_space *mapping,
   struct iov_iter *iter, ssize_t len)
{
 struct inode *inode = mapping->host;
 struct block_device *bdev
       = ext4_find_bdev_for_inode(inode);
 return blockdev_bd_read(mapping, inode,
    inode->i_sb->s_bdev, iter, iocb->ki_pos, len,
    ext4_dio_get_block, 0);
}
struct address_space_operations
  ext4_aops = {
  ...
  .fs_bd_read = ext4_bd_read,
  .fd_bd_write = ext4_bd_write,
};
```

The file system function, ext4_bd_read(), passes the function ext4_dio_get_block() of this file system to the blockdev_bd_read() of the BDIO layer. The function ext4_dio_get_block() plays the crucial role of finding the logical block number in the disk for the given block number in the file.

The file system functions, fs_bd_read() and fs_bd_write(), can be implemented similarly to the traditional direct I/O functions of the file system, except that the traditional direct I/O function calls the kernel function __blockdev_direct_IO(). fs_bd_read() and fs_bd_write() are implemented by just calling blockdev_bd_read() and blockdev_bd_write() of the BDIO layer, respectively.

2.4.2. Block device interface for BDIO

A block device that supports BDIO has an additional function, which is declared in structure block_device_operations that are used to register the block device in the kernel as shown below.

```
struct block_device_operations {
   ...
   int (*bd_read)(struct block_device *bdev,
     struct iov_iter *iter, loff_t pos, size_t len);
   int (*bd_write)(struct block_device *bdev,
     struct iov_iter *iter, loff_t pos, size_t len);
};
```

These functions have arguments that indicate a byte-range request and an application memory. The argument *bdev* points to the block device, *iter* is the address of user memory, *pos* is the physical offset in bytes, and *len* is the requested data length in bytes.

This block device interface for BDIO can support not only for the memory but also for the peripheral memory devices that are not addressable by a memory management unit.

These functions actually transfer data between the RAM and the application memory. A simple implementation of these functions in a simple BDIO ramdisk are shown ins the following code:

```
static int ramdisk_bd_read(
   struct block_device *bdev,
   struct iov_iter *user_memory,
   loff_t pos, size_t len)
{
  ...
   ret = copy_to_iter(ram_addr, len, user_memory);
  ...
}
static int ramdisk_bd_write(
   struct block_device *bdev,
   struct iov_iter *user_memory,
   loff_t pos, size_t len)
{
  ...
   ret = copy_from_iter(ram_addr, len, user_memory);
  ...
}
static struct block_device_operations
     dramdisk_blkdev_ops = {
   ...
   bd_read = ramdisk_bd_read,
   bd_write = ramdisk_bd_write,
};
```

Supporting BDIO in a block device is simple. We developed a ramdisk supporting BDIO and the SSD-backed ramdisk supporting BDR. The BDIO ramdisk needs only 30 lines to support BDIO from the traditional ramdisk. Only 10 lines are required to support BDR from the SSD-backed ramdisk. The byte direct IO layer makes device drivers simple to support BDIO.

3. Performance evaluation

Linux kernel 3.10 and 4.15 were modified for BDIO. In addition, the ramdisk supporting BDIO, the SSD-backed ramdisk supporting BDR, and XFS and ext4 file systems modified to support BDIO were implemented. The evaluations used the 3.10 kernel and XFS.

Experiments were performed with two 8-core 3.4 GHz Xeon E5-268 W CPUs that were interconnected by eight memory interconnection channels with 128 GiB of main memory. The ramdisk capacity was set to 122 GiB.

BDIO and BDR were evaluated using various benchmarks and real applications from the block level to the file level. All experiments were performed with a small free memory that is below 1 GB to mitigate the effect of the page cache.

3.1. Microbenchmark at the block level

This section evaluates the throughput of 4KiB random reads and 4KiB random writes at the block level. At the block level, no file system was used. Each evaluation ran for 10 min.

In this experiment, the ramdisk that supports BDIO was employed; this BDIO ramdisk can selectively turn off its BDIO capability to evaluate buffered I/O and direct I/O with the same ramdisk module. In this experiment, buffered I/O, direct I/O, and BDIO were compared using the same ramdisk module.

Fig. 7 shows block-level random read performance. BDIO performed 2.9 times better than that of the buffered I/O and 1.18 times better than that of the direct I/O on average. Both BDIO and direct I/O had no redundant memory copies, but BDIO performed better than direct I/O because BDIO has lighter computing complexity than direct I/O in the kernel. Buffered I/O performed noticeably worse. In the ramdisk, I/O did not cause bottlenecks. Instead, computing overhead and memory copies mainly affected the performance.

Fig. 8 shows the random write performance results. The performance differences among the three IO methods are similar to those for random read. The reason why the write performance of BDIO was better than that of direct I/O is again because of the lower computational complexity in the kernel.

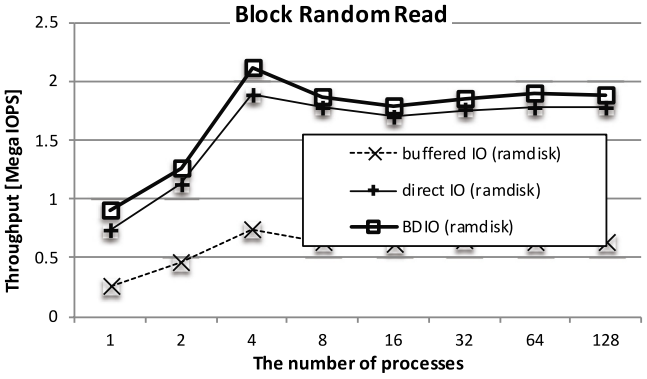

Fig. 7. Random read performance without file system.

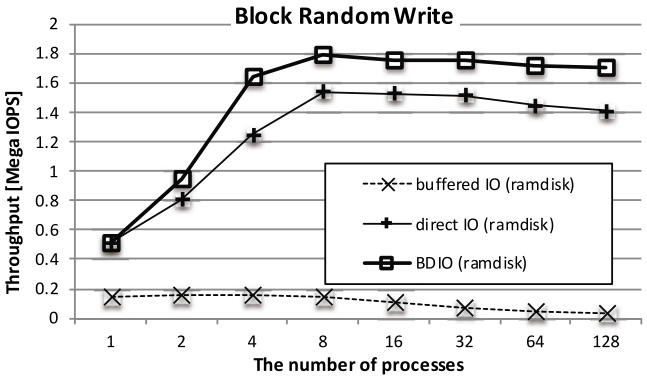

Fig. 8. Random write performance without file system.

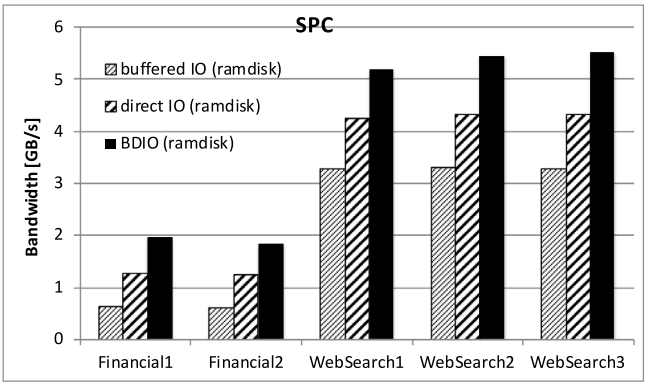

Fig. 9. SPC benchmark with ramdisk.

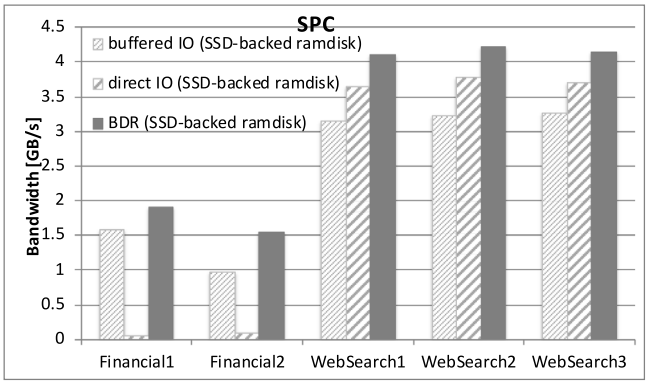

Fig. 10. SPC benchmark with SSD-backed ramdisk.

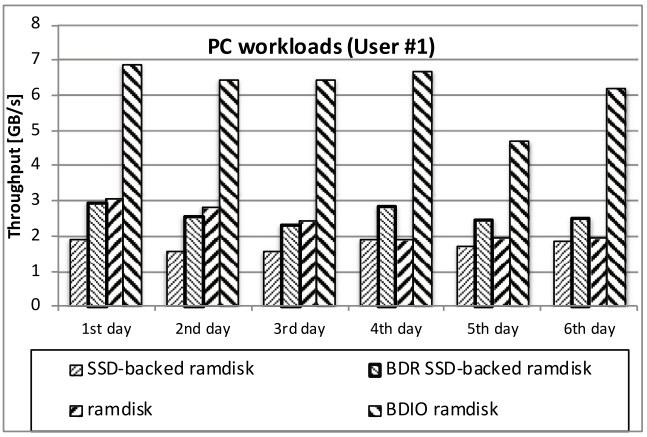

Fig. 11. Performance of BDIO and BDR for the six-day desktop computer workload of a director (user1).

3.2. Macrobenchmarks at the block level

3.2.1. OLTP and web search engine benchmarks

Figs. 9 and 10 show the results for five I/O traces obtained from the UMass Trace Repository of the Storage Performance Council (SPC) [26]. These I/O traces consist of two I/O traces from OLTP applications running at two large financial institutions (Financial1 and Financial2) and three I/O traces from a popular search engine (WebSearch1-WebSearch2). The traces were replayed at block level.

Fig. 9 compares of buffered I/O, direct I/O, and BDIO with the ramdisk. In the random read results reported in Section 3.1, BDIO performed 4.8 times better than that of buffered I/O. For the WebSearch1 trace, the performance of BDIO was 1.59 times better than that of buffered I/O. That is, the actual traces, such as SPC, demonstrate the cache effect of buffered I/O.

Fig. 10 compares the performance of buffered I/O, direct I/O, and BDR of the SSD-backed ramdisk using SPC traces. BDR is a way to process byte-addressable direct reads and to write them in a buffered fashion.

As shown in this figure, the performance of direct I/O for the two OLTP financial traces was very poor. Because the buffered write simultaneously processes large amounts of data that have accumulated in the page cache, it can effectively process dozens of flash memories in the SSD at once. However, direct write serially processes data and suffers from severe performance degradation. The three WebSearch traces consist of a single sequential write and many concurrent reads, which is not bad for direct I/O.

Direct I/O is a well-known method to prevent redundant memory copy, but it was not effective for the SSD-backed ramdisk. BDR and BDIO showed the best performance for the OLTP and web search engine.

3.2.2. PC workload

BDIO and BDR were evaluated using the workload traces of the desktop computers of a director (user1) and an engineer (user2) for six days. The daily workload of user1 consists of an average of 28 GiB and 1.8 millions of reads and an average 8.9 GiB and 269 thousands of writes. The daily workload of user2 consists of average 30 GiB and 1.7 millions of reads per day and average 17 GiB and 842 thousands of writes per day.

Fig. 11 and Fig. 12 show that the performance of the BDIO ramdisk was 2.5 times better than that of the traditional ramdisk, and the performance of the BDR SSD-backed ramdisk was 1.5 times better than the traditional SSD-backed ramdisk on average.

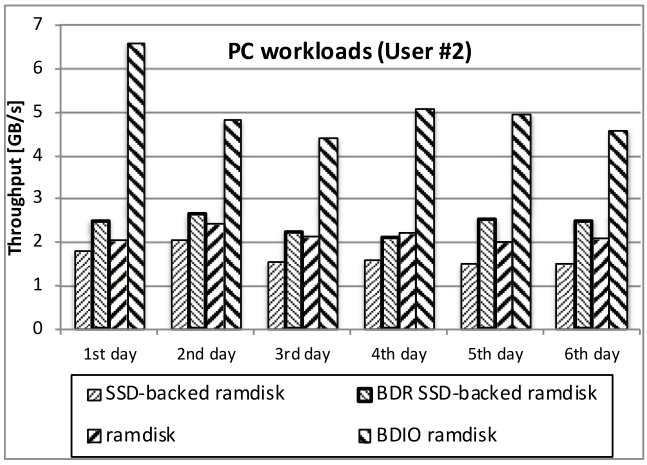

Fig. 12. Performance of BDIO and BDR for the six-day desktop computer workload of an engineer (user2).

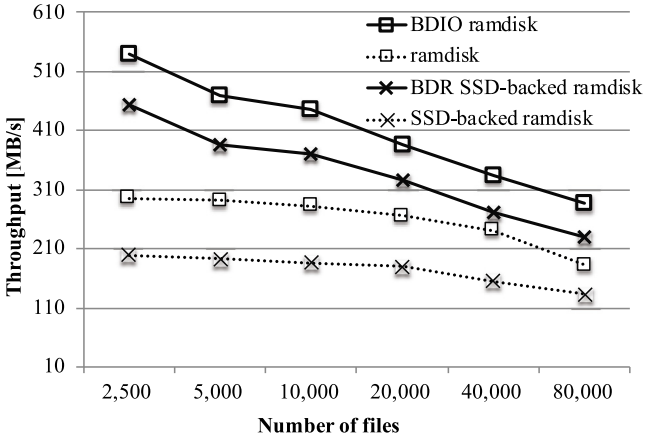

Fig. 13. Throughput of Filebench with the file server workload.

The PC workload results show that BDIO is especially effective for a desktop virtualization server. The desktop virtualization server runs a number of guest operating systems to provide a desktop computer environment to clients through the network. Some desktop virtualization servers use the ramdisk to store the disk image of the guest operating systems [27]. BDIO hence has the potential to solve the performance problems of virtual servers that are experiencing a storage bottleneck.

3.3. File-level benchmark

This section presents the performance of BDIO and BDR when considering the file system. For these experiments, the XFS file system was upgraded to support BDIO.

3.3.1. Filebench

This section presents the results of an evaluation using the file server workload of Filebench, which creates, appends, reads, and deletes files with multiple threads [28]. Figs. 13 and 14 show the throughput and latency of Filebench with respect to the file server workload as the number of files increases. The total size of files ranges from 327 MB to 10.5 GB. The average size of each file is 128 KiB.

The ramdisk with BDIO showed 58% higher throughput and 33% lower latency in comparison to the ramdisk without BDIO on average. The SSD-backed ramdisk with BDR showed 94% higher

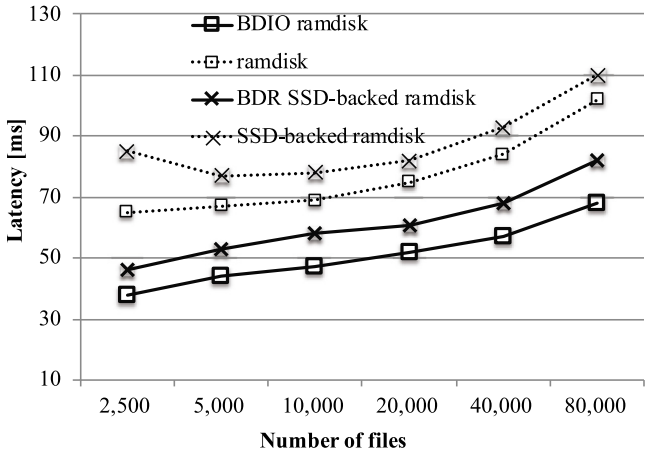

Fig. 14. Latency of Filebench with the file server workload.

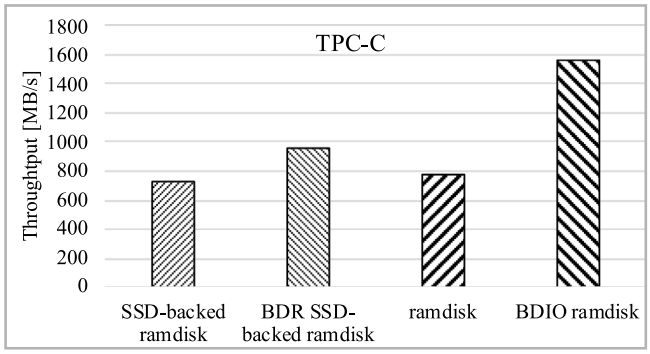

Fig. 15. Performance results with TPC-C benchmark trace on the XFS file system.

throughput and 30% lower latency than that without BDR on average. BDIO and BDR are also quite effective for file servers that share disk access.

3.3.2. TPC-C

Fig. 15 shows the results obtained using a TPC-C benchmarks trace of the Transaction Processing Performance Council (TPC) [29] collected at Microsoft using the event tracing for the Windows framework. TPC-C is a complex online transaction processing benchmark. Because this trace consists of file-level I/O information, this experiment replayed the TPC-C trace on a file system.

The ramdisk with BDIO showed 102% better throughput than the ramdisk without BDIO. The SSD-backed ramdisk with BDR showed 32% higher throughput than that without BDR.

These results demonstrate that BDIO and BDR achieve high performance improvement with various workloads with the file system.

3.3.3. Virtual machine

A Windows 7 guest operating system was installed on the disk using a kernel-based virtual machine (KVM) and the boot time of the operating system was measured using BootRacer. The Linux host operating system ran the BDIO ramdisk and the BDR SSD-backed ramdisk. The file system XFS was mounted on these ramdisks and the disk image file of the guest OS was installed on top of the file system.

With respect to the boot time of a Windows 7 guest operating system on a KVM virtual machine, the BDIO ramdisk outperformed the traditional ramdisk by 18%, and the performance of

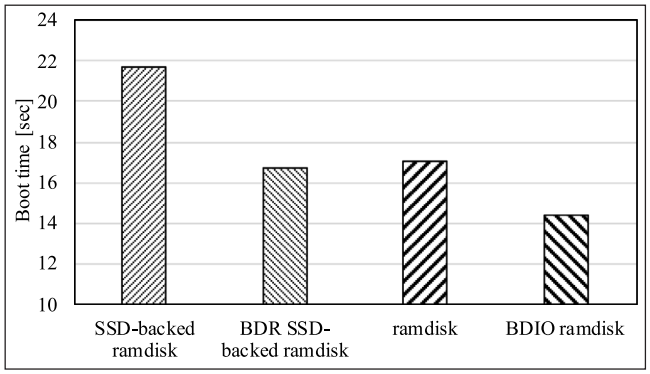

Fig. 16. Boot time of a Windows 7 guest operating system on a KVM virtual machine. The guest operating system was installed on the ramdisks.

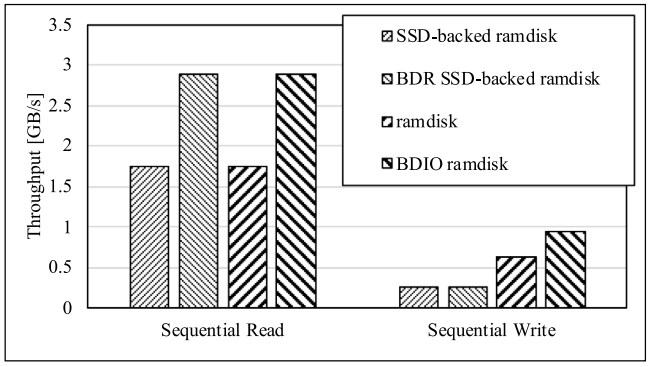

Fig. 17. Sequential read/write performance using benchmark CrystalDisk on the guest operating system, which is installed in the ramdisks.

the BDR SSD-backed ramdisk was 30% better than that of the SSD-backed ramdisk without BDR as shown in Fig. 16.

Fig. 17 shows the performance of the ramdisks running the CrystalDisk benchmark on the guest operating system. The BDIO ramdisk outperformed the ramdisk without BDIO by 65% and 46%, for sequential read and sequential write, respectively.

Through this experiment, we confirmed that the boot time and read/write performance of virtual machines can be greatly improved. The proposed technology will effectively improve the performance of in-memory virtual desktop infrastructures (VDI).

3.4. Applications

Fig. 18 show the running time of several applications. In this experiment, an archived Linux kernel source was extracted using the tar tool (untar), a kernel source database for browsing the source code was built using the cscope tool (cscope), and the kernel source code was compiled (compile). The *y*-axis is the running time, which was normalized by the performance of the SSD-backed ramdisk.

For the untar step, the BDR SSD-backed ramdisk was 9% faster than that without BDR, and the BDIO ramdisk was 8% faster than that without BDIO. For the compile step, the performance difference was very small, about 1%. This is because compilation requires a lot of computing power, which becomes the dominant bottleneck instead of the I/O.

4. Conclusion

This paper presented a new Linux I/O layer, BDIO and BDR, that uses the standard file API for RAM-based peripheral storage.

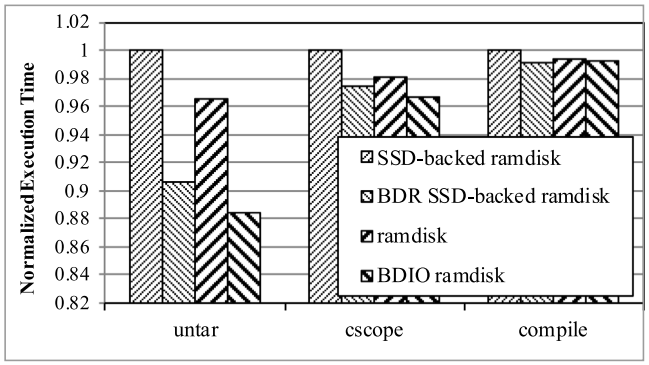

Fig. 18. Running time for extracting (untar), building the source code database (cscope), and compiling the source code of a Linux kernel. The *y*-axis is the running time, which was normalized by the performance of the SSD-backed ramdisk.

BDIO and BDR bypass the page cache without modifying buffered I/O applications to use direct I/O and can perform byte-range I/O without redundant memory copy. In addition, BDR can provide data consistency while using buffered write for the SSD-backed ramdisk.

The BDIO ramdisk, SSD-backed ramdisk supporting BDIO, the BDIO-enabled XFS, and the BDIO layer of the Linux kernel were implemented in Linux so that block devices and file systems can be easily upgraded to support BDIO.

This paper also presented a systematic evaluation of the performance of random I/O, OLTP traces, PC workload, file server, and a desktop operating system on a virtual machine. The results show that the use of BDIO in the file system improved performance by up to 102%.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Sung Hoon Baek: Writing - original draft. **Ki-Woong Park:** Writing - review & editing.

Acknowledgment

This work was supported by a Jungwon University Research Grant (South Korea) (Management Number: 2017-030).

References

[1] B. Fitzpatrick, Distributed caching with memcached, Linux J. 2004 (124) (2004) 5.
[2] D. Zhao, I. Raicu, Hycache: A user-level caching middleware for distributed file systems, in: Parallel and Distributed Processing Symposium Workshops & Ph.D. Forum (IPDPSW), 2013 IEEE 27th International, IEEE, 2013, pp. 1997–2006.
[3] T. Lahiri, M.-A. Neimat, S. Folkman, Oracle TimesTen: An in-memory database for enterprise applications, IEEE Data Eng. Bull. 36 (2) (2013) 6–13.
[4] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S.J. Park, H. Qin, M. Rosenblum, et al., The RAMCloud storage system, ACM Trans. Comput. Syst. 33 (3) (2015) 7.
[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2012, p. 2.

[6] A. Uta, A. Sandu, S. Costache, T. Kielmann, Scalable in-memory computing, in: Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, IEEE, 2015, pp. 805–810.
[7] K. Miller, M. Pegah, Virtualization: virtually at the desktop, in: Proceedings of the 35th Annual ACM SIGUCCS Fall Conference, ACM, 2007, pp. 255–260.
[8] N. Ruest, D. Ruest, Virtualization, A Beginner's Guide, McGraw-Hill, Inc., 2009.
[9] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H.C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al., Scaling memcache at Facebook, in: Nsdi, Vol. 13, 2013, pp. 385–398.
[10] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al., The case for RAMClouds: scalable high-performance storage entirely in DRAM, Oper. Syst. Rev. 43 (4) (2010) 92–105.
[11] datagram, RAMDISK software - What is RAMDisk? 2012, URL http://www.dataram.com/blog/?p=136.
[12] S.T. Diehl, System and method for persistent RAM disk, US Patent 7, 594, 068, Google Patentsm, 2009.
[13] M.D. Flouris, E.P. Markatos, The network RamDisk: Using remote memory on heterogeneous NOWs, Cluster Comput. 2 (4) (1999) 281–293.
[14] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, J. Jackson, System software for persistent memory, in: Proceedings of the Ninth European Conference on Computer Systems, ACM, 2014, p. 15.
[15] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, D. Coetzee, Better I/O through byte-addressable, persistent memory, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ACM, 2009, pp. 133–146.
[16] J. Xu, S. Swanson, {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories, in: 14th {USENIX} Conference on File and Storage Technologies ({FAST} 16), 2016, pp. 323–338.
[17] wikichip, Direct access (DAX) - SNIA, 2019, URL https://en.wikichip.org/wiki/snia/direct_access.
[18] S.H. Baek, A durable and persistent in-memory storage for virtual desktop infrastructures, Korean Inst. Next Gener. Comput. (6) (2016) 23–31.
[19] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, IEEE, 2010, pp. 1–10.
[20] Y. Luo, S. Luo, J. Guan, S. Zhou, A RAMCloud storage system based on HDFS: Architecture, implementation and evaluation, J. Syst. Softw. 86 (3) (2013) 744–750.
[21] A. Bezerra, P. Hernandez, A. Espinosa, J.C. Moure, Job scheduling in hadoop with shared input policy and RAMDISK, in: Cluster Computing (CLUSTER), 2014 IEEE International Conference on, IEEE, 2014, pp. 355–363.
[22] J. Moyer, Persistent memory in Linux, 2016, SNIA NVM Summit.
[23] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, T. Anderson, Strata: A cross media file system, in: Proceedings of the 26th Symposium on Operating Systems Principles, ACM, 2017, pp. 460–477.
[24] S. Lee, T. Kim, K. Kim, J. Kim, Lifetime management of flash-based SSDs using recovery-aware dynamic throttling, in: USENIX Conf. on File and Storage Technologies, FAST, 2012, pp. 1–6.
[25] S.N.P.T. Workgroup, Persistent memory programming, 2002, URL https://pmem.io/pmdk/.
[26] U.T. Repository, OLTP application I/O and search engine I/O, 2009, URL http://traces.cs.umass.edu.
[27] K. technology, Memory and Storage Best Practices for Desktop Virtualization, A Linquidware and Kingston Technology Corporation white paper, 2013.
[28] R. McDougall, J. Mauro, FileBench: File system microbenchmarks, 2008, URL http://www.opensolaris.org/os/community/performance/filebench.
[29] SNIA, TPCC traces 1, 2011, SNIA IOTTA Repository.