Exploiting Peak Device Throughput from Random Access Workload

Young Jin Yu, †Dong In Shin, Woong Shin, Nae Young Song,
Hyeonsang Eom, and Heon Young Yeom
Seoul National Unversity, †Taejin Infotec, Korea

Abstract

In this work, we propose a new batching scheme called *temporal merge*, which dispatches discontiguous block requests using a single I/O operation. It overcomes the disadvantages of narrow block interface and enables an OS to exploit peak throughput of a storage device for small random requests as well as a single large request. Temporal merge significantly enhances device and channel utilization regardless of access sequentiality of a workload, which has not been achievable by traditional schemes.

We extended the block I/O interface of a DRAM-based SSD in cooperation with its vendor, and implemented temporal merge into I/O subsystem in Linux 2.6.32. The experimental results show that under multi-threaded random access workload, the proposed solution can achieve 87%∼100% of peak throughput of the SSD. We expect that the new temporal merge interface will lead to better design of future host controller interfaces such as NVMHCI for next-generation storage devices.

1 Introduction

Advanced memory technology has driven the development of new type of SSDs, and forced us to re-evaluate the whole software stack. Those SSDs have multiple memory chips as their storage medium and a response time of a few microseconds. To make the best use of the low-latency benefit, recent research has focused on minimizing software overhead in the storage stack of an OS, e.g. by communicating with devices via poll [1, 3, 12], short-curcuiting a couple of layers [9], or moving parts of kernel functionality to applications [2].

The technique of merging spatially adjacent I/O requests has been one of the most successful optimizations in handling a storage device [10]. We call this technique as *spatial merge* in further explanation. It helps the OS get the maximum throughput from a storage device by 1) mitigating mechanical overhead in case of HDDs, e.g. seek-time and rotational delay, of merged requests, and 2) accessing multiple memory chips in parallel in case of flash-based SSDs.

However, the exising interface only allows an I/O request to write into or read from "contiguous" sectors. It is natural for HDDs consisting of mechanical moving parts, but, too restrictive for SSDs with no moving parts.

In this paper, we propose a new request batching scheme called *temporal merge*, which combines multiple block requests into one I/O request regardless of their spatial-adjacency. By using an extended block I/O interface, this technique dispatches the merged I/O request to a storage device, which exploits the parallelism inside the device and mitigates the per-request overhead. This approach essentially needs hardware modifications for adding a customized interface beyond the standard. Although it is known to be hard to reach a consensus between OS communities and storage vendors [5], the effectiveness of our solution will be a drive to rethink the current block I/O interface and revise a standard for next-generation host controller interfaces like NVMHCI [6].

Our contribution is about 1) designing a new block I/O interface for low-latency storage devices that finish an I/O request within a few microseconds, and 2) implementing two types of I/O subsystems that utilize the interface. A DRAM-based SSD [11] is chosen as our experimental storage device since 1) it is a low-latency memory-based storage device that requires the re-evaluation of the existing software stacks or schemes, and 2) emerging *Storage-Class-Memory* pursues near-DRAM latency and throughput.

In the following section (§2), we give a brief explanation about the motivation that led us to believe that we need a new batching scheme. Next, we propose an extended block I/O interface that enhances device and channel utilization (§3) and two strategies of merging discontiguous block requests (§4). Performance improvements by our solution are evaluated in §5.

2 Motivation

Peak device throughput is defined as the maximum data transfer rate of a storage device, and random throughput, as the throughput measured under a random access workload. Random throughput achieved by an exisiting I/O subsystem is usually much lower than peak device throughput. Considering that memory-based storage devices show uniform latency to access fixed-size data unlike a disk, we can infer the performance gap would originate from the way of device handling, e.g. the optimizations in an OS and the interface to communicate with a device.

2.1 The Limitations of *Spatial Merge*

Spatial merge builds a single large I/O request from multiple contiguous requests, and achieves peak device throughput, e.g. 80∼100 MB/s in case of a disk. However, this scheme has some limitations, when it comes to handling low-latency memory-based storage devices.

High Software-latency: I/O scheduler blocks up a request queue to prevent I/O requests from being sent to a storage device, which is called *plugging*. From this point on, each I/O request is enqueued into the request queue and tested whether it is spatially-adajcent to any previous requests within.Even if the queue is empty, a newly enqueued I/O request should wait until the queue becomes *unplugged*, usually triggered by `kblockd`'s wakeup. The plug/unplug mechanism is the main source of I/O scheduler overhead since it accompanies OS delay due to process scheduling. For this reason, many previous works [1,3,9,12] tried to bypass I/O scheduler instead of trying to benefit from it.

Low Device and Channel Utilization: When a flash-based SSD receives a single large I/O request, it splits the request into smaller ones and stripes them to multiple flash chips for maximizing parallelism [8]. However, the benefit is exploited only by a large-sized request; if discontiguous small requests are dispatched to a storage device one by one, the concurrent access to flash chips would hardly occur, lowering overall device utilization. The small data transfer in an I/O operation is also harmful to channel utilization. As shown in Figure 1, the smaller size of an I/O request leads to lower I/O throughput.

2.2 The Limitations of *Command Queueing*

Command queueing technology enables an OS to dispatch a next I/O request to a storage device before the completion of previous ones, increasing the number of concurrent I/O requests inside a disk command queue. It is known that a flash-based SSD implementing SATA-2 protocol uses Native Command Queueing (NCQ) [4] to parallelize I/O accesses or to pipeline micro-operations [8].

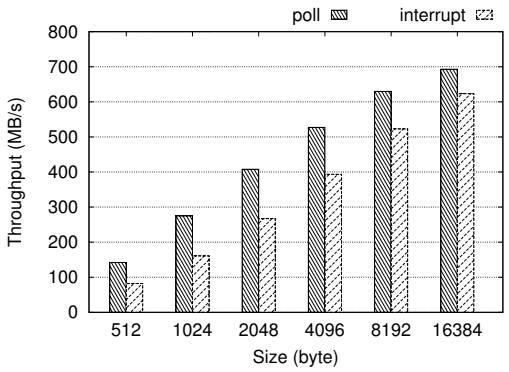


Figure 1: The influence of data transfer size on device and channel utilization

Hardware-latency that occurs from an intermediate controller, however, is considerable. Advanced Host Controller Interface (AHCI) intentionally puts off notifying an OS of each I/O completion in order to aggregate interrupts [7]. According to our preliminary evaluation, AHCI incurs the quite high waiting delay ranging from a few tens of microseconds to hundreds of microseconds.

3 Extending Block I/O Interface

The current block I/O interface supports N:1 scatter-gather operation that transfers data between discontiguous host memory segments and contiguous storage address space. The mapping information is contained in a `bio` structure in case of Linux, and utilized for setting up a DMA operation.

We propose an extension of the block I/O interface, called *device-level scatter-gather interface* that transfers data from discontiguous host memory segments to discontiguous storage address space, and vice versa. Multiple `bio` structures are packed into one I/O request, representing N:N mapping, and dispatched together. NVMHCI (or NVM Express) [6] currently doesn't cover this type of interface yet.

The new interface is directly implemented into our target DRAM-based SSD [11]. The SSD is connected to a host via a PCI-E channel and has a separate DMA engine that can scatter-to or gather-from discontiguous sectors by using a list of request descriptors. *Block Control Table* (BCT) supports 1,024 block requests at maximum, each of which is encoded as host memory segment, storage segment and data size. The kernel memory region of BCT is allocated as a consistent DMA region.

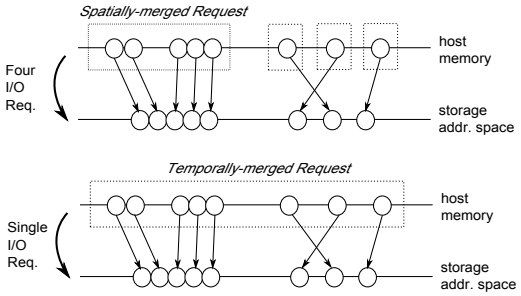


Figure 2: The comparison between spatial merge and temporal merge when given four block I/O requests

4 Merging Temporally-Adjacent Requests

By utilizing the extended block I/O interface, we design a new request batching scheme, called *temporal merge* that overcomes the disadvantages of spatial merge. The scheme combines multiple (even non-sequential) block requests arriving within a short time window into one I/O request, which is made possible by the new interface that relaxes the constraint of spatial-adjacency of requests. Figure 2 illustrates the concept of temporal merge and its use of the extended interface.

Temporal merge is implemented in I/O subsystem of Linux 2.6.32 in two different versions. The first is called Synchronous Temporal Merge (STM), which bypasses I/O scheduler and combines concurrent block requests only. On the contrary, Asynchronous Temporal Merge (ATM) actively makes use of I/O scheduler to pile up block requests and dispatch all of them using a single I/O operation. Each version has its own advantages over the other under a specific workload, which is demonstrated in the evaluation.

4.1 Synchronous Temporal Merge (STM)

Concurrency is defined as the number of CPU contexts (or threads) staying inside I/O subsystem; if N threads simultaneously invoke the 'read' system call, then the concurrency will be N or less since system calls from some of the contexts may not have reached the I/O subsystem yet. Among the concurrent threads, STM chooses one of them by using an atomic operation. It is called a *winning thread*. It gathers the block requests from the *losing threads*, and dispatches the temporally-merged I/O request to the storage device using the new interface. Temporal merge overcomes the disadvantages of spatial merge (as discussed in §2.1) because 1) a winning thread follows a synchronous I/O path requiring no plug/unplug (low-latency) and 2) the amount of data transfer is usually large regardless of I/O access pattern (high device and channel utilization). The overhead of sleep/wakeup latency of the losing threads could be masked by the benefit of the large data transfer of the winning thread.

4.2 Asynchronous Temporal Merge (ATM)

To benefit from temporal merge even when the number of the concurrent contexts is low (e.g., a single write-back thread), we devised a different type of temporal merge scheme, called ATM, that gives up synchronous I/O path, but instead aggressively piles up block requests by using I/O scheduler. ATM customizes the dispatch routine of I/O scheduler and fetches all the block requests from a request queue as long as BCT has any available slot. The merged I/O request is then sent to our DRAM-based SSD through the extended block I/O interface.

Unlike the previous works that mainly focused on bypassing I/O scheduler [1, 3, 9, 12], our approach actively utilizes its asynchronous I/O path to build a large I/O request regardless of the number of concurrent threads. The evaluation demonstrates that the benefit of the large data transfer dominates the disadvantages of the software overhead caused by I/O scheduler.

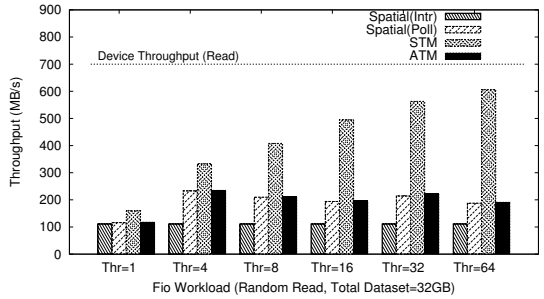
5 Evaluation

We used two micro-benchmarks, fio and iozone, to evaluate our schemes under various concurrency configurations. The target machine has two Xeon E5630 2.53 GHz quad core CPUs (total 8 cores) and 8 GB of RAM. It runs a Linux 2.6.32 vanilla kernel. The DRAM-based SSD [11] has 64 GB (=8 GB of DDR2*8) capacity and is controlled by FPGA firmware that implements the new block I/O interface. The measured peak device throughput is 700 MB/s for read and 650 MB/s for write. The response time of reading/writing a 4KB page is about 7 usecs and the per-request software overhead in I/O subsystem is about 4 usecs.

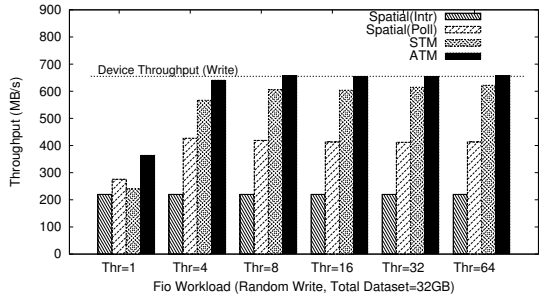
The baseline I/O subsystem, called **Spatial(Intr)**, performs spatial merge on incoming block requests, dispatches them and waits for notifications by interrupt. On the other hand, **Spatial(Poll)** uses poll instead of interrupt to eliminate context switch overhead. **ATM** adds temporal merge in **Spatial(Poll)**, while **STM** makes a detour around I/O scheduler to combine concurrent requests without plug/unplug mechanism. All of the four versions are implemented as loadable kernel modules, so do not require the kernel core to be modified at all.

5.1 Spatial vs. Temporal Merge

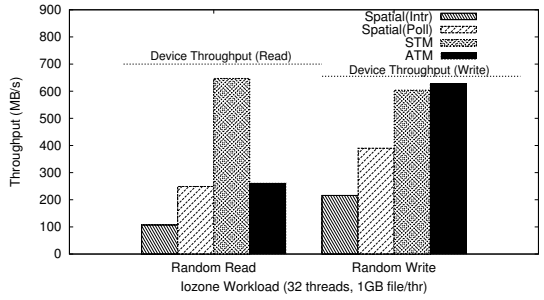
Random read performance: Figure 3(a) shows that **STM** performs 38% better than **Spatial(Intr, Poll)** and **ATM** under single-threaded workload because it does



(a) Fio (random read)



(b) Fio (random write)



(c) Iozone (random read/write)

Figure 3: Random throughput of micro-benchmarks

not suffer from software overhead inherent in I/O scheduler. As the concurrency becomes higher, **STM** combines more block requests and transfers more data in an I/O operation. The random I/O throughput eventually reaches 87% of the peak device throughput. On the other hand, the other two do not benefit much from the increased concurrency. They failed to pile up requests due to the critical section design of read I/O path in Linux; if a thread is polling on a completion, others cannot insert their requests into I/O scheduler due to the *queuelock* spinlock.

Random write performance: In Figure 3(b), **ATM** exploits 100% of the peak device throughput under random access workload even when the concurrency level is as low as 8. The result proves that the benefit of transferring data in large size dominates the harm of I/O scheduler overhead. Due to the constraint of spatial-adjacency, **Spatial(Intr, Poll)** always fail to combine block requests, showing worse performance than both **ATM** and **STM**.

A similar result is observed using `Iozone` benchmark as in Figure 3(c). The attained random read and write throughput are 92% and 93% of the peak device throughput respectively by **STM**, and 37% and 97% of that by **ATM**. **STM** is superior to **ATM** under a read-only workload, while **ATM** outperforms **STM** under a write-only workload. This result comes from the fact that Linux has different designs between read I/O path and write I/O path, giving us a hint to design a hybrid temporal merge scheme. We are investigating this as part of future work.

5.2 Effect of Concurrency

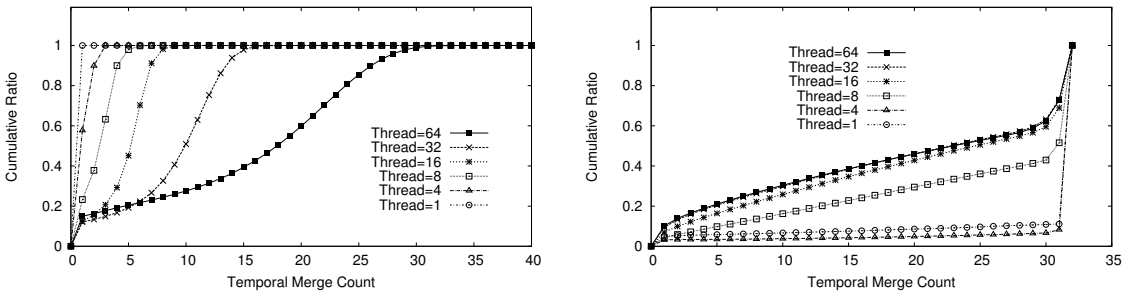
The concurrency significantly affects the distribution of *temporal merge count*, i.e. the number block requests in an I/O operation. Figure 4(a) shows that the higher concurrency results in the dispatch of a larger I/O request.

ATM effectively collects multiple (write) requests when the concurrency is not high, as shown in Figure 4(b). For example, when there is only one thread that submits write requests, the transfer size of 89% of requests is 128 KB containing 32 pages and contributes to the high device and channel utilization. Interestingly, the temporal merge count becomes lower when the concurrency is higher. The reason is that a user thread does not rely on page cache and synchronously dispatches a write request if the page cache is heavily pressured by write-intensive workloads. This causes a request queue to be unplugged prematurely before it reaches the threshold, which is `unplug_thresh` currently set to 32. throughput

5.3 Latency Breakdown

The timeline in Figure 5 shows the temporal merge behavior of STM for concurrent block requests. In the first phase, CPU #6 becomes a winner and gathers block requests from others, i.e. #0, #2, and #4. It takes 28 usecs for #6 from the start of dispatching the merged request and to the completion of the request. The POLLING period is 11 usecs for a page, 16 usecs for two pages, and 28 usecs for four pages. In case of **Spatial(Poll)**, the pure data transfer time of four pages is about 44 usecs (=11*4), which is 57% higher than STM.

Although STM decreases average latency of block requests, some of them should wait longer than they deserve due to large data transfer. For instance, the request issued by CPU #6 was expected to be serviced within 11 usecs if dispatched alone but took 28 usecs to be completed. The increased throughput comes at the cost of the increased latency of some requests.



(a) random read (STM) (b) random write (ATM)

Figure 4: The cumulative distribution of merge count under Fio workload with 4 KB random read/write

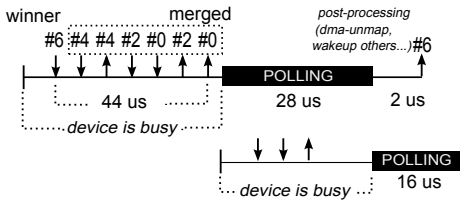


Figure 5: The example of latency breakdown when performing STM on four block requests

6 Discussion and Future Work

Temporal merge utilizes an extended block I/O interface to make random throughput close to peak device throughput, given a low-latency memory-based storage device. Unlike the previous researches that have focused mostly on bypassing several software layers to eliminate OS delay [1, 3, 9, 12], we have paid much attention to redesign the existing layers to make the best use of a low-latency device. There are still some issues to be explored to utilize our technique for commercial use.

Reliability Problem: Partial updates among multiple writes due to crash failure may lead to irrecoverable corruption to the file system state since a device may re-schedule the service order of requests and not preserve OS policy. One possible solution is to implement an 'atomic update interface' that guarantees all-or-nothing semantics; by shadowing the destinations of write requests and manipulating logical-to-physical mapping state, I/O subsystem would provide atomicity for multiple write requests.

Standardization Issue: A well-designed interface between an OS and a storage device is very important since 1) it is critical to the I/O performance experienced by an OS and 2) once fixed, it is hard to change for generations, which we have already learned from the experience [5]. The extended block I/O interface, i.e. device-level scatter-gather I/O, gives a chance for an OS to exploit the maximum throughput from low-latency memory-based storage devices. We suggest that a next-generation host controller interface, e.g. NVMHCI [6], should include this kind of functionality into its design.

Acknowledgements

We would like to thank the anonymous reviewers and Ajay Gulati, our shepherd, for their valuable feedback on our work. This work was supported by the Technology Innovation Program (Industrial Strategic technology development program, 10039163) funded by the Ministry of Knowledge Economy (MKE, Korea).

References

[1] AMEEN, A., ET AL. Onyx: A protoype phase-change memory storage array. In *HotStorage'11* (2011), pp. 1–5.

[2] CAULFIELD, A. M., ET AL. Providing safe, user space access to fast, solid state disks. In *ASPLOS'12*, ACM.

[3] CAULFIELD, A. M., ET AL. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO'10* (2010), pp. 385–395.

[4] DEES, B. Native command queuing - advanced performance in desktop storage. *Potentials, IEEE 24*, 4 (oct.-nov. 2005), 4 – 7.

[5] GANGER, G. R. Blurring the line between oses and storage devices, 2001.

[6] HUFFMAN, A. Nvm express revision 1.0c. Tech. rep., Intel Corporation, 2012.

[7] INTEL, AND SEAGATE. Serial ata native command queueing. Joint whitepaper, Intel Corp. and Seagate Technology, 2003.

[8] NAM, E. H., ET AL. Ozone (o3): An out-of-order flash memory controller architecture. *Computers, IEEE Transactions on 60*, 5 (may 2011), 653 –666.

[9] SEPPANEN, E., ET AL. High performance solid state storage under linux. In *Mass Storage Systems and Technologies (MSST'10)*.

[10] SHIN, D. I., ET AL. Request bridging and interleaving: Improving the performance of small synchronous updates under seek-optimizing disk subsystems. *ACM Trans. on Storage* (July 2011).

[11] TAILWINDSTORAGE. Extreme 3804, http://tailwindstorage.com /products/.

[12] YANG, J., ET AL. When poll is better than interrupt. In *FAST'12*.