Open access • Journal Article • DOI:10.1145/2619092

Optimizing the Block I/O Subsystem for Fast Storage Devices — Source link

Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song ...+4 more authors

Institutions: Seoul National University

Published on: 01 Jun 2014 - ACM Transactions on Computer Systems (ACM)

Topics: Paging, Linux kernel, Context switch and Performance improvement

공학박사학위논문

고성능 저장 장치를 위한 블럭 입출력 서브시스템 최적화

Optimizing Block I/O Subsystem for Fast Storage Devices

2012년 7월

서울대학교 대학원
전기컴퓨터공학부
유영진

Abstract

Recent development of storage devices has been driven by advanced memory technology, which shifts the paradigm of data access mechanism from magnetics and mechanics to electronics. As a result, the latency of Solid State Drives (SSD) is cut down to order of microseconds. In spite of the emergence of fast storage devices, however, the existing storage stack cannot keep pace with the speed of the new device since the software has been too optimized to a slow disk for decades; storage vendors like Fusion-IO and OCZ started to implement their own fast storage stack that would maximize the benefit of their products. Storage systems are now facing an important challenge exploiting the low-latency characteristics of these devices.

In this paper, we propose six types of block I/O subsystem that exploit the performance feature of an SSD with ultra low latency. Our optimization principles are 1) minimizing per-request overhead by redesigning I/O path, and 2) mitigating per-request overhead by using a request batching scheme. The techniques of *device polling* and *synchronous I/O path* belong to the first principle, while *dispatching discontiguous block requests in a single I/O operation* belongs to the second one. Unlike the previous works that mostly focus on the elimination of some software layers, we actively optimize the existing OS components and implement new functionalities to achieve both low latency and high throughput. We demonstrate the effectiveness of our designs with actual prototypes that operate in recent Linux kernel 2.6.32. Evaluation results show that synchronous I/O path (SyncPath) attains 3.3x reduction in software-latency under single-threaded workload and a new merge scheme with double buffering (2Q) leads to 4.4x improvement in throughput under multi-threaded workload, compared to the Linux block I/O subsystem. In addition, the hybrid I/O path design (HTM) delivers 87%∼100% of the performance of low-latency devices to an application regardless of request access patterns or request types. We believe that our block I/O subsystem designs are general enough to be implemented for next-generation SSDs pursuing near-DRAM latency and throughput.

Keywords : I/O subsystem, Storage device, Latency, Throughput

Student Number : 2006-21228

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Improving I/O performance has always been one of the most important challenges in optimizing computer system. Since mechanism of accessing data in a storage device heavily affects the performance of applications, researchers have sought the ways of reducing time to access data by developing new hardware and software technologies.

An I/O subsystem [1, 2] is a part of storage stack in an OS and handles I/O requests to serve file system or a user process. It reflects the underlying hardware features of a storage device into its design to make the best I/O performance with the device. For decades, this I/O subsystem has been optimized for magenetic hard disk drives. Considering the mechanism of a disk, the I/O subsystem tries 1) to maximize the amount of data transfer in a single I/O operation by merging contiguous requests [3], and 2) to minimize the seek movement of a disk head by reordering the requests [4]. After dispatching an I/O request to a disk, the I/O subsystem yields CPU resource for other useful jobs and waits for *interrupt* notification of the request completion.

Nowaday, fast storage devices have been emerging into the market. As the paradigm of data access mechanism moves toward electronics from magnetics and mechanics, the new storage devices achieve the ultra-low latency of a few microseconds and orders of magnitude higher throughput than a disk can provide. Flash memory contributes to the success of the memory-based storage devices in the market [5, 6], and the next-generation memory, called *Storage Class Memory*, is expected to keep the success for a while by providing near-DRAM performance and non-volatility [7].

Unfortunately, the ordinary practice to put new storage devices into an existing I/O subsystem limits the I/O performance of storage system, which is a well-announced problem in previous studies; even recent versions of Linux I/O subsystem have difficulty in exploiting the full performance of a fast SSD. Without redesigning many OS components as a whole, storage system cannot realize the true potential of the device. As a result, Moneta [8] and Onyx [9] strive to modify software stack in OS, and provide an optimized hardware interface for their *Phase-Change-Memory* [10] based SSD. Fusion-IO [11] is also known to distribute a customized I/O subsystem [12] for their high-performance SSD products to deliver the maximum throughput to user applications.

Hence, a modern storage system is now facing an important challenge of exploiting the performance of fast storage devices. Although the characteristics of the underlying hardware are changing rapidly, existing I/O subsystems are unaware of them and still has a specific design optimized for a disk. In this dissertation, we will investigate the design problems inherent in the existing I/O subsystems and explore new designs to maximize the benefits of a memory-based storage device with ultra-low latency.

1.1 Motivation: Slow Software on Fast Hardware

For clarity, we define a few terms regarding the performance of storage system.

- **Device throughput** is the maximum possible throughput measured inside a storage device. It is determined by hardware technology such as the storage medium and the device architecture.

- **Application throughput** is the throughput measured by a user-level application. We assumed that the size of working set is much larger than physical memory and an application cannot benefit from *memory cache hit*. Based on the assumption, application throughput is limited by device throughput; the former is always less than the latter. We further define **sequential throughput** and **random throughput** which correspond to the observed application throughputs under sequential access workload and random access workload, respectively.

- **Hardware latency** is the response time of a storage device, i.e. the time between the dispatch of an I/O request and the completion of it. A disk has non-uniform hardware latency due to its internal state like the current position of a disk head and the logical-to-physical mapping [13].

- **Software latency** is the sum of 1) the time taken before the dispatch of an I/O request, and 2) the time taken after the completion of it. It represents *software overhead* in storage stack in an OS, including overheads of merging requests, copying data into/from DMA buffers, setting up a DMA controller, waking up the processes that issues the requests and etc.

The two metrics, latency and throughput, are widely used to evaluate the I/O performance of storage system [14]. Latency is the sum of hardware latency and software latency and throughput is *data transfer rate (bytes/sec)*. The reduced latency usually contributes to the increased throughput; if the latency of a request is minimized, the random throughput would be maximized. On the other hand, the sequential throughput may not be directly affected by per-request latency. Rather, certain optimization techniques such as request merging enhances the sequential throughput at the cost of the increased latencies of some requests.

Based on the intuition about the relationship between the two performance metrics, the existing I/O subsystem pursues a balance between the low-latency and the high-throughput. If the high random throughput is favored, the software latency of each I/O request should be reduced by optimizing the *I/O path* in the I/O subsystem. For the case of enhancing the sequential throughput, the techniques of batching multiple requests and mitigating the per-request software latency are the effective solution.

To understand the performance effect of the disk-based I/O subsystem on a low-latency storage device, we choose to use the recent Linux I/O subsystem and the DRAM-SSD [15]. The hardware latency of the SSD to access a 4KB page is 7 usecs and the device throughput is about 700 MB/s according to the vendor's information. Since the performance goal of the

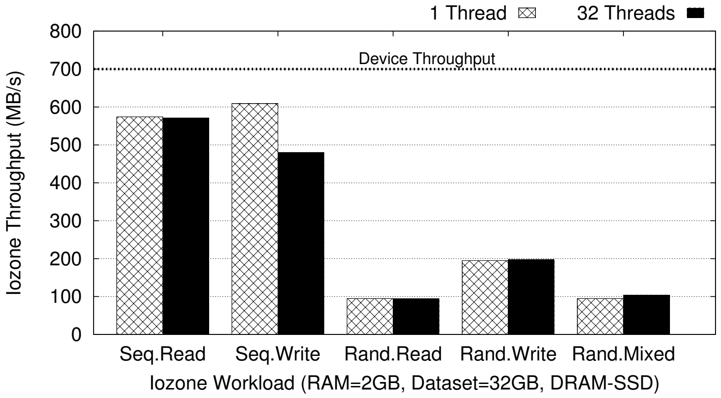


Figure 1: Baseline I/O subsystem evaluation with Iozone

next-generation memory such as Phase-Change Memory [10], FeRAM [7], STT-RAM [16] and MRAM is to achieve near-DRAM latency and throughput, we believe that our observations will be effective even in the near future without loss of generality.

Figure 1 reports the application throughput of Iozone [17]. Comparing the application throughput to the device throughput under the different workloads, we found two obvious performance-gap problems, which are clarified by the following problem definitions:

P1. The application throughput is lower than the device throughput.

P2. The random throughput is lower than the sequential throughput.

When the hardware latency dominated the latency of an I/O request, the software latency was insignificant and it was relatively easy to deliver the device throughput to an application. For example, the device throughput of a disk is around 100 MB/s and an application is able to exploit the full performance if it accesses I/O requests contiguously in increasing order. However, when it comes to a low-latency storage device, even a small software delay can degrade I/O performance. To solve **P1**, the software overhead caused by each layer in the storage stack in an OS should be deeply examined.

P2 is natural for a disk having mechanical moving parts, but unfair for an SSD consisting of multiple memory chips; at least in the hardware perspective, a single large request that stripes to multiple chips inside a device is no different than a bunch of small requests, each of which targets a different chip. To solve **P2**, we should attack this point and rethink the way of dispatching multiple block requests both at hardware and software level.

Our final goal is to design a high-performance I/O subsystem that exploits the full performance of a low-latency storage device regardless of request access patterns of an application (sequential or random) or request types (read or write).

1.2 Contributions

The contributions are summarized as follows:

- We investigated the software latency of an I/O request in the recent Linux I/O subsystem to contrast it with the hardware latency of a low-latency storage device

- We designed a new block I/O interface to dispatch multiple discontiguous block requests in an I/O request

- We designed six types of I/O subsystems to deeply understand the interaction between an I/O subsystem and a low-latency storage device, and to find the most efficient one for next-generation fast storage devices

One of our optimization techniques, called *temporal merge*, essentially needs hardware modifications to utilize a customized interface beyond the standard. Although it is known to be hard to reach a consensus between OS communities and storage vendors [18], 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 [19]. All of our work have targeted I/O subsystems for the DRAM-SSD, but we believe that the design rationales or discussions will still be useful for other storage devices with different performance features.

1.3 Overview

This paper is organized as follows:

- **Background:** The chapter 2 covers the basic explanation about the I/O path in Linux I/O subsystem. It also describes the recent trends in storage technology that make our solution feasible.

- **Motivation:** The chapter 3 analyzes the four factors that prevent the I/O subsystem from achieving the device throughput of a low-latency storage device. The detailed description will be based on the two problem definitions P1 and P2.

- **Solution:** The chapter 4 proposes various I/O subsystem designs, which are improved step by step by reflecting the hardware features and the interaction between OS components and an I/O subsystem. In the chapter 5, the implementation issues specific to the Linux kernel development are given.

- **Evaluation:** The different I/O subsystem designs are evaluated in the chapter 6, which include varying concurrency and request access patterns.

- **Conclusion:** We summarize the key points of our I/O subsystem designs and their contributions in the chapter 8. Finally, we discuss some issues about our solution to be used for commercial use.

Chapter 2

Background

In this chapter, we will look into the recent trends in storage technology which makes it possible for our solution to be more practical. Then, the detailed description about the architecture of the existing I/O subsystem will be presented for further discussion in the later chapters.

2.1 Trends in Storage Technology

Recent development of storage devices have been driven by advanced hardware technology, which shifts the paradigm of data access mechanism from magnetics and mechanics to electronics. As a result, the I/O performance of recent SSDs is an order of magnitude higher than that of traditional disks in terms of latency and throughput. Two critical technologies have enabled the dramatic progress in a storage device:

- **New Storage Medium:** Storage medium is the physical material for storing bit data. It determines the data access mechanism. In case of a HDD, magnetic platter is used for storing data and requires a mechanical moving head to sense magnetic flux over the surface of the platter, which takes a few milliseconds even in the best case. On the contrary, accessing flash memory can be done by transistor technology without mechanical overhead. The academic society is now actively studying

Storage Class Memory [20] to use it either as a storage device [21] or as main memory in a computer system [22, 23, 24, 25].

- **Efficient Device-Architecture:** To benefit from the parallelism of the multiple units of storage medium, recent SSDs have an efficient architecture to pipeline successive I/O requests, to concurrently stripe non-conflicting requests to multiple destinations [26, 27, 28], to service requests out-of-order [29] and etc. While the storage medium affects the hardware latency, the architecture of a storage device influences the device throughput of a storage device. It is known that a Flash-based SSD manufactured by Fusion-IO [30] also uses many flash chips to maximize the device throughput.

Along with the development of storage devices, computer systems are experiencing radical progress in hardware components to deal with storage devices; the host bus interface is moving towards faster ones, e.g. from SATA-2 (3 Gb/s) to SATA-3 (6 Gb/s) or PCI-Express (4 GB/s per direction). Additionally, the advent of many-core processors is especially a momentous change in storage system environment since the benefit of dedicating certain cores to I/O processing would come at reasonable cost, which was considered harmful in the past.

Recently, high processing-power and high IO-performance became *mutually essential* to each other. If the processing power is low, the utilization of the underlying storage device would also be low, making it hard to attain high I/O performance. On the other hand, if the performance of the storage device is not high enough, each CPU would race against others (e.g. due to polling [8]) to perform I/O operations on the shared device, and result in the collapse of the scalability in many-core system. Fusion-IO also states that the CPU cost for I/O operations is no longer cheap and affects the application's performance [11].

2.2 Analysis of I/O Path

An *I/O request* contains all the necessary information to perform an I/O operation on the underlying storage device, such as the sector address, a list of host memory segments, transfer size and etc. The sequence of software layers that I/O request goes through is defined as the *I/O path*. We can classify it into two types depending on the direction of I/O requests; 1) the *descending I/O path* reaches the storage device from an application, and 2) the *ascending I/O path* does the opposite, i.e. from the storage device to an application.

In the context of Linux storage stack, the I/O subsystem is composed of Block Layer, SCSI Subsystem, and Device Driver. Figure 2 illustrates the common I/O path in the I/O subsystem. When the **Virtual File System (VFS)** cannot find a requested page from page cache (*read miss*), or run short of available pages for buffering writes (*write miss*), it builds an I/O request filled with a target storage address retrieved by the **File System**, and initiates the descending I/O path by submitting the request to the **Block Layer**. This layer performs two important block-level optimizations, *I/O scheduling* and *request merging*. Those techniques are effective only when there are multiple requests in a request queue; if I/O scheduler has more

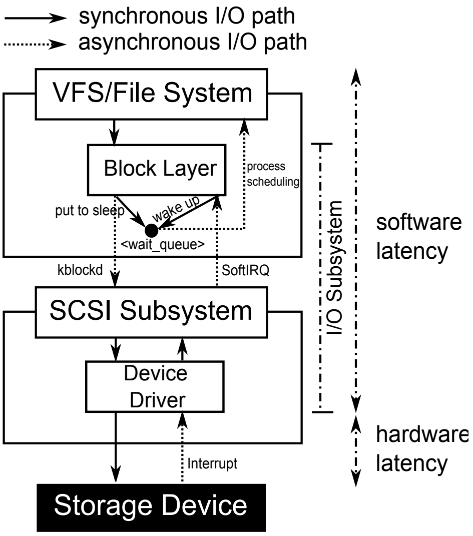


Figure 2: Common I/O path in Linux storage stack

candidates, the possibility of generating more efficient schedules would be increased. For this purpose, the descending I/O path is usually suspended at this moment by *plugging* mechanism.

The rest of the descending I/O path is resumed either 1) by a kernel thread called `kblockd`, or 2) by a conditional trigger based on the state of a request queue such as the number of I/O requests in the queue and the time passed since the last dispatch of a request. This event is called *unplugging* and one of I/O requests in the request queue is dispatched to the underlying **SCSI Subsystem**. The layer prepares DMA buffer, sets up scatter-gather entries, and initiates DMA transfer by invoking the callback routine registered by the **Device Driver**.

The storage device issues an interrupt signal to notify the **Device Driver** of the completion of an I/O request. After freeing resources allocated by the

SCSI subsystem, the Device Driver puts off further completion handling, anticipating that the remaining work can be done by SoftIRQ handler on the original CPU that submitted the I/O request. Finally, the handler wakes up the user process waiting for the pages in the completed I/O request. When the OS schedules the awaken process, it starts to copy the requested pages into user space and finishes the ascending I/O path.

2.3 Optimization Techniques by I/O Subsystem

Generally, there are two approaches to improve the performance of I/O subsystem. They are 1) reducing per-request latency and 2) hiding (or mitigating) per-request latency by batching multiple requests in an I/O operation. The first one has to be solved by re-designing the I/O path in the I/O subsystem. The other one involves the request batching scheme (1) at software-level like request merge by Block Layer, or (2) at hardware-level like *command queueing* [31] which accumulates multiple requests inside a device and mitigates per-request protocol overhead.

- **Reducing per-request latency:** I/O scheduler, contained in the Block Layer, estimates the hardware latency of an I/O request by calculating the seek distance when the request is dispatched, which is called *seek-optimization heuristics*. The Linux I/O subsystem has three I/O schedulers [32], CFQ, Anticipatory [33] and Deadline, mainly based on one-way elevator algorithm. NCQ [31, 34] is regarded as a hardware-level I/O scheduler; it overcomes the inability of rotational-latency-sensitive scheduling at software-level [35] and reschedules multiple commands by considering the exact position of a disk head with the help of the firmware.

- **Mitigating per-request latency:** The technique of merging spatially adjacent I/O requests has been one of the most successful optimizations in handling a storage device [3]. We call this technique *spatial merge* in further explanation. It helps the OS to get the maximum throughput from a storage device by 1) mitigating mechanical overhead in case of a disk, e.g. seek-time and rotational-delay, and 2) accessing multiple memory chips in parallel in case of a memory-based SSD. A single large request produced by this technique always enhances the utilization of a storage device, thus increasing application throughput.

Chapter 3

Analyzing the Legacy of Disk-based I/O Subsystem

In this chapter, we deeply investigate the effect of using the existing I/O subsystem on a low-latency storage device. A reasonable conclusion is drawn from the analysis that the current I/O subsystem cannot fully exploit the performance of a low-latency storage device due to several reasons, which are 1) *interrupt overhead* that emerges as the hardware latency of a storage device becomes as low as a few microseconds, 2) *delayed execution* that incurs the overhead of context switch and process scheduling, 3) *request merging* based only on spatial-adjacency, which is too restrictive for SSDs with no moving parts, 4) *disk-oriented configuration* that favors only seek-optimization which may not be useful for storage devices other than disks.

The two performance gap problems, P1 and P2 has been described in §1.1. In the following sections, we examine how each of the four factors contribute to the problems when a low-latency storage device is used with the existing disk-based I/O subsystem.

3.1 Problem 1: High Software Latency

As shown in Figure 1, the Linux I/O subsystem incurs performance degradation when delivering the device throughput to an application. We have performed latency analysis to examine whether any software overhead exists in the I/O path.

3.1.1 Interrupt Latency

To analyze the interrupt overhead, we split the hardware latency of our DRAM-SSD into several components as shown in Figure 3. The time spent in each phase is measured by a logic analyzer. *Interrupt latency* is defined as the time between the generation of an interrupt by a device and the servicing of the interrupt. The measurement shows that the interrupt latency for an I/O request approaches 2∼3 usecs. This overhead consumes 25% of the hardware latency for 4 KB access and more than 40% for 512 bytes and 1 KB access.

Consequently, we revisit the problem of interrupts, which has been a classic topic in network stack design [36, 37, 38, 39, 40], but this time in the Linux I/O subsystem. Traditionally, the I/O subsystems in modern OSes have used interrupt to confirm the completion of an I/O request. Since the interrupt latency is just 0.1% of the hardware latency of a disk, interrupt mechanism has been regarded as the low-cost notification method. However, this is not true anymore. The interrupt latency of 2∼3 usecs now becomes significant enough to affect the application throughput when we use a low-latency storage device that responds to OS within a few microseconds. Now

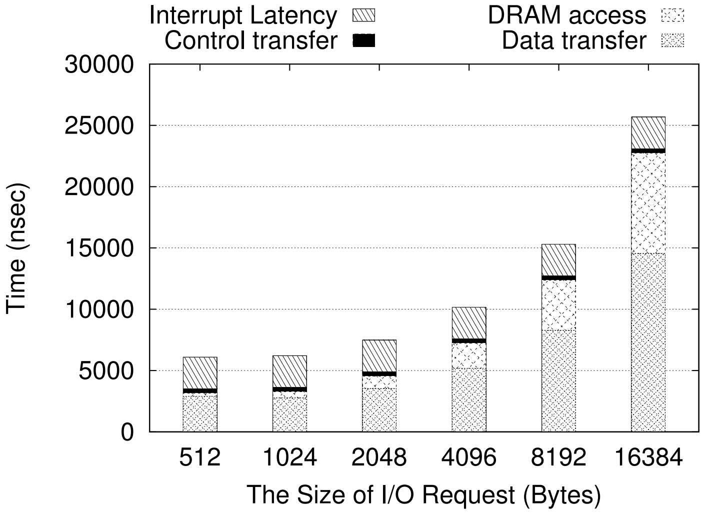


Figure 3: Hardware-latency breakdown of DRAM-SSD

we have to question the old custom of using interrupt notification for storage devices.

3.1.2 Delayed Execution

Using the kernel-level profiling tool `kprobe`, we perform latency analysis of the I/O path and show the timeline of an I/O request passing through each software layer. At any point in the I/O path, CPU is in one of the four different contexts; 1) a user process, 2) a kernel thread (`kblockd`), 3) a SoftIRQ handler, or 4) an interrupt handler. When the CPU context is switched from one to another, the I/O path is suspended and resumed. We define such property *asynchronous*, and such asynchronous points in the I/O path, *asynchrony*. Asynchrony due to kernel preemption is not included in our discussion. Linux I/O subsystem is composed of BLK (block layer), SCSI (SCSI subsystem), and DEV (device driver). To simplify the measurement of per-request latency breakdown, we used a single-threaded workload {dd, 1 thr, seq(read), direct I/O} where the next I/O request enters I/O subsystem only when the previous one is completed. R represents the hardware latency of reading a 4KB page from a storage device. In case of DRAM-SSD, R is 7 usec.

Table 1 shows that the software latency (53 usecs) is much higher than the hardware latency (7 usec) and also show the performance bottleneck is not in hardware anymore; now the bottleneck is in software. Major software latencies stem from the asynchronous I/O path design. It took 9 usecs for the CPU context to switch from the interrupt handler to the SoftIRQ handler, and 4 usecs from the SoftIRQ handler to the user process. The I/O path in the Linux storage stack contains 3∼4 asynchrony in a typical case, each of which essentially leads to a context switch.

There are two reasons why Linux I/O subsystem depends on the SoftIRQ handler to perform post-processing on the completed I/O requests:

- **Cache-friendly request retirement:** The interrupt handler has a belief that the completed I/O request may exist in the L1/L2 cache of the CPU that originally submitted the request. Expecting the post-processing of the completed request would benefit the CPU cache, the interrupt handler puts off further *request retirement* [41].

- **Limitation of top-half design:** Linux and most OSes, have a model of two split handlers, called top-half and bottom-half, to deal with interrupt. The OS usually puts some timing constraints on the execution

Layer	Context	Function	In/Out	Time
VFS	dd	do_sync_read	in	0
BLK	dd	generic_make_request	in	4
BLK	dd	generic_make_request	out	7
SCSI	dd	scsi_request_fn	in	9
SCSI	dd	scsi_request_fn	out	16
VFS	dd	io_schedule	in	18
DEV	interrupt	SSD_intr	in	R+18
DEV	interrupt	SSD_intr	out	R+27
BLK	softirq	blk_done_softirq	in	R+36
BLK	softirq	bio_endio	in	R+39
BLK	softirq	bio_endio	out	R+41
SCSI	softirq	scsi_run_queue	in	R+45
SCSI	softirq	scsi_run_queue	out	R+46
BLK	softirq	blk_done_softirq	out	R+47
VFS	dd	io_schedule	out	R+51
VFS	dd	do_sync_read	out	R+53
VFS	dd	do_sync_read	in	R+56

Table 1: Latency-breakdown of the common-case I/O path

of top-half such as *the top half should not follow any slow path that makes the current CPU context schedule out*. If the execution of the slow path is unavoidable, the remaining work must be delivered to the bottom half through the asynchoronous I/O path.

As the Linux I/O subsystem relies on a few event notification channels, e.g. reserving the invocation of SoftIRQ handler, or adding a work to the queues of other kernel threads, the software delay of a few usecs is inevitable. These overheads have been insignificant when the hardware latency is high enough, but become noticeable with the emergence of low-latency storage devices. Therefore, we should consider whether the asynchronous I/O path design is suitable for high-performance I/O subsystems.

3.2 Problem 2: Low Random Throughput

Figure 1 indicates that the random throughput achieved by the existing I/O subsystem is only about 25% of the sequential throughput. A similar result is also observed in a heavily-loaded scenario where 32 threads are concurrently requesting data access. Without batching multiple requests, the random throughput would hardly attain higher performance than the current version. In this section, we investigate the limitations of the request merging scheme in the Linux I/O subsystem and point out some configuration issues that make it hard for the I/O scheduler to accumulate many I/O requests in a request queue.

3.2.1 Narrow Block I/O Interface

Spatial merge builds a single large I/O request from multiple contiguous requests, and achieves 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 a request queue to prevent I/O requests from being sent to a storage device, which means plugging. From this point on, each I/O request is enqueued into the request queue and tested whether it is spatially-adjacent 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 [41, 8, 9, 42] tried to bypass I/O scheduler instead of trying to benefit from it.

- **Low Device-/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 4, smaller size of an I/O request leads to lower I/O throughput.

The limitations of spatial merge originate from the narrow block I/O interface that supports the dispatch of an I/O request that only have contiguous block addresses. This interface seems reasonable when we consider the mechanism of a disk because non-adjacent block requests can never be serviced concurrently. On the contrary, recent SSDs already have an architecture that services multiple I/O requests in parallel (as discussed in §2.1). The current block I/O interface is too restrictive for those SSDs since it limits the benefit of the parallelism inside a device. The narrow interface enhances the device-/channel- utilization only when a single large request is dispatched, which is not achievable under random access workload because

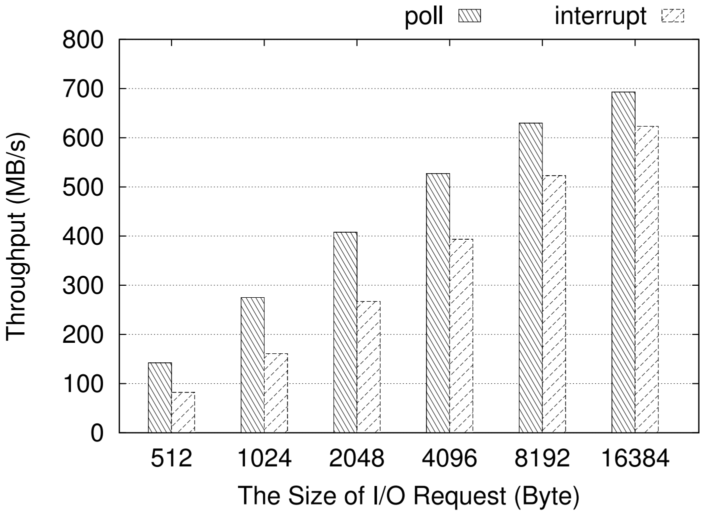


Figure 4: Effect of data transfer size on device-/channel- utilization

of the failure of spatial merge.

The plug/unplug mechanism to check spatial-adjacency incurs significant software overhead; if we can relax the constraint of spatial-adjacency and instead utilize only a very small time window to combine requests, the software latency taken by request merging scheme would be reduced.

3.2.2 Disk-oriented Configuration of I/O Subsystem

I/O scheduler maintains the number of I/O requests to a certain level in a request queue to apply spatial merge or schedule algorithm on them. To prevent some requests from being starved, the I/O scheduler forces the request queue to be unplugged to flush the requests in the queue if some conditions are met. The threshold value, called unplug_thresh, limits the maximum number of block requests in a request queue. It is one of the configuration of a request queue as explained in Table 2, and is compared against the current number of block requests that are not dispatched yet, whenever a new request is inserted. If there are more requests in a queue than the threshold, the CPU context that is about to insert the request synchronously performs the unplug routine.

However, the existing I/O subsystem is configured based only on the characteristics of a disk. For the following reasons, the I/O scheduler (contained in the I/O subsystem) is hardly able to pile up many block requests:

- **Threshold for slow storage devices:** The default unplug_thresh is set to 4, which triggers the unplug event very frequently when used on low-latency storage devices. Unlike HDDs, low-latency SSDs service an I/O request within a few microseconds, reducing the chances of accumulating successive block requests.

- **Fairness policy of I/O scheduler:** In case of CFQ scheduler, the block requests in a request queue (in fact, multiple queues of CFQ) is often flushed before the number of block requests reach the unplug_thresh value; CFQ tries to guarantee fairness among the concurrent processes by forcing a specific request queue to be unplugged.

- **Favor of read requests:** A read request mostly results from a blocking system call by a user process, so it is favored by the Linux I/O subsystem by triggering the unplug event immediately after inserting the read request. Due to this, the I/O subsystem usually fails to accumulate many block requests in a request queue.

Parameter	Default	Description
I/O scheduler	CFQ	Complete Fair Queueing scheduler
unplug_delay	3 ms	Timer expiration value
unplug_thresh	4	The maximum limit of I/O requests in a queue
ra_pages	128	The maximum limit of pages for read-ahead

Table 2: Default configuration of a request queue

Additionally, some heuristics of an I/O scheduler produces unnecessary software delay. For example, Anticipatory scheduler waits for future requests to be spatially-adjacent to any in a request queue and leaves a storage device idle for a few milliseconds. Even if Noop scheduler is used, it is known that the code increases the I/O path and incurs a few usecs software overhead [41]. Some research [43, 2] point out that they are not suitable for recent SSDs that do not have any mechanical moving parts.

Chapter 4

Design Exploration of I/O Subsystem

In this section, we explore five I/O subsystem designs to fully understand the interaction between each I/O subsystem and low-latency storage devices in the hope of relaying the device throughput to an application without performance loss. Each design has some advantages over the others, or faces unexpected semantic gap between the I/O subsystem and the upper layer. Although there is no silver bullet that achieves the best latency and throughput in all workloads due to their tradeoff relationship [14], our design exploration would help us find the most efficient design for future low-latency SSDs even if their characteristics may vary from their origin. Our designs are mainly based on the following optimization techniques:

O1. Using poll instead of interrupt,

O2. Establishing synchronous I/O path,

O3. Merging and dispatching discontiguous requests with the help of an extended block I/O interface,

O4. Eliminating disk-assumptions in I/O scheduler

Each technique influences the I/O subsystem design to reduce unnecessary delay (by O1 and O2) or to hide per-request latency (by O3 and O4). Some of them are mutually exclusive (e.g. O2 and O4), and others can be combined (e.g. O1 and O3) to benefit from both approaches.

4.1 Baseline Design: Asynchronous I/O Path and Interrupt

Our baseline I/O subsystem, called SCSI_INTR, is registered to the SCSI subsystem as shown in Figure 2. It relies on asynchronous I/O path; a write request goes through the four context switches, which is *a user process → kblockd → interrupt → SoftIRQ handler → a user process*, and a read request follows the I/O path of *a user process → interrupt → SoftIRQ handler → a user process*. In the case of a read request, it is favored by the existing I/O subsystem, hence dispatched to a storage device synchronously in the context of a user process instead of kblockd (as discussed in §3.2.2).

4.2 Design 1: Making Entire I/O Path Synchronous

The first optimization was to change the detection method of I/O completions from interrupt to poll. In this design, the I/O subsystem, which we call ”SyncPath”, busily waits on a specific I/O port to check the completion of I/O requests. As soon as a completion is detected, SyncPath initiates a post handling routine without a context switch, which removes 1 asynchrony in the ascending I/O path.

Due to the poll mechanism, SyncPath does not have to split the post handling process into top-half (interrupt handler) and bottom-half (SoftIRQ handler). This is possible since the polling is performed in the context of a specific process, i.e. a user process or kblockd, and allows any slow path execution that happens to yield CPU resource to other processes. This further eliminates 1 asynchrony (*interrupt* → *SoftIRQ handler*) and reduces the software latency in the ascending path.

Bypassing the I/O scheduler is a well-known technique to reduce asynchrony in the descending I/O path [8, 41]. By registering a customized function to the block layer, I/O requests are directed to SyncPath instead of the I/O scheduler. Then, SyncPath dispatches them directly to the underlying storage device, not depending on a background kernel thread `kblockd`.

The positive effect of utilizing both the poll mechanism and the bypass of the I/O scheduler is that a user process is not necessarily put to sleep after submitting an I/O request; at the time when a user process returns to the VFS/FS layer after submitting an I/O request, the requested data already becomes ready, i.e. *page is up-to-date* in terms of Linux kernel. Consequently, SyncPath makes the I/O path fully synchronous without any context switches during the descending/ascending path. Figure 5 illustrates an example that four processes submit I/O requests and follow their I/O paths synchronously. Each symbol is explained in Table 3.

According to the result in Figure 6, SyncPath achieves the improvement of 1.58x∼2.47x under random access workload, and the improvement of 1.14x under sequential access workload over SCSI_INTR. Those improvements come from the reduced number of asynchrony in the I/O path and get higher as the per-request size is smaller; when sequential read workload is given, the read-ahead condition is met and a large-sized request (32 pages by default) is created. The request increases the hardware latency of

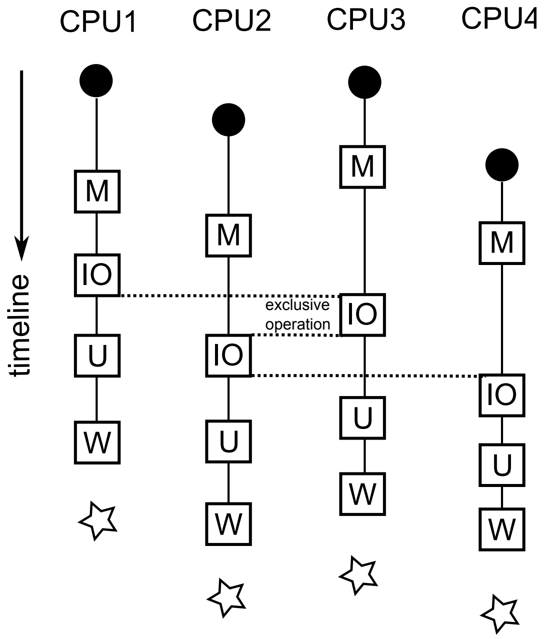


Figure 5: Fully-synchronous I/O path in SyncPath design

transferring data, while the software latency becomes relatively ignorable. Consequently, the benefit of optimizing the I/O path is reduced.

Interestingly, in the case of SyncPath, the sequential write throughput is close to the random write throughput. The reason is that the synchronous I/O path gives up the chance to combine successive write requests; we cannot anticipate any change of latency-hiding technique. Although SyncPath is more effective than SCSI_INTR in most workloads, the application throughput by SyncPath is still lower than the device throughput.

Symbol	Description
●	The current I/O request
☆	The next I/O request
M	DMA pre-processing (DMA-map)
IO	Initiate DMA and poll on the completion
U	DMA post-processing (DMA-unmap)
W	Wake up the process that requested I/O
B	Prepare DMA-able buffer (Bounce)

Table 3: Symbol description for I/O subsystem design

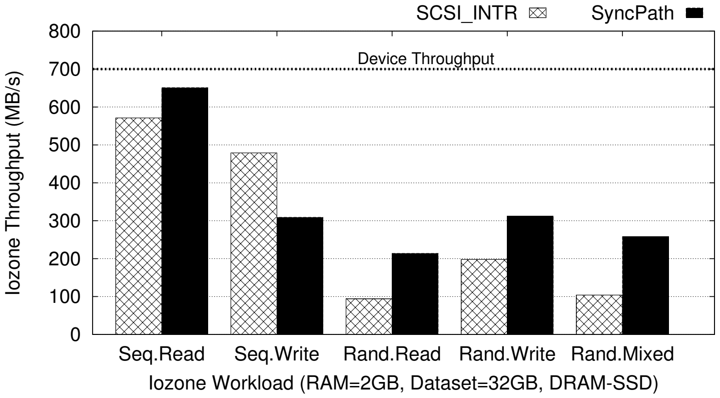


Figure 6: Iozone evaluation of SyncPath and SCSI_INTR

4.3 New I/O Interface: Dispatching Discontiguous Block Requests in a Single I/O Request

Spatial merge in the Linux I/O subsystem identifies spatial-adjacency (or spatial locality) from contiguous block requests. Sending a set of contiguous requests in an I/O request can actually be a good way to hide per-request latency. However, it should be questioned that such design based on spatial-adjacency is still the best choice for low-latency storage devices with different characteristics compared to disks.

Considering the two limitations of spatial merge technique (in §3.2.1), we devise a new batching scheme called *temporal merge* efficient for non-rotational and non-seeking memory-based storage devices. This mechanism is not based on spatial-adjacency but rather combines requests into one, based on temporal locality even if their addresses are not contiguous. Thus, the requests that reach I/O subsystem within a short time window can be sent to the underlying device together.

Of course, the mechanism requires hardware modification since the current DMA operations only allow mappings of data from discontiguous memory segments into a single contiguous storage address space. We implemented a *device-level scatter/gather operation* in the controller inside our target SSD and exposed an interface for the OS to issue temporally-merged I/O requests. This new functionality is illustrated in Figure 7. It plays a key role of increasing random throughput close to sequential throughput in the subsequent I/O subsystem designs.

4.4 Design 2: Merging Discontiguous Block Requests Synchronously

The second design of our I/O subsystem is called STM (Synchronous Temporal Merge), and implements temporal merge by using the extended block I/O interface mentioned in the previous section. We call the property of STM 'synchronous' since the technique combines multiple block requests without relying on plug/unplug mechanism; as long as there is any

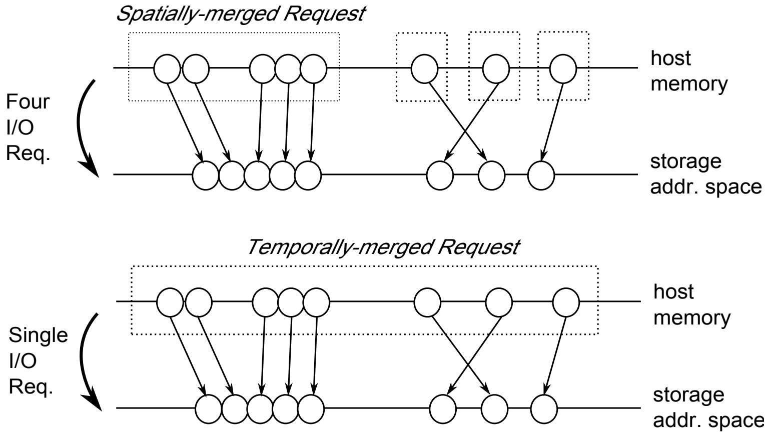


Figure 7: Comparison between spatial merge and temporal merge

request inside this I/O subsystem, at least one CPU context will be spinning until the storage device becomes idle. Unlike the case in SyncPath, not all threads follow the whole I/O path. Instead, only one thread (a winning thread) gathers block requests from other threads (losing threads) and execute the synchronous I/O path on behalf of others.

Figure 8 describes a situation where four threads are concurrently submitting block requests to STM. STM chooses only one thread (called *winner*) among the concurrent ones by using an atomic operation, and makes the thread follow the descending/ascending I/O path on behalf of others (called *losers*); the losing threads yield their CPU resource for other useful jobs and are put to sleep. The winner builds a temporally-merged request, maps buffers into DMAable region, and initiates DMA transfer. After detecting the request completion, the thread unmaps the DMA buffers and wakes up the losing threads.

STM overcomes the two limitations of spatial merge discussed in §3.2.1.

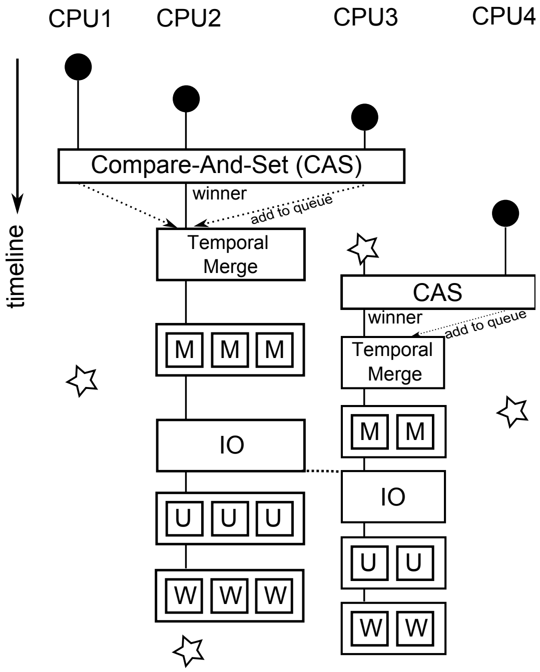


Figure 8: Synchronous temporal merge in STM design

First, there is no process scheduling overhead during the merge operation. The winner does not rely on plug/unplug mechanism and follows synchronous I/O path, which minimizes the software latency. Second, temporal merge always succeeds in building a large I/O request as long as the request queue has enough block requests, enhancing device-/channel- utilization. The technique makes it possible to raise the random throughput to the sequential throughput, which is not achievable by spatial merge with traditional block I/O interface.

Figure 9 shows that the performance improvement of STM over SyncPath is about 48∼150%. The sequential write throughput of STM is 93% higher than that of SyncPath and 24% higher than that of SCSI_INTR that

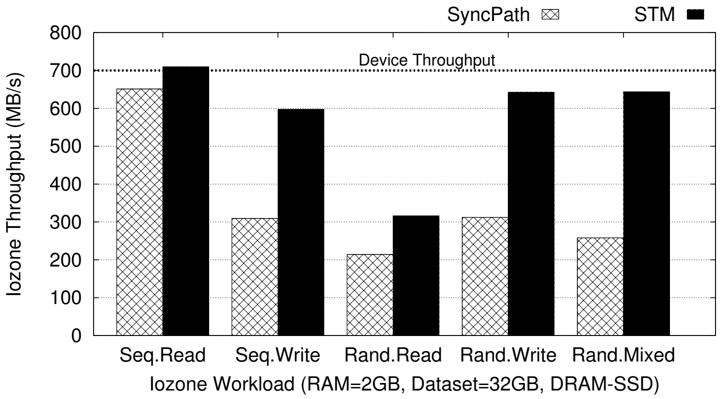


Figure 9: Iozone evaluation of STM and SyncPath

utilizes spatial merge; temporal merge effectively overcomes the limitations of spatial merge.

The weak point of STM is that the number of the concurrent block requests directly affects the performance. If a single thread issues I/O requests one-by-one, no improvement is expected over SyncPath since that thread will be a winner and always proceed with one I/O request at all time. This kind of problem is observed when a kernel thread performs write-back on dirty pages; even though multiple user processes concurrently invoke `write` system calls, their data remain in the page cache, and is later flushed by a single thread. Single-threaded write-intensive applications like `cp` or `dd`, suffer from performance degradation.

4.5 Design 3: Merging Discontiguous Block Requests Asynchronously

To decouple the I/O performance from the number of the concurrent block requests in the I/O subsystem, we choose to design another I/O subsystem, called ATM, that accumulates I/O requests regardless of the concurrency.

ATM substitutes the SCSI subsystem in the existing storage stack (§2.2) to make the best use of underlying storage devices. Instead of bypassing the I/O scheduler, ATM actively utilizes it to pile up I/O requests in a queue. When `kblockd` is scheduled by the OS, it invokes a customized dispatch routine of ATM, which dequeues multiple I/O requests at a time. Then, ATM builds a temporally-merged request and dispatches it.

Figure 10 shows an example of 4 threads' submitting six block requests. Each thread in the context of a user process prepares a DMAable buffer by *queue-bouncing* before it enters into ATM. So the burden of mapping multiple DMA buffers by a single thread, i.e. 6 ”M”s, is removed. After the I/O request is completed, SoftIRQ jobs of unmapping DMA buffers and waking up user processes are handed over to other cores. Consequently, the whole ascending I/O path is executed on the CPU where the I/O request was originally submitted.

At first, it was anticipated that ATM would form a larger I/O request than the one by STM and increase throughput. However, such expectation proved to be wrong; the number of requests was not high enough (4 at maximum) that the benefit of temporal merge had been canceled out by the soft-

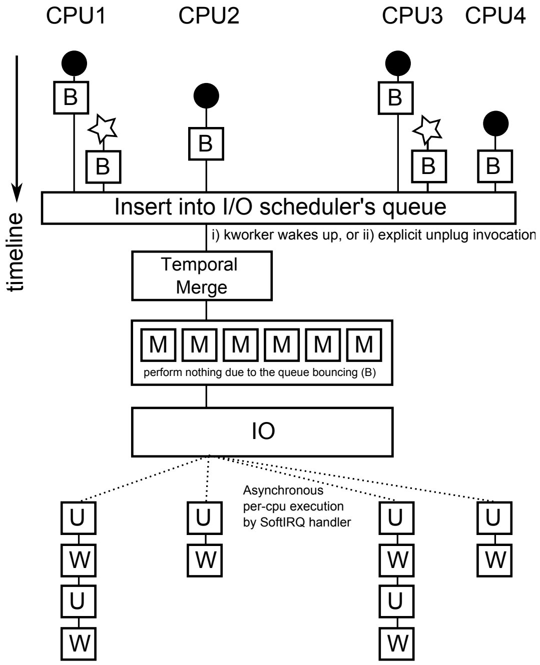

CPU1
CPU2
CPU3
CPU4
timeline
B
B
B
B
B
B
Insert into I/O scheduler's queue
i) kworker wakes up, or ii) explicit unplug invocation
Temporal Merge
M
M
M
M
M
M
perform nothing due to the queue bouncing (B)
IO
Asynchronous per-cpu execution by SoftIRQ handler
U
W
U
W
U
W
U
W
U
W
U
W

Figure 10: Temporal merge with I/O scheduler in ATM design

ware latency of the I/O scheduler. Some default parameters, in Table 2, that controls the behavior of I/O schedulers were only effective for HDD and thus limited the potential performance of low latency storage devices.

So, our next step was to eliminate disk assumptions in the I/O scheduler. ATM chooses `noop` instead of `cfq` which is the default I/O scheduler in the recent Linux releases. The former accumulates I/O requests to the maximum threshold, while the latter unplugs a request queue prematurely as described in §3.2.2. Increasing `unplug_thresh` from 4 to 32 shows a good balance between throughput and latency under various environments, but it should be tuned when the performance features of the underlying storage device change. `Unplug_timer` is set to 1 ms, the minimum timer pe-

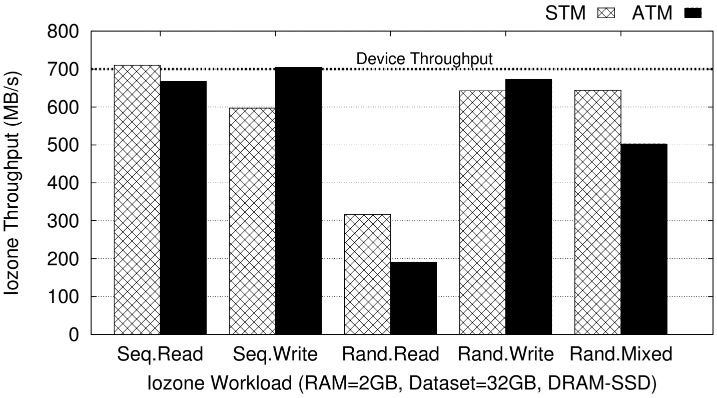


Figure 11: Iozone evaluation of ATM and STM

riod in our system, to prevent a storage device from being idle for too long. This can be more fine-grained by using high-resolution timers [44, 45].

Figure 11 indicates that ATM performs better than STM under write workloads; the improvements are 18%, 5% under sequential and random write workloads having multiple threads respectively and 50% when only a single thread issues many writes. In spite of the increased write throughput, ATM fails to perform temporal merge on the read requests due to the critical section design in the Block Layer. A read I/O request is dispatched as soon as they are inserted into a request queue with a spinlock *queuelock* held. This allows ATM only one I/O request at any time, lowering I/O performance to that of spatial merge. There exists a semantic gap between the Block Layer and ATM, which motivates our next I/O subsystem design.

4.6 Design 4: Choosing I/O Path Dynamically Based on a Request Property

From the previous results, we can infer that STM is a good choice for read-intensive workloads since the `read` system call is a blocking operation and thus significantly affected by the latency of I/O request. On the contrary, write-intensive applications would prefer the throughput-oriented design like ATM since `write` is non-blocking and rapid flushes of dirty pages into the storage device is a critical factor that enhances the application throughput.

Based on intuition, we came up with a hybrid design called HTM (Hybrid Temporal Merge). The general idea is to take advantages of both designs. If the I/O subsystem identifies an incoming I/O request to be latency-sensitive, it directs the request to STM and otherwise to ATM. HTM infers the ”latency-sensitivity” based on the `bio->bi_rw` flag. If the flag indicates that the request is a read type or an O_DIRECT type, HTM regards it as a latency-sensitive one, and makes it follow the synchronous I/O path in STM. As shown in Figure 12, HTM helps an application to observe the throughput achieved by STM in case of read workloads, and by ATM in case of write workloads.

Still, the random read throughput is only 45% of the device throughput. Considering that the random write throughput is very close to the device throughput, it is regarded that temporal merge is effectively enhancing device-/channel- utilization. The root cause of the low random read throughput is due to the semantic gap between the read-ahead logic in VFS layer

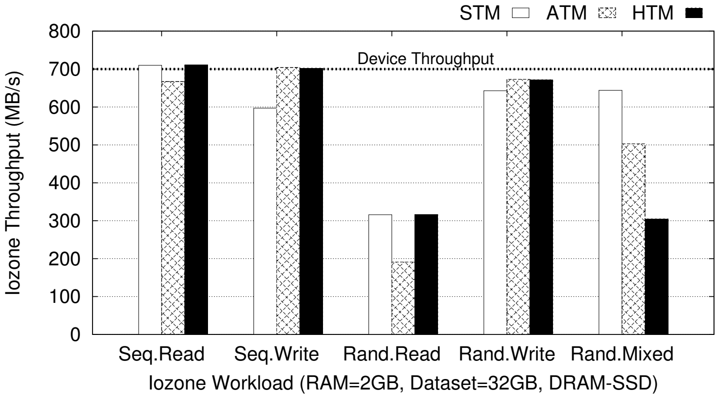


Figure 12: Iozone evaluation of HTM and ATM

and the underlying I/O subsystem. This problem motivates the need for designing a new type of I/O subsystem.

4.7 Design 5: Including Upper Layer to Bridge Semantic Gap between VFS and Block I/O Subsystem

Figure 13 shows the relationships between the size of read-ahead and the sequential/random read throughput of Iozone. The size of read-ahead determines how many pages should be read from a storage device when sequential access is detected and therefore can affect the sequential read throughput.

However, the variation of the random read throughput was unexpected since the access pattern cannot be sequential. It is confirmed that this misbehavior was due to the *context lookup* heuristics implemented inside VFS

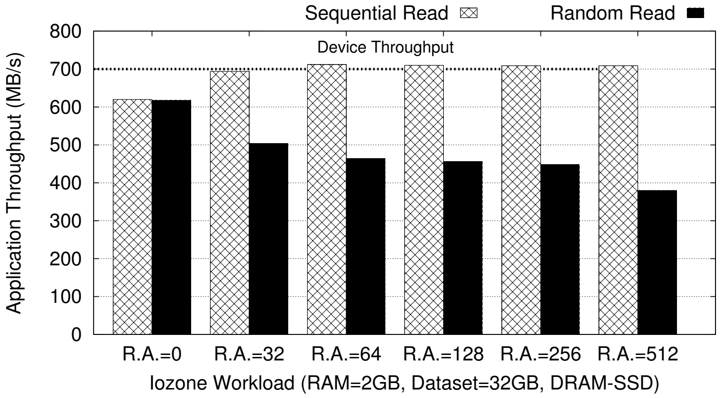


Figure 13: Read-ahead dilema in HTM

layer. The algorithm was for detecting sequentiality from the multiple sequential streams, but misunderstood when the random requests arrive in a small region within a short time interval. As a result, the random read throughput observed at HTM was in fact 610 MB/s while the *goodput*, the throughput observed at a user application, was 316 MB/s.

The *read-ahead dilema*, whether to enable read-ahead or not, can be resolved by simply disabling the context lookup logic. The new I/O subsystem, called VFS-HTM, is designed to cover this upper layer to avoid applying modification to kernel core in place. The evaluation result of VFS-HTM is shown in Figure 14. The random read throughput is increased by 94%, which achieves 87% of the device throughput.

It is noticeable that the mixed random throughput still remains at 74% of the device throughput. In spite of the use of the hybrid I/O path, the throughput is lower than both the throughput of the random read and the throughput of the random write. This performance gap is due to the lack of

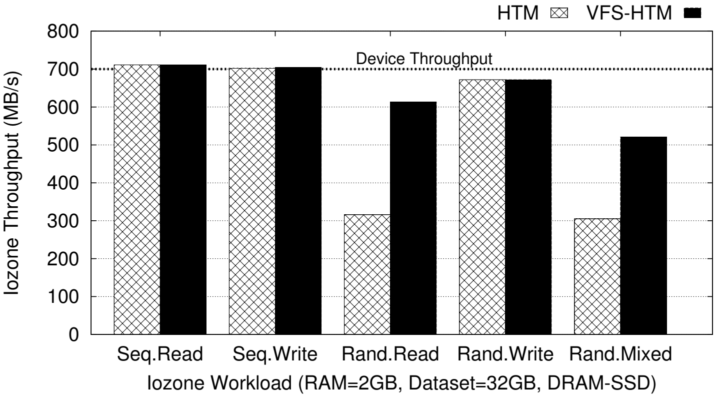


Figure 14: Iozone evaluation of VFS-HTM and HTM

global management of the two I/O paths for reads and writes. When a block request follows read I/O path, the write requests in a request queue may be prematurely dispatched to a storage device since the unplug event would be triggered to deal with the read request. The five I/O subsystem designs discussed up to now are optimized for read-only or write-only workloads but not for the mixture of reads and writes.

4.8 Design 6: Using Double Buffering to Avoid Lock Contention

To overcome the disadvantage of VFS-HTM regarding the interference between the read and the write path, we choose to improve the design of ATM by redesigning the critical section. The inherent problem is that the Block Layer in Linux uses the same spinlock both for inserting a request and for dispatching a request.

To mitigate the contention and give a chance for a request queue to

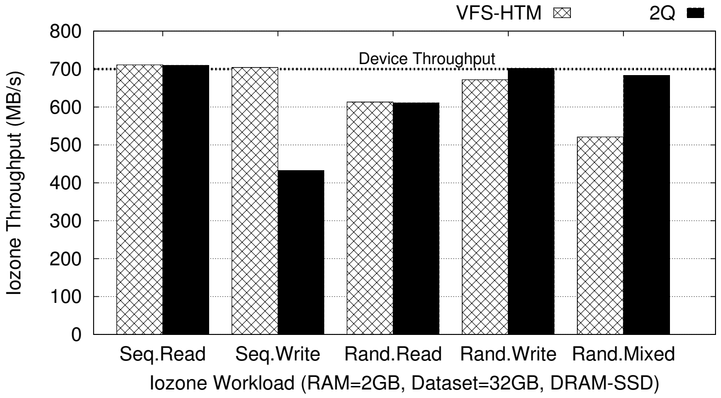


Figure 15: Iozone evaluation of 2Q and VFS-HTM

accumulate more requests, our new I/O subsystem, called 2Q, uses one more queue, called Shadow Queue (SQ), to buffer the requests to be dispatched soon. The procedure of moving block requests from the request queue to the SQ is called a *draining*. Draining block requests is short enough that holding the spinlock *queuelock* does starve other threads.

As described in Figure 15, 2Q shows 44% of improvement over VFS-HTM. Unlike VFS-HTM that unplugs a request queue for every read request, 2Q effectively piles up block requests regardless of their read/write types. However, the significant performance degradation is occurred under a sequential write workload, which is not observed when ext2 is used. It seems that the synchronous writes issued by a journaling thread prevent the next requests from entering into the I/O subsystem, while VFS-HTM does not experience this problem due to the bypass of the I/O scheduler. Identifying such synchronous requests and immediately unplugging a request queue is the remaining challenge of 2Q.

I/O subsystem design	O1 (Polling)	O2 (Synch. path)	O3 (Read-merge)	O3 (Write-merge)	O4 (De-diskify)	Read-ahead Fix
SCSI_INTR (Baseline)	×	×	×	△	×	×
SyncPath (Zero context switch)	O	O	×	×	×	×
STM (Synch. Temporal Merge)	O	△	O	△	×	×
ATM (Async. Temporal Merge)	O	△	×	O	O	×
HTM (Hybrid Temporal Merge)	O	△	O	O	O	×
VFS-HTM (VFS-included)	O	△	O	O	O	O
2Q (Double Buffering)	O	×	O	O	O	O

Table 4: Optimizations applied to each I/O subsystem version where each symbol indicates the followings, O: fully support, △: partially support, and ×: don’t support.

4.9 Design Summary

Key optimization techniques applied to the I/O subsystem designs are summarized in Table 4. The baseline I/O subsystem, SCSI_INTR, partially supports write-merge since it can combine requests based only on spatial-adjacency. ATM may synchronously dispatch I/O requests when the number of requests in a requst queue reaches the limit of unplug_thresh, so it is checked as ’partially support’.

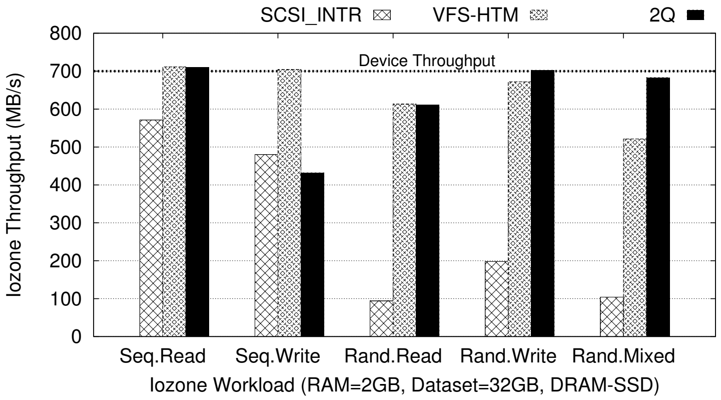


Figure 16: Iozone evaluation of 2Q, VFS-HTM and SCSI_INTR

Chapter 5

Implementation Details

In this chapter, the implementation details of the I/O subsystems and the hardware interface are presented.

5.1 Block I/O Subsystem in Linux

Each I/O subsystem is implemented as a loadable kernel module for linux 2.6.32 and requires no modification of the OS. The Linux storage stack is composed of several layers as described in §2.2 and each layer invokes the lower/upper layer's functionalities with function pointers. This OS design enables any device module to register its own customized function between two adjacent layers, which is exploited by our I/O subsystem implementation.

- *scsi_host_template→queuecommand*: The function pointer is invoked by SCSI subsystem to perform a device-specific protocol. SCSI_INTR registers the customized dispatch function that extracts DMA information from the SCSI command and issues a PCI request to our DRAM-SSD.

- *request_queue→make_request_fn*: File system uses the function pointer to utilize the I/O service provided by the I/O subsystem. Every block device has a request queue that provides a callback to which other device driver can register the device-specific dispatch routine. When an I/O request is about to be submitted to the I/O subsystem, the callback function is invoked. SyncPath, STM, HTM use this function pointer to bypass the I/O scheduler. Before entering into the I/O subsystem, a thread does not hold any spinlock, which enables the multiple threads to be in the I/O subsystem concurrently. STM utilizes such concurrency to combine the multiple block requests into one I/O request.

- *request_queue→request_fn*: The function pointer is invoked either 1) asynchronously by kblockd, or 2) synchronously by a process context. The semantics of the function pointer is that *it is time to dispatch an I/O request, so pick the most appropriate candidate from a request queue and dispatch it to a device*. This routine is usually invoked after spatial merge or request scheduling is performed on a request queue. ATM and HTM hooks the function pointer and dequeues as many block requests from a request queue as possible to perform temporal merge on them. Since a thread holds a spinlock, called *queuelock*, before calling the function pointer, no two threads can exist at this layer. So, simply moving STM into ATM does not produce the benefit of merging read requests. To take advantage of both I/O path, HTM hooks the make_request_fn first, checks the attribute of a block request and selectively re-directs the request to ATM by invoking the function address remembered at loading time of the module.

ATM implements cache-friendly request retirement (as discussed in §3.1.2) by using a SoftIRQ handler instead of Inter-Processor Interrupt (IPI). IPI forces other CPU cores to deal with the registered interrupt handler which incurs two context switches to make the original job resume its execution. On the contrary, SoftIRQ is an implementation of Soft Timer [36] and is invoked at the appropriate moment such as the point after an interrupt context is ended, which does not need to save the CPU context. The main benefit of ATM comes from the write-intensive workloads which favor high throughput rather than low latency, so SoftIRQ is the better choice for ATM than IPI. The existing post-processing handler in SCSI subsystem is designed to serially unmap DMA buffers, which can be significantly high for low-latency storage devices. Since ATM substitutes the SCSI subsystem, it can reuse SCSI-specific variables in a request. Four of them, tag, cmd_len, sense, and sense_len, are chosen to contain DMA-related information such as the direction, the number of scatter-gather entries, the pointer to the scatter-gather list and the size of the sgtable. Then the SoftIRQ handler invoked on each CPU core is able to unmap DMA buffers and keep the routine of request retirement.

To drain block requests, 2Q breaks the assumption that *request_fn function should keep holding a spinlock queuelock*. The 2Q works just like ATM, but when requested to dispatch an I/O request, it releases the spinlock and drains as many block requests as possible to perform temporal merge on the shadow queue. Before returning to the upper layer, 2Q should acquire the spinlock to guarantee the post conditions as expected by the upper layer.

5.2 New Storage Device Interface

To support polling mechanism for request completions, we developed a new interface in the FPGA of the DRAM-SSD. First, we added a control register, called INT_DISABLE, so that I/O subsystem can enable/disable interrupt mechanism. Next, we added a status register to report an I/O completion (IO_DONE). Our I/O subsystems utilize the two registers to implement polling mechanism.

Our DRAM-SSD has a separate DMA engine that performs device-level scatter-gather I/O operation by using a list of request descriptors. We defined a new data structure, Block Control Table (BCT), which can contain 1,024 requests in a table at maximum. Each element in BCT is encoded as (host memory segment, storage segment, data size). The kernel memory region of BCT is allocated when the device driver is loaded and accessed by the FPGA with consistent DMA access function.

Chapter 6

Evaluation

To evaluate our I/O subsystem designs for low latency storage devices, we used a set of microbenchmarks as shown in Table 5. For evaluation we used a machine with two Xeon E5630 2.53 quad core CPUs (total 8 cores) running a Linux 2.6.32 vanilla kernel. DRAM-SSD [15] is used as the low-latency storage device for evaluation. The tested I/O subsystems are SCSI_INTR, SyncPath, STM, ATM, HTM, and 2Q.

Benchmark	Configuration
IOzone [17]	Thr∈[1,8,16,32], RecSize=4KB, IO=buffered, FileSystem=ext3
fio [46]	Pattern=randrw, Size=32G, Thr∈[1,8,16,32,64]
postmark [47]	# of postmark instances∈[1,4,8,16], # of files=200K, # of trans.=300K

TPC-C	BenchmarkSQL [48] with Postgresql,
	Warehouse∈[10,50], Terminal∈[1, 5,10,15]
	fsync=off, auto_vaccuum=off, full_page_writes=off
	random_page_cost=1.0

Table 5: Configurations of the tested I/O workloads

6.1 Latency Reduction

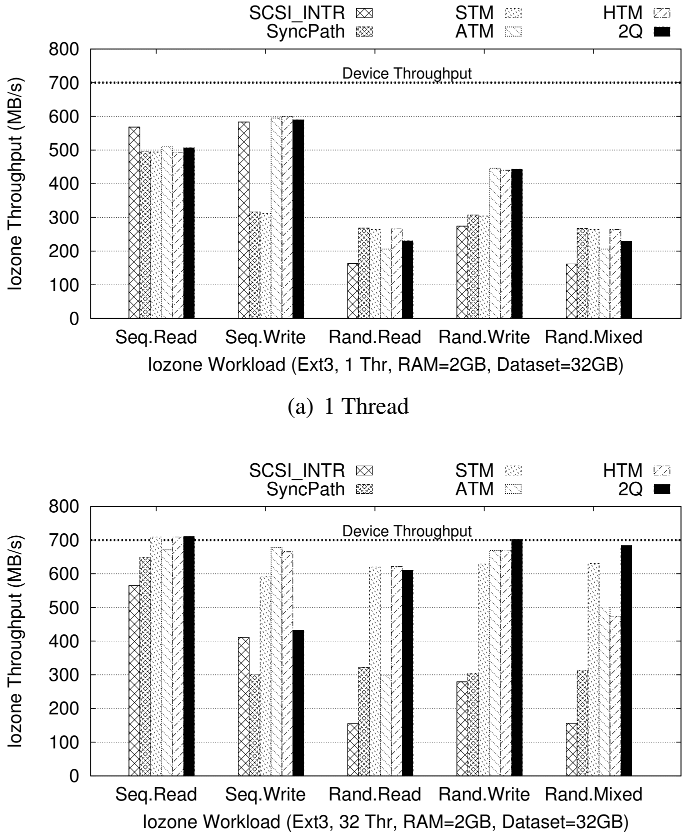
We measured the latency of a read I/O request by profiling the response time of *page cache miss* routine that starts from `do_sync_read` kernel

	Software-latency	Hardware-latency
SCSI_INTR	20 usecs (74%)	7 usecs (26%)
SCSI_POLL	13 usecs (65%)	7 usecs (35%)
SyncPath, STM, HTM	5 usecs (42%)	7 usecs (58%)
ATM	8 usecs (53%)	7 usecs (47%)
2Q	6 usecs (46%)	7 usecs (54%)

Table 6: Latency of accessing a 4KB page by each I/O subsystem

function, which is shown in Table 6. To measure a pure per-request latency, a single-threaded workload `dd` is executed with O_DIRECT option, which allows I/O subsystem only one I/O request at any time.

The baseline I/O subsystem SCSI_INTR incurs the highest software latency which accounts for 74% of the total latency. Using poll instead of interrupt cuts down the software latency by 7 usecs and eliminating any context switch in the I/O path reduces the additional delay by 8 usecs. The synchronous I/O path achieves 3.3x reduction in software latency when compared to the I/O path in the existing Linux I/O subsystem. In the case of STM and HTM, a user thread, dd, always becomes a winner and follows the synchronous I/O path just as in SyncPath; context switch does not occur in the path. Since both ATM and 2Q utilize the I/O scheduler to accumulate block requests in a request queue, the delay of 1∼3 usecs is added to the software latency. 2Q checks the state of a storage device and synchronously dispatches an I/O request if the device is idle, which accounts for the lower software latency than ATM.



(a) 1 Thread

(b) 32 Threads

Figure 17: Iozone evaluation of the proposed I/O subsystems

6.2 Microbenchmark 1: Iozone

Interrupt vs. Poll: In Table 7, the existing interrupt-based I/O subsystem, SCSI_INTR, outperforms all of the poll-based designs under a single-threaded sequential read workload. In the workload, VFS always detects the sequentiality of data access, and issues a large-sized read request containing 32 pages (= 128 KB). It takes 177 usecs for our DRAM-SSD to service the request. From the result, when per-request size is high enough, using interrupt instead of poll can be a better solution because the interrupt latency becomes relatively insignificant. On the contrary, when a system is heavily-loaded by many threads, poll is always better than interrupt, showing 15% of improvement in this case (SyncPath vs. SCSI_INTR). Both STM and 2Q can benefit more by performing temporal merge on multiple read-ahead requests.

Reduced context switch: When a single-threaded random read workload is given, the per-request software latency can be easily identified. The number of context switches of the I/O path in each I/O subsystem is 3∼4 for SCSI_INTR (a read and a write respectively), 1∼2 for (ATM, 2Q) and 0 for (SyncPath, STM, HTM). It is noticeable that the performance improvement over SCSI_INTR is inversely-proportional to the number of context switches: 27∼42% by (ATM, 2Q) and 63∼65% by (SyncPath, STM, HTM). Although the I/O path in STM may experience more context switches under a multi-threaded workload, the benefit of temporal merge cancels out the harm caused by context switch, showing more improvement.

STM vs. ATM: In Figure 17(a), STM fails to merge write requests under a sequential write workload since only a single journaling thread submits a series of 4 KB pages one by one; when the concurrency is 1, STM cannot merge any requests. On the contrary, ATM uses the asynchronus I/O path to accumulate many requests even when the concurrency is 1, and exploits the device throughput from the workload. Under a multi-threaded random read workload, ATM is equal to or less than SyncPath that does not perform any merge operation. The failure of ATM's merging read requests is due to the critical section design (as discussed in §4.5), while STM succeeds

Thr= 1, Ext3	Seq.R	Seq.W	Rand.R	Rand.W	Rand.M
SyncPath	0.87	0.54	1.65	1.12	1.65
STM	0.87	0.54	1.63	1.11	1.63
ATM	0.90	1.02	1.27	1.63	1.27
HTM	0.87	1.03	1.63	1.60	1.63
2Q	0.89	1.01	1.42	1.62	1.41

Thr=32, Ext3	Seq.R	Seq.W	Rand.R	Rand.W	Rand.M
SyncPath	1.15	0.74	2.07	1.09	2.02
STM	1.25	1.44	3.99	2.26	4.05
ATM	1.19	1.65	1.93	2.40	3.22
HTM	1.25	1.62	4.00	2.40	3.05
2Q	1.29	1.05	3.94	2.52	4.39

Table 7: Application throughput normalized to SCSI_INTR

in combining multiple read requests. These results obviously motivate the design of HTM that takes advantage of both designs. In the case of ext2, a write-back thread instead of a journaling thread submits a bunch of requests (total of 32 pages), so the sequential write throughput becomes high enough in spite of the lack of merge operations.

No performance penalty of STM: When we compare single-threaded throughput between SyncPath and STM, it is hard to find meaningful performance difference as in Table 7. Even when STM always fails to merge requests due to the single-thread workload, the performance degradation is less than 2% over SyncPath.

ATM vs. 2Q: ATM fails to merge read requests since the I/O scheduler in Linux is designed to dispatch an I/O request without delay if it is a read request. Compared to ATM, 2Q shows 2.04x throughput under a random read workload while the random write throughput shows no big difference.

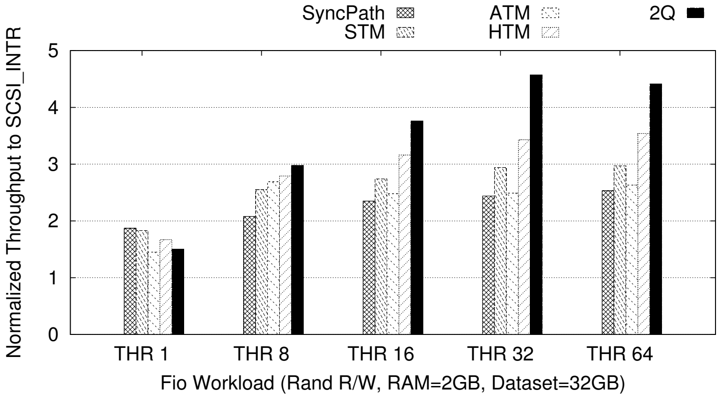


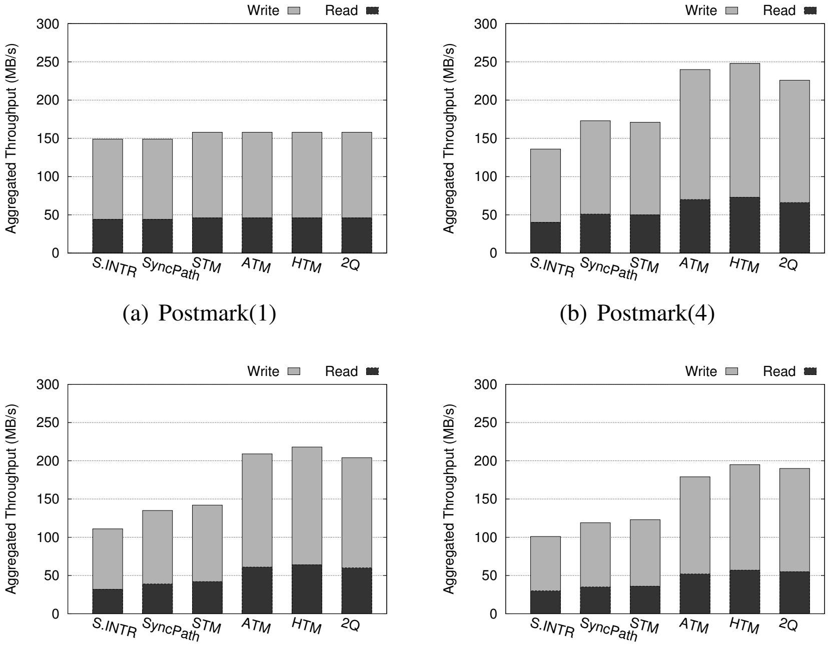
Figure 18: Fio evaluation of the proposed I/O subsystems

6.3 Microbenchmark 2: Fio

We have used other microbenchmarks to confirm the previous observations with different workloads. As shown in Figure 18, when the concurrency is 1, the I/O subsystems that rely on the I/O scheduler show worse throughput due to the increase of per-request software latency. As more threads exist in a system, temporal merge becomes more beneficial especially when used with the I/O scheduler. In the case of 2Q, when 32 threads issue read/write requests in random access pattern, 4.6x of improvement over SCSI_INTR is observed.

6.4 Macrobenchmark 1: Postmark

Postmark [47] simulates web servers by creating/appending/deleting a set of files, each of which corresponding to one user mail. Recently, it is reported that postmark does not generate I/O traffic enough to test a storage



(a) Postmark(1)

(b) Postmark(4)

(c) Postmark(8)

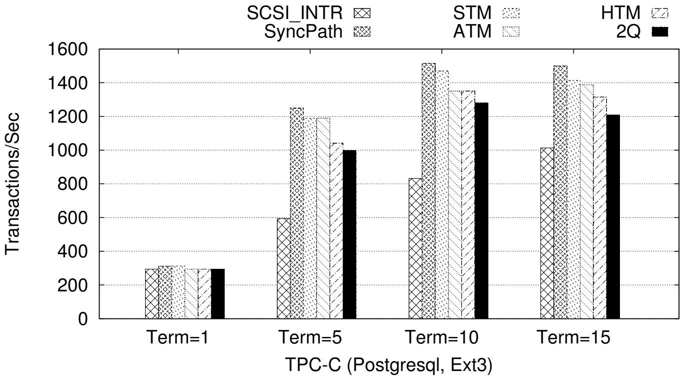
(d) Postmark(16)

Figure 19: Aggregated postmark throughput where postmark(N) means that the N instances of postmark are simultaneously executed

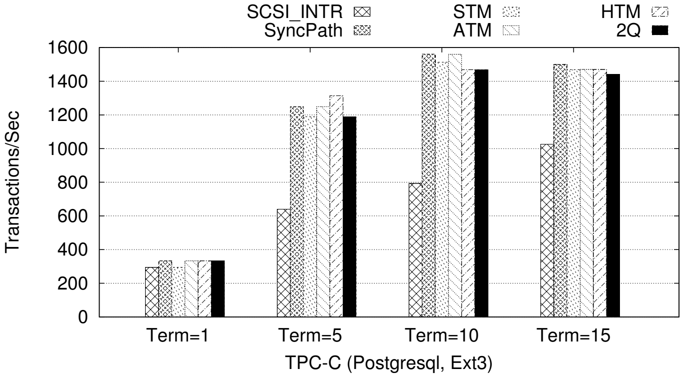
system [49]. One possible solution is to run multiple instances of postmark in parallel [50, 51]. This approach gives more stress to the underlying block I/O subsystem, so is chosen as our evaluation method. The configuration of each postmark instance is described in Table 5. From the results of aggregated postmark throughput in Figure 19, we have a few observations like the following.

Positive Effect of Increased Concurrency: Comparing Figure 19(a) and 19(b), we can observe the following throughput improvement: {SyncPath: 16%, STM: 8%, ATM: 52%, HTM: 57%, 2Q: 43%}. In spite of the incapability of merging requests, SyncPath achieves the improvement because high concurrency usually increases the number of outstanding I/O requests and reduces *inter-arrival delay* [52] between two successive requests. This positive effect also contributes to the improvements of other I/O subsystems. ATM with Postmark(4) accomplishes 1.52x improvement over ATM with Postmark(1), which is much more than what STM achieves. The reasons of the difference are 1) postmark generates write-intensive workload and 2) I/O schduler is very advantageous to the characteristic. As postmark produces write requests 7 times more than read ones, ATM, HTM, and 2Q with I/O scheduler show the relatively high performance.

Negative Effect of Increased Concurrency: Comparing Figure 19(b) and 19(d), we can observe the following throughput degradation: {SyncPath: -32%, STM: -29%, ATM: -26%, HTM: -22%, 2Q: -16%}. This result stems from resource contention; each I/O operation, e.g. DMA map/unmap and polling, requires a spinlock resource before the beginning for the purpose of guaranteeing the safe transition of a device state. If the concurrency in a system is increased, many CPU cycles would be consumed by each core during waiting for the spinlock resource. In the experiment, when the number of postmark instances is close to or higher than the number of CPU cores, the performance degradation is observed. It convinces us that the concurrency higher than a certain threshold may be harmful to I/O performance. To prevent the excessive number of threads from participating in I/O operations, we can think of another block I/O subsystem design having a limited number of CPU cores dedicated to I/O operations. SCSI_INTR experiences 32% of throughput degradation (149→101 MB/s) as the concurrency changes from 1 (Figure 19(a)) to 16 (Figure 19(d)). Such problem originates from the



(a) Warehouses=10



(b) Warehouses=50

Figure 20: TPC-C throughput with varying the number of warehouses

frequent interrupt events that prevents a foreground task from progressing, which is usually observed in receive livelock problem [37].

6.5 Macrobenchmark 2: TPC-C

BenchmarkSQL [48] is a multi-threaded java client implementation that simulates TPC-C workload. We used Postgresql as a backend DBMS and ran BenchmarkSQL on the same machine. The ratio of each transaction type that will be issued during an experiment follows TPC-C specification [53]: {NewOrder: 45%, Payment: 43%, OrderStatus: 4%, Delivery: 4%, StockLevel: 4%}. To maximize the number of outstanding I/O requests in a kernel, the configuration in Table 5 was used.

Slow I/O Path in DBMS (Term=1): As shown in Figure 20, the performance of each I/O subsystem is about the same to another when the number of terminals is 1. Since the most part of I/O path is contained in DBMS itself, the duration of CPU time in I/O subsystem is relatively small; high inter arrival delay causes the I/O subsystems to be in idle state often. Without proper optimizations to the I/O path in DBMS such as embedding device-awareness into application-level heuristics, our block I/O subsystem cannot benefit much from this workload.

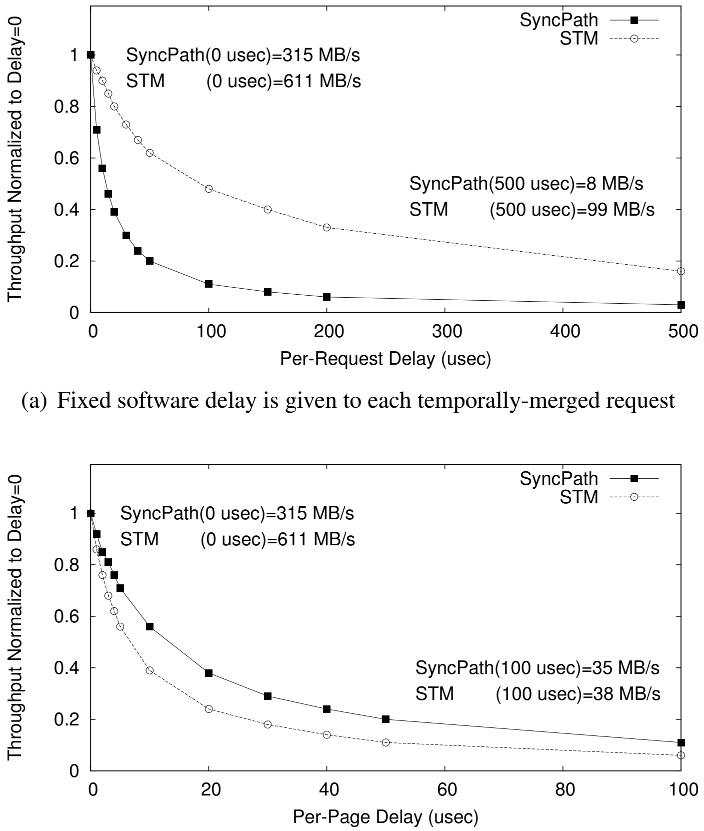
DBMS's Conservatism in Consistency Enforcement (Term>1): When the number of terminals increases (in Figure 20(a)), inter arrival delay becomes shorter due to the queued requests inside DBMS. In this case, the overhead of I/O subsystems dominates the software latency and the different throughputs are observed by the I/O subsystems. Interestingly, SyncPath without supporting request merging shows the highest transactions/sec (i.e. tps). Before experiments, we expected that ATM and 2Q would show the best throughput since TPC-C workload is known to incur many write requests, which is proven to be wrong by this experiment. DBMS mostly tries to keep the number of outstanding I/O requests low; InnoDB engine in MySQL is known to maintain only 2 or 3 concurrent requests in a kernel [54]. This kind of conservatism is to enforce consistency requirement of DBMS but at the cost of giving up futher optimizations achievable in block I/O subsystems. Hence, the high concurrency at application-level (or DBMS-level) is not directly translated to the benefit of request merging. SyncPath that does not incur the software latency due to request merging shows the best performance.

6.6 Sensitivity Analysis

To show the generality of our optimization techniques toward other storage devices with different performance features, we first emulate the various hardware latency of a storage device by using intentional software delay, and measure the Iozone throughput. We used the following two strategies to determine software delay for each temporally-merged I/O request: (**F**) the fixed amount of software delay I/O request and (**C**) the calculated software delay proportional to the size of an I/O request.

Graceful Performance Degradation of STM: Figure 21(a) shows the throughput variation of SyncPath and STM when the delay type **F** is given. The result proves that temporal merge scheme makes the I/O subsystem less sensitive to the increase of hardware latency; if hardware latency of a new storage device is about 500 usec (i.e. **F**=500) and device throughput remains the same to the DRAM-SSD in our evaluation (i.e. 700 MB/s), STM is able to achieve 16% of the original throughput with **F**=0, which is the 11.1 times of improvement over SyncPath.

Break-even Point of Request Merging: Figure 21(a) shows the throughput variation of SyncPath and STM when the delay type **C** is given. It is observed that the two I/O performance of SyncPath and STM are the same



(a) Fixed software delay is given to each temporally-merged request

(b) Software delay is proportional to request size

Figure 21: Influence of hardware latency on the benefit of temporal merge

(315*0.38 $\simeq$ 611*0.2) if 25 usec of software delay is given to each 4 KB data (i.e. **C**=25). We define the value (25 usec in this evaluation) *break-even point*. If hardware latency becomes higher than break-even point, the benefit of temporal merge could be trivial; For the case of **C**=0, sending 4 pages in a single I/O operation performs 1.57x better than sending just 1 page, while the former performs only 1.03x better than the latter when given **C**=20. In summary, if hardware latency of a new device has higher latency and lower device throughput than our DRAM-SSD, the temporal merge scheme would be beneficial only when the hardware latency is less than a certain break-even point.

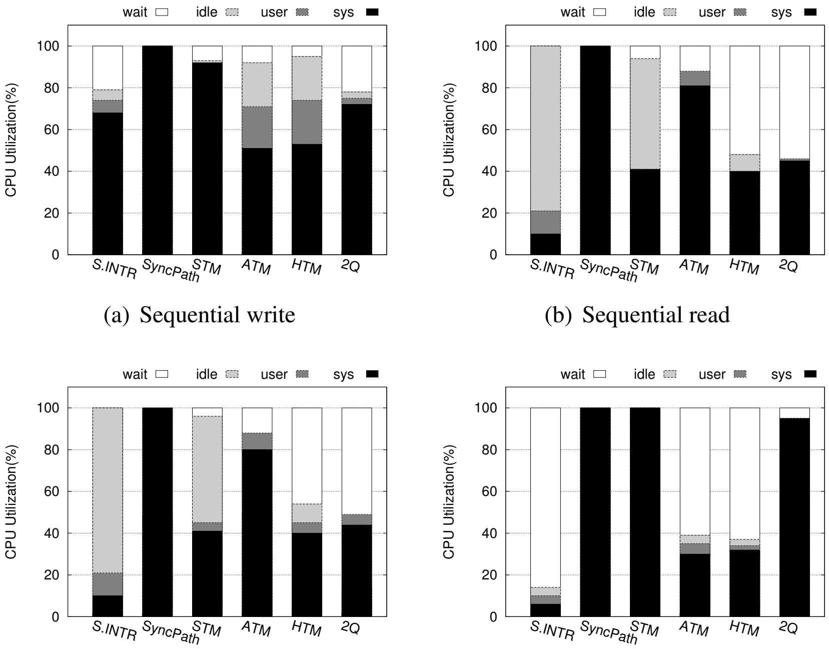
6.7 CPU Utilization

CPU time is classified into the four types as described in Table 8. *Sys time* includes the period of device polling, and if it becomes 100%, the overall system utilization may be degraded since any foreground tasks or OS services may be delayed.

CPU Time	Description
wait	CPU is idle & the number of outstanding requests>0
idle	CPU is idle & the number of outstanding requests=0
user	CPU is executing at the user level
sys	CPU is executing at the kernel level

Table 8: Description of the types of CPU time

CPU Utilization at Both Ends: SCSI_INTR always shows the least sum of user+sys time; Figure 22(c) indicates that SCSI_INTR makes 80% of CPU time *idle*, which means that there is no outstanding request for the most of time. High software overhead due to context switches or asynchronous I/O path causes the high idle time, which accounts for the low random read throughput of SCSI_INTR as shown in Figure 17(b). The other end is SyncPath; SyncPath saturates CPU resources under all of {Iozone, 32 Threads} workloads when the concurrency is high. Although SyncPath performs the best under TPC-C workload evaluation (§6.5), it may degrade responsive-



(a) Sequential write

(b) Sequential read

(c) Random read

(d) Random write

Figure 22: Profiling CPU utilization under {Iozone, 32 Threads}

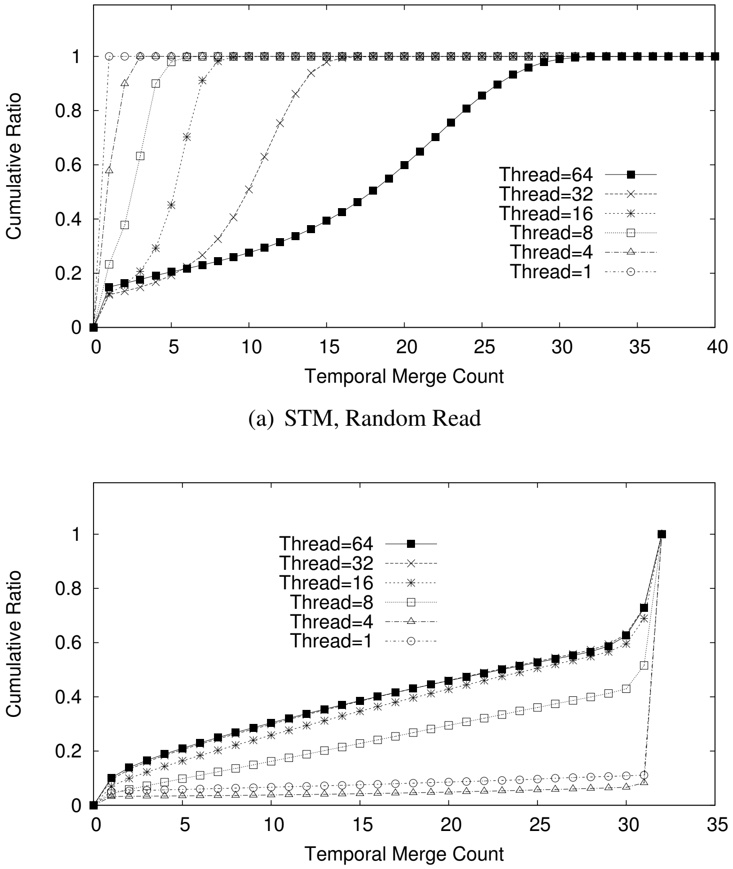
ness of some foreground tasks if they are highly-interactive such as bash or a movie player.

STM vs. ATM: Figure 22(c) and 22(d) describe that STM and ATM save CPU cycles under different workloads; STM saves 55% of CPU cycles under random read workload, while ATM does 65% of CPU cycles under random write workload. In the opposite case, STM/ATM saturates 100%/88% of CPU resources under random write/read workloads respectively. The main difference originates from the use of I/O scheduler; I/O scheduler is useful to accumulate write requests but cannot pile up read requests since the critical section design of block layer in Linux as discussed in (§4.5). HTM takes the advantages from the both and always maintains the moderate CPU utilization.

6.8 Temporal Merge Count

Figure 23(a) demonstrates that the concurrency significantly affects the distribution of *temporal merge count*, i.e. the number block requests in an I/O operation. In the cases of SCSI_INTR and ATM, 95% of the merge count was 1 (not shown in this graph).

ATM effectively collects multiple (write) requests when the concurrency is not high, as shown in Figure 23(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-/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.



(a) STM, Random Read

(b) ATM, Random Write

Figure 23: The cumulative distribution of merge count under Fio workload with 4 KB random read/write

Chapter 7

Related Work

An OS has various software stack including block I/O subsystem, network I/O subsystem, virtual memory subsystem, and etc. They have layered architectures [55] that compose of multiple software components; two adjacent components communicate with each other via a specific interface. The advantage of a layered architecture is that optimizations can be applied to each layer independently of others as long as the interface is correctly used. However, as the interface is static, those layers fail to exchange useful information about advanced hardware features, which limits the potential I/O performance of the hardware. We call this as *semantic gap problem* [18, 56].

In the following section, we examine many optimization techniques applied to network I/O subsystem and how they affect the high-performance storage stack in other studies. Hardware functionalities as well as software optimizations can help enhance the I/O performance of storage system. Finally, we will describe many researches that focus on bridging the semantic gap between software and hardware by using an extended interface.

7.1 Software Stack Optimization

7.1.1 Network I/O Subsystem

Network Interface Card (NIC) is another low-latency device that strongly demands highly-optimized software stack. Recent NICs such as Inifiband [57], Myrinet [58], 10g Ethernet and etc, guarantees latency as low as a few microseconds, and therefore could not be fully exploited by the existing OS components. To achieve maximum throughput from NICs, as high as network link speed, network I/O subsystem applies many optimizations to incoming packets based on two major principles [59, 60]: 1) reducing per-packet overhead and 2) reducing per-byte overhead. The former principle mitigates per-packet processing by batching multiple packets and benefits from the high link utilization. Extended frame [59, 61] and interrupt coalescing [60] belong to this case. The latter one boosts the speed of per-packet processing by offloading checksum calculation [59], zero-copy networking [60, 61], and etc. Those principles are very similar to ours; the I/O subsystems proposed in this paper 1) minimize per-request latencies by merging temporally-adajcent block requests, and 2) maximize the performance of request processing by utilizing synchronous I/O path.

Intel recently announced the new technology, *Intel I/O Acceleration Technology (IOAT)* to improve data flow across the platform to enhance system performance [62, 63]. They found network bottleneck in multi-gigabit ethernet environment and clarified what factors had made such bottleneck. They are classified into three categories: 1) system overhead (e.g. interrupt handling, buffer management), 2) TCP/IP processing overhead, and

3) memory access overhead (e.g. data moves, CPU stalls). To eliminate these overhead, IOAT uses parallel processing of TCP, asynchronous low-cost copy, TCP/IP stack optimization and etc. IOAT gives us some hints to improve our I/O subsystem designs. Memory copy overheads and cache-awareness are not considered in our proposed solutions. If hardware latency, which is 7 usec for reading/writing a page, becomes lower due to memory technology, system overhead would become more distinguishable that the IOAT-like approach could contribute to an optimized storage stack.

RouteBricks [64] sets two performance goals, which are 1) fully exploiting network bandwidth with a single server, and 2) achieving scalable performance across multiple servers. To achieve the first goal, RouteBrick devised per-core network queue to avoid contention between cores; when packets arrive within a few microseconds, a single queue suffers from serious contention. The paper explores the design of network I/O subsystem to run Click router [65] efficiently, e.g. whether to use pipelining or parallelizing, whether to use multiple queues, where to split packets at NIC or CPU cores, and etc. This approach is very similar to our design exploration of block I/O subsystems considering several design choices. The second performance goal can support RAID construction of multiple storage devices. The existing RAID software cannot benefit from our I/O subsystem since it has false assumption about underlying devices; it believes that the underlying I/O subsystem would interact with devices via interrupt instead of poll. Due to the synchronous I/O path optimization, the striping code intended for parallel accesses to multiple devices is serialized and limits the scalability of RAID performance. Hence, making block I/O subsytem RAID-aware and exploiting parallelism across multiple devices will be one of our future works.

PacketShader [66] offloads some computation to GPU cores and acceleartes routing functionality. The basic idea is to exploit the processing power of hundreds of GPU cores in network I/O subsystem. PacketShader optimized network I/O subsystem in many aspects since the latency of a graphic card is about a few microseconds and its benefit is canceled out by the slow network stack; the kernel launch time, i.e. the latency between the time to hand over a computation job to GPU and the time to start the job, was much higher than the computation time itself. PacketShader implements a new packet engine that utilizes huge packet buffer, gather/scatter machanism, and etc. to minimize per-packet overhead and to maximize parallelism across GPU cores.

7.1.2 Block I/O Subsystem

With the advent of high performance storage devices, researches assert the need for optimized storage stack to fully utilize their performance. Moneta [8] and Onyx [9] applied several software techniques such as the bypass of I/O scheduler, avoiding interrupts and the removal of spinlocks as well as a few hardware optimizations; The paper [41] also suggests that short-circuiting SCSI subsystem and ATA driver is a way to reduce software-latency. However, these works mainly focus on the benefit of individual optimization technique but not much on its side-effects. For example, spinning [8] reduces software-latency but may degrade the performance of foreground tasks due to high CPU usage.

Recently, the Linux block layer started to learn from the lessons in network stack by reflecting polling mechanism into its design. Blk-poll [67] is known as a general framework to implement a customized polling routine for a block device for the purpose of reducing per-request latency. It gives us a chance of designing another I/O subsystem, which is one of our future works.

7.2 Exploiting Device Functionality

A new device funcationality can contribute to high I/O performance and affect to the design of software stack. For example, modern ethernet NIC supports RDMA [60] and TCP offloading [68]. They minimize overhead in network I/O subsystem by copying data from a device to memory (or vice versa) without host CPU intervention and relieving the burden of TCP/IP processing from host CPU. They can operate either in interrupt mode or in polling mode; using the hybrid approach [40], network I/O subsystem achieves high throughput under heavily-loaded situation while maintaining high responsiveness under the opposite situation.

Block I/O subsystem has been developed to make the best use of new storage device features. Modern storage devices implement command queueing technique (e.g. TCQ, NCQ [31]) which enable the devices to reorder I/O requests with hardware/firmware-level information; for example, NCQ correctly estimates the disk head position and reflects the information into the service order, which had been impossible for software-level approach (Ref: SUNY). Block I/O subsystem exploits the feature and handles out-of-order request completions with tag management scheme.

The DRAM-SSD used in Moneta [8] supports full-duplex mode enabling concurrent data transfer between read stream and write stream. Considering the new feature, Moneta proposes 2Q design that maintains two independent request queues for reads and writes. It is reported that the scheme enforces no performance penelty over the existing schemes and achieves about two times of throughput improvement under mixed read/write workloads.

7.3 Extending Device Interface

Some new device features proposed in academic society can be realized only when storage vendors implement new interfaces to exploit the benefits. We call the new interfaces as *extended interfaces* since they contain more high-level information than the existing narrow interfaces [18, 56, 69] The most representative interface extension of a storage device is Object-based Storage Interface (OSD) [70] specified by T10 technical committee. The OSD interface re-defines the roles of block I/O subsystem and block devices; the extended interface directly relays high-level information such as file creation/deletion to the underlying device, while the existing interface only allows block-level read/write requests not having block liveness information. This gives additional optimization chances to storage devices since they can choose the most appropriate block candidates based on hardware knowledge.

The new block I/O interface proposed in this paper is beyond the standard and needs hardware modifications. Designing a new interface and implementing its semantics in hardware should be supported by vendors. For example, in case of SATA-2 devices, adding a new block I/O interface requires modifications to Advanced Host Controller Interface (AHCI) [71] by Intel and the firmware of storage devices by manufacturers such as WD, Seagate and etc. For recent SSDs connected to a PCI-E channel like FusionIO's [30] or OCZ's [72], the vendors should design their own register maps and distribute the new device driver to allow communication with the new I/O ports.

Chapter 8

Conclusion

We have explored the six I/O subsystem designs to find the most efficient one that relays the high performance of a low-latency storage device to an application without performance loss. To achieve the performance goal, the I/O subsystem should part from disk-based storage stack; our optimized I/O subsystems proposed in this paper are based on two key optimizations, 1) a low-latency synchronous I/O path, considered unpractical in traditional disk-based storage system, and 2) a new request batching scheme, called temporal merge, not achievable by the existing block I/O interface. Unlike the previous researches that have focused mostly on bypassing several software layers to OS delay [41, 8, 9, 42], 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 discussed to utilize our technique for commercial/enterprise use:

- **Parallelism Across Devices:** It is a very common practice to combine multiple storage devices in a RAID device when we consider enterprise-class storage products [73, 74]. RAID enables an OS to access multiple devices concurrently by splitting a large I/O request into smaller ones and dispatching them in a parallel fashion. However, the existing RAID software is based on the assumption that an

OS interacts with storage devices asynchronoulsy via interrupt mechanism; the optimization of synchronous I/O path and polling mechanism would prevent an OS from dispatching requests in parallel. To deal with this problem, our block I/O subsystem should be distinguish whether the underlying device is a physical block device or a RAID device. If it is a RAID device, the polling mechanism should be virtualized; 1) an I/O request could be dispatched to an idle storage device even if poll is started, 2) all of busy devices part of a RAID device should be polled in a round-robin fashion. The one of our future works is to implement RAID-awareness into our next design of block I/O subsystem.

- **Parallelism Across Servers:** In this paper, only a single server environment is considered; the benchmarks do not stress out network I/O subsystem or virtual memory subsystem but put pressure on block I/O subsystem only. For a hadoop cluster [75] consisting of multiple server nodes, HDFS would read data from one node and transfer it to other nodes, consuming CPU resources for block I/O subsystem and network I/O subsystem respectively. In this case, I/O path of a request becomes longer than the one discussed in §2.2, which affects design choices such as synchronous vs. asynchronous data transfer, packet batching schemes and etc. The cross-domain optimization performed by IO-Lite [76] is required in this example scenario.

- **Application-level Optimization:** DBMSes including Postgresql, Mysql and Oracle usually perform most of data management for themselves instead of relying on OS services; page buffering, prefetching, flushing, synchronizing operations are usually done at application-level. In the case that most parts of I/O path are hidden by an application, block I/O subsystem cannot benefit much from temporal merge scheme due to the small number of requests at the time of batching them. According to our TPC-C evaluation (§6.5), SyncPath that simply dispatches block requests as soon as possible without batching scheme outperforms others. In the end, optimizing application layer with storage stack is essential to achieve device throughput at application-level. Event channel between threads should be synchronous if it becomes bottleneck, and any heuristic, e.g. spatial merge, based on the assumption that the underlying storage device is a disk must be eliminiated. Optimizing applications such as DBMS, Java VM, web server and making them device-aware is one of our future works.

- **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.

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 hardly changed for generations, which we have already learned from old lesson [18]. The extended block I/O interface, i.e. device-level scatter-gather I/O, gives an OS a chance to exploit the maximum throughput from low-latency memory-based storage devices. We suggest that a next-generation host controller interface, e.g. NVMHCI [19], should include this kind of functionality into its design. One of our future work directs to broaden the scope of an I/O subsystem to cover the upper layers like file system, VFS, or even user-level library [77]. We believe that the semantic gap we found between the VFS and the I/O subsystem can be an example leading us to extend discussions of ”low-latency device”-aware optimizations to the entire storage stack.

참고 문헌

[1] G. R. Ganger and Y. N. Patt, "Using system-level models to evaluate i/o subsystem designs," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 47, no. 6, pp. 667–678, 1998.

[2] M. Dunn and A. L. N. Reddy, "A new i/o scheduler for solid state devices," Tech. Rep. TAMU-ECE-2009-02, Department of Electrical and Computer Engineering Texas AM University, 2009.

[3] D. I. Shin, Y. J. Yu, H. S. Kim, H. Eom, and H. Y. Yeom, "Request bridging and interleaving: Improving the performance of small synchronous updates under seek-optimizing disk subsystems," *Trans. Storage*, vol. 7, pp. 4:1–4:31, July 2011.

[4] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," ACM SIGMETRICS '94, pp. 241–251, 1994.

[5] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, (New York, NY, USA), pp. 145–158, ACM, 2009.

[6] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, (New York, NY, USA), pp. 1075–1086, ACM, 2008.

[7] T. Mikolajick, C. Dehm, W. Hartner, I. Kasko, M. Kastner, N. Nagel, M. Moert, and C. Mazure, "Feram technology for high density applications," *Microelectronics Reliability*, vol. 41, no. 7, pp. 947 – 950, 2001.

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, “Moneta: A high-performance storage array architecture for next-generation, non-volatile memories,” in *Proceedings of the 2010 43rd Annual IEEE/ACM MICRO’10*, pp. 385–395, 2010.

[9] A. Ameen, Caulfield, T. I. Adrian M., Mollov, R. K. Gupta, and S. Swanson, “Onyx: A protoype phase-change memory storage array,” in *Proceedings of the 3rd USENIX HotStorage’11*, pp. 1–5, 2011.

[10] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. c. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. h. Chen, H. l. Lung, and C. H. Lam, “Phase-change random access memory: A scalable technology,” 2008.

[11] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, “Dfs: a file system for virtualized flash storage,” in *FAST’10*, pp. 7–7, USENIX Association.

[12] Fusion-IO, “iomemory virtual storage layer (vsl), http://www.fusionio.com/overviews/vsl-technical-overview/.”

[13] C. Ruemmler and J. Wilkes, “An introduction to disk drive modeling,” *Computer*, vol. 27, pp. 17–28, March 1994.

[14] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, “Storage performance virtualization via throughput and latency control,” *Trans. Storage*, vol. 2, pp. 283–308, August 2006.

[15] TailwindStorage, “Extreme 3804, http://tailwindstorage.com /products/.”

[16] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano, “A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram,” in *Electron Devices Meeting, 2005.*, pp. 459 –462.

[17] W. Norcott and D. Capps, “Iozone benchmark, http://www.iozone.org.”

[18] G. R. Ganger, "Blurring the line between oses and storage devices," 2001.

[19] A. Huffman, "Nvm express revision 1.0c," tech. rep., Intel Corporation, 2012.

[20] R. F. Freitas and W. W. Wilcke, "Storage-class memory: the next storage system technology," *IBM J. Res. Dev.*, vol. 52, pp. 439–447, July 2008.

[21] A. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pp. 1 –11, nov. 2010.

[22] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 133–146, ACM, 2009.

[23] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," ISCA '09, pp. 2–13, 2009.

[24] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual ISCA'09*, pp. 24–33, 2009.

[25] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," ISCA '09, pp. 14–23, ACM.

[26] S. Lee, K. Fleming, J. Park, K. Ha, A. M. Caulfield, S. Swanson, Arvind, and J. Kim, "Bluessd: An open platform for cross-layer ex-

periments for nand flash-based ssds," in *The 5th Workshop on Architectural Research Prototyping*, 2010.

[27] B. Yoo, Y. Won, J. Choi, S. Yoon, S. Cho, and S. Kang, "Ssd characterization: from energy consumption's perspective," in *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2011.

[28] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pp. 187 – 196, 2002.

[29] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, "Ozone (o3): An out-of-order flash memory controller architecture," *Computers, IEEE Transactions on*, vol. 60, pp. 653 –666, may 2011.

[30] Fusion-IO, "iodrive octal data sheet, http://www.fusionio.com/data-sheets/iodrive-octal-data-sheet/."

[31] B. Dees, "Native command queuing - advanced performance in desktop storage," *Potentials, IEEE*, vol. 24, pp. 4 – 7, oct.-nov. 2005.

[32] S. Seelam, R. Romero, P. Teller, and B. Buros, "Enhancements to linux i/o scheduling," in *Linux Symposium*, 2005.

[33] S. Iyer and P. Druschel, "Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, (New York, NY, USA), pp. 117–130, ACM, 2001.

[34] Intel and Seagate, "Serial ata native command queueing," joint whitepaper, Intel Corp. and Seagate Technology, 2003.

[35] L. Huang and T. Chiueh, “Implementation of a rotation latency sensitive disk scheduler,” Technical Report ECSL-TR81, SUNY, Stony Brook, 2000.

[36] M. Aron and P. Druschel, “Soft timers: efficient microsecond software timer support for network processing,” *ACM Trans. Comput. Syst.*, vol. 18, pp. 197–228, August 2000.

[37] X. Chang, J. Muppala, Z. Han, and J. Liu, “Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts,” in *ICC’08.*, pp. 1835 –1839.

[38] I. Kim, J. Moon, and H. Y. Yeom, “Timer-based interrupt mitigation for high performance packet processing,” in *In Proc. 5th International Conference on HighPerformance Computing in the Asia-Pacific Region, Gold*, 2001.

[39] K. Salah, K. El-Badawi, and F. Haidari, “Performance analysis and comparison of interrupt-handling schemes in gigabit networks,” *Comput. Commun.*, vol. 30, pp. 3425–3441, November 2007.

[40] K. Salah and A. Qahtan, “Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme,” *Comput. Commun.*, vol. 32, pp. 179–188, January 2009.

[41] E. Seppanen, M. O’Keefe, and D. Lilja, “High performance solid state storage under linux,” in *Mass Storage Systems and Technologies (MSST’10)*, pp. 1 –12.

[42] J. Yang, D. B. Minturn, and F. Hady, “When poll is better than interrupt,” in *FAST’12*.

[43] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, “Disk schedulers for solid state drivers,” in *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT ’09, (New York, NY, USA), pp. 295–304, ACM, 2009.

[44] T. Gleixner and D. Niehaus, “Hrtimers and beyond: Transforming the linux time subsystems,” in *the Ottawa Linux Symposium (OLS)*, vol. 1, pp. 333–346, July 2006.

[45] T. Gleixner and I. Molnar, “hrtimers - subsystem for high-resolution kernel timers, kernel documentation ”hrtimers.txt”.”

[46] J. Axboe, “Fio benchmark, http://freshmeat.net/projects/fio.”

[47] J. Katcher, “Postmark: A new file system benchmark,” 1997.

[48] BenchmarkSQL, “http://sourceforge.net/ projects/benchmarksql.”

[49] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, “A nine year study of file system and storage benchmarking,” *Trans. Storage*, vol. 4, pp. 5:1–5:56, May 2008.

[50] A. Aranya, C. P. Wright, and E. Zadok, “Tracefs: A file system to trace them all,” in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST ’04, (Berkeley, CA, USA), pp. 129–145, USENIX Association, 2004.

[51] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan, “Buttress: A toolkit for flexible and high fidelity i/o benchmarking,” in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST ’04, (Berkeley, CA, USA), pp. 45–58, USENIX Association, 2004.

[52] N. Talagala, R. Arpaci-Dusseau, and D. Patterson, “Micro-benchmark based extraction of local and global disk,” tech. rep., Berkeley, CA, USA, 2000.

[53] T. P. P. Council, “Tpc benchmark c, standard specification, revision 5.11, http://www.tpc.org/tpcc/spec/tpcc_current.pdf,” 2010.

[54] C. Hall and P. Bonnet, “Getting priorities straight: improving linux support for database i/o,” in *Proceedings of the 31st international con-*

ference on Very large data bases, VLDB ’05, pp. 1116–1127, VLDB Endowment, 2005.

[55] T. Shiroshita, “A data processing performance model for the osi application layer protocols,” in *Proceedings of the ACM symposium on Communications architectures & protocols*, SIGCOMM ’90, (New York, NY, USA), pp. 60–68, ACM, 1990.

[56] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Bridging the information gap in storage protocol stacks,” in *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC ’02, (Berkeley, CA, USA), pp. 177–190, USENIX Association, 2002.

[57] Infiniband, “http://www.mellanox.com/.”

[58] Myrinet, “http://www.myricom.com/.”

[59] A. Menon and W. Zwaenepoel, “Optimizing tcp receive performance,” in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC’08, (Berkeley, CA, USA), pp. 85–98, USENIX Association, 2008.

[60] J. Chase, A. Gallatin, and K. Yocum, “End-system optimizations for high-speed tcp,” *IEEE Communications Magazine*, vol. 39, pp. 68–74, 2000.

[61] S. Makineni and R. Iyer, “Architectural characterization of tcp/ip packet processing on the pentium&#174; m microprocessor,” in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA ’04, (Washington, DC, USA), pp. 152–, IEEE Computer Society, 2004.

[62] Intel, “Improving network performance in multi-core systems,” tech. rep., 2007.

[63] Intel, “Accelerating high-speed networking with intel i/o acceleration technology,” tech. rep., 2006.

[64] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, “Routebricks: exploiting parallelism to scale software routers,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP ’09, (New York, NY, USA), pp. 15–28, ACM, 2009.

[65] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM Trans. Comput. Syst.*, vol. 18, pp. 263–297, Aug. 2000.

[66] S. Han, K. Jang, K. Park, and S. Moon, “Packetshader: a gpu-accelerated software router,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 195–206, Aug. 2010.

[67] J. Corbet, “Interrupt mitigation in the block layer, http://lwn.net/articles/346219.”

[68] H.-y. Kim and S. Rixner, “Connection handoff policies for tcp offload network interfaces,” in *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI ’06, (Berkeley, CA, USA), pp. 293–306, USENIX Association, 2006.

[69] G. Sivathanu, S. Sundararaman, and E. Zadok, “Type-safe disks,” in *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI ’06, (Berkeley, CA, USA), pp. 15–28, USENIX Association, 2006.

[70] M. Mesnier, G. Ganger, and E. Riedel, “Object-based storage,” *Communications Magazine, IEEE*, vol. 41, pp. 84 – 90, aug. 2003.

[71] J. Boyd, “Serial ata advanced host conroller interface (ahci) 1.3,” June 2008.

[72] OCZ, “Ocz revodrive pci-express ssd, http://www.ocztechnology.com/ocz-revodrive-pci-express-ssd.html.”

[73] NetApp, “Fas6200 series enterprise storage systems, http://www.netapp.com/us/products/storage-systems/fas6200/.”

[74] EMC, “Emc clariion raid 6 technology: A detailed review, http://www.emc.com/collateral/hardware/white-papers/h2891-clariion-raid-6.pdf.”

[75] Hadoop, “http://hadoop.apache.org/.”

[76] V. S. Pai, P. Druschel, and W. Zwaenepoel, “Io-lite: a unified i/o buffering and caching system,” *ACM Trans. Comput. Syst.*, vol. 18, pp. 37–66, Feb. 2000.

[77] A. M. Caulfield *et al.*, “Providing safe, user space access to fast, solid state disks,” in *ASPLOS’12*, ACM.

Abstract

메모리 기술의 발달은 저장 장치 하드웨어의 발전을 가져오게 되었고, 이는 데이터 접근의 패러다임을 기계적 방식에서 전기적 방식으로 이동하게 만들었다. 그 결과, 솔리드 스테이트 드라이브 (SSD)의 응답시간은 마이크로초 수준으로 줄어들게 되었다. 하지만 이러한 빠른 저장 장치의 등장에도 불구하고, 기존의 스토리지 스택은 그러한 새로운 장치의 속도를 따라올 수 없는 문제를 가지고 있는데, 그 이유는 스토리지 스택이 수십년간 매우 느린 디스크에 기반하여 최적화되어왔기 때문이다. Fusion-IO 나 OCZ 와 같은 저장 장치 제조 회사들은 자사의 고성능 저장 장치의 이점을 극대화하기 위해 최적화된 별도의 스토리지 스택을 구현하기 시작했다. 이제 스토리지 시스템은 빠른 저장 장치의 낮은 응답 시간 특성을 최대한 이용할 수 있어야 한다는 도전에 직면해 있다.

본 논문에서는, 매우 낮은 응답 속도를 가지는 SSD의 성능을 최대한 이용할 수 있는 블럭 입출력 서브시스템의 6가지 타입에 대해 제안한다. 우리의 최적화 기법은 다음의 두가지로 요약할 수 있다; 1) 입출력 경로를 재디자인 함으로써 개별 요청의 오버헤드를 줄이는 것, 2) 다수의 요청을 모아서 처리함으로써 개별 요청의 오버헤드를 가리는 것이다. *디바이스 폴링과 동기적 입출력 경로*가 첫번째 기법에 해당하고, *비연속 요청을 하나의 I/O로 처리하는 것*이 두번째 기법에 해당한다. 기존의 일들이 불필요한 소프트웨어 계층을 제거하는데 초점을 두었던 것과는 달리, 우리는 적극적으로 기존의 소프트웨어 컴포넌트들을 최적화하고 새로운 기능을 추가하여 낮은 응답속도와 높은 처리량

(throughput)을 달성할 수 있도록 하였다. 우리의 블럭 입출력 서브시스템은 리눅스 커널 2.6.32 기반으로 구현되었다. 실험 결과에 따르면, 동기적인 입출력 경로 (SyncPath)의 경우, 단일 쓰레드 기반 워크로드에서 약 3.3배 정도 소프트웨어 오버헤드를 줄일 수 있었고, 이중 버퍼링 (2Q)의 경우 다중 쓰레드 기반의 워크로드에서 4.4배 정도의 처리량 향상을 볼 수 있었다. 또한 혼합 입출력 경로 디자인 (HTM)의 경우 입출력 요청의 접근 패턴이나 타입과 상관없이 저장 장치의 성능을 87%∼100% 까지 이끌어 낼 수 있었다. 제안된 블럭 입출력 서브시스템 디자인은 매우 일반적이기 때문에 차세대 SSD가 등장할 시점에도 효과적으로 적용될 수 있을 것으로 기대한다.

Keywords : I/O subsystem, Storage device, Latency, Throughput

Student Number : 2006-21228

감사의 글

지난 박사 과정을 돌이켜 보면, 어느 때보다도 바쁘고 힘든 시간이었지만 어느 때보다도 값진 시간이 아니었나 생각합니다. 연구의 매순간은 실험 결과에 대한 실망과 희망으로 뒤섞여 울고 웃었던 기억이 가득했지만, 이러한 경험들은 저를 성장시켰고 이제는 연구의 즐거움이 무엇인지 어렴풋이 알 수 있게 된 듯도 합니다.

무엇보다도 특히, 연구를 지도해 주신 염헌영 교수님께 깊은 감사를 드립니다. 언제나 도전적인 연구 주제와 최신의 연구 환경을 지원해 주셨고, 제가 연구 방향을 잃지 않도록 조언과 격려를 아끼지 않으셨습니다. 교수님의 지도 덕분에 연구 과정에서 부딪힌 많은 어려움들을 현명하게 극복할 수 있었고, 앞으로는 어떤 난관도 극복할 수 있을 것이라는 자신감을 가지게 되었습니다. 연구 논문 및 프로젝트 진행에 있어 완결성과 일관성이라는 소중한 가치를 가르쳐 주신 엄현상 교수님께도 깊은 감사의 말씀을 드리고, 바쁘신 와중에도 저의 박사 논문 심사를 위해 시간을 내어주시고 귀중한 조언을 해주신 민상렬 교수님, 유혁 교수님, 김지홍 교수님께도 이 지면을 빌려 감사의 말씀 드립니다.

제가 연구에만 집중할 수 있도록 항상 곁에서 보살펴 주시고 지원해주신 부모님께도 감사드립니다. 힘들고 지쳐 마음이 약해질 때마다 용기를 북돋아 주시는 아버지와 따뜻한 마음으로 공감해 주시는 어머니가 아니었다면 힘든 시간을 이겨내지 못했을 것입니다. 그리고 대학원 생활의 시작과 끝을 함께 하고 기다려준, 언제나 밝은 표정으로 믿고 따라와준, 소중한 제 아내 선희에게도 말로 표현하지 못할 감사를

전합니다.

같이 연구실 생활을 했던 소중한 인연들에 대해서도 새삼 감사의 말씀을 드리고 싶습니다. 연구실의 전통을 만들어 주신 임영 누나, 현주 누나, 영상 형, 형수 형, 정기 형, 종필 형, 현준 형, 효려 누나, 재욱이, 호섭 형, 성환 형께 감사드립니다. 그리고 기영, 형준, 윤기, 지현, 범모 형, 해욱 형, 용경 형, 경호 형, 정희 형, 승미, 창규, 보영, 성범, 운태, 승민, 완희, 민규 형, 계신, 국태 형, 윤희, 재우, 찬호, 세훈 형, 동유, 민영, 내영, 설웅 형, 신웅 형께도 감사드립니다. 특히 매번 창의적인 연구 주제로 토론하길 즐기셨던 동인 형, 자유로우면서도 균형 잡힌 시각을 가지신 형석 형, 연구 내적/외적인 모든 면에서 노련하셨던 은성 형, 미래의 연구 흐름 예측에 탁월한 감각을 가지신 혁 형, 언제나 가장 솔직한 의견을 내주시는 인순 누나, 아버지 같은 마음으로 모두를 이끌어 준 영원한 랩장 신규 형을 포함하여 많은 선후배님들에게서 많은 것들을 배울 수 있었습니다. 세세한 감사의 말씀을 적지는 못했지만, 모든 분들과 나누었던 추억들을 소중하게 간직하도록 하겠습니다.

마지막으로, 이제 늠름한 박사가 된 손자를 흐뭇하게 내려보실, 하늘에 계시는 할머니께 마지막 감사의 말씀을 전합니다. 할머니 소온~자!