Kernel Based Mechanisms for High Performance I/O

A thesis submitted in partial fulfilment of the requirements for the degree of

Master of Science

by

Evgeny Budilovsky

This work was carried out under the supervision of

Prof. Sivan Toledo

Tel Aviv University

Apr 2013

Acknowledgments

First of all, I would like to thank my advisor, Professor Sivan Toledo, for his amazing help during the work on this thesis and for contributing his effort and experience to guide me through the first steps in the research world. I would like to thank to my friend, Aviad Zuk, with whom we worked on the hinting mechanism. Thanks to Dmitry Sotnikov, Avishay Traeger, and Ronen Kat from IBM Haifa Research Lab for cooperation and help in research of I/O traces.

Finally I would like to thank my parents Sofia and Viktor, for encouraging me to pursue this degree and to my wife Liel for her love, care and support.

Abstract

This thesis deals with mechanisms that assist to improve I/O performance in Linux kernel. We focus on Linux kernel due to its enormous popularity and the openness of this platform. We begin by presenting the Linux I/O stack, the common storage protocols and the frameworks which are used to prototype storage devices.

Next, in chapter two, we describe clever host-assisted hinting mechanism that uses RAM on the host to compensate for the small amount of RAM within the Solid State Disk (SSD). This mechanism is one of the key contributions to paper which was submitted to the 4th Annual International Systems and Storage Conference. The mechanism is implemented as an enhanced SCSI driver (kernel module) and prototyped using emulation of SSD device.

Finally, in chapter three, we describe implementation of end-to-end I/O tracing tool which was born as a part of collaboration project with IBM Haifa Research Lab. This tool allows us to capture live I/O traces on Linux systems. Later we can use the collected traces to follow individual requests as they make their way through different layers of the Linux I/O stack. Working with this tool to generate and analyze traces, allows researchers to better understand various performance bottlenecks and assists them in finding ways to improve I/O performance on live systems.

Table of Contents

I Introduction and Background

We live in period of time in which I/O performance is extremely important to guaranty stability of computer systems and to allow them to serve millions of users throughout the world or within large organizations.

Many techniques applied to increase I/O capability of existing systems. Some of the techniques combine different hardware types to create monster machines which are capable to serve amazing amounts of iops. Good example of it is the recently announced Oracle's exadata X3 machine [1] that combines large amounts of RAM, flash and mechanical disks to create high performance capabilities. Other techniques [2] try to scale performance around large clusters of machines creating distributed file systems which benefit from the distribution of iops between many low quality machines. Finally we have techniques [3, 4, 5] which try to tweak the heart of the operating system (kernel) to build new efficient mechanisms for I/O improvement of the overall system.

This theses will focus on the third kind of techniques by exploring the structure of the I/O subsystem inside the Linux kernel and implementing solutions to some of the I/O performance related problems. Traditionally, modifying the kernel is considered hard because kernel mechanisms are delicate and require low level skills which are hard to acquire. Furthermore, a bug in the kernel can easily cause the operating system to crash, making debugging very challenging. Indeed writing kernel drivers is completely different experience then working on Firefox plug-in or Java application using conventional set of tools (IDE, debugger, compiler). However, doing kernel development today is much easier then several years ago. Virtualization technology was introduced to our life through products such as KVM, VMware and completely changed the rules of low level development [6]. Now instead of crashing hardware and kernel panics on our real hardware we can simply reset the virtual machine in which our code runs. We don't need to buy expensive equipment but can simulate many of the devices in software. All these technologies allow rapid prototyping of kernel solutions and make our life much less painful during the process of low level development.

1.1 The Linux I/O Stack

We start by describing the Linux I/O stack. The goal is to gain detailed understanding of how this important subsystem works. Later we will modify some aspects of it to gain I/O performance.

Figure1.1 outlines different layers of Linux I/O stack.

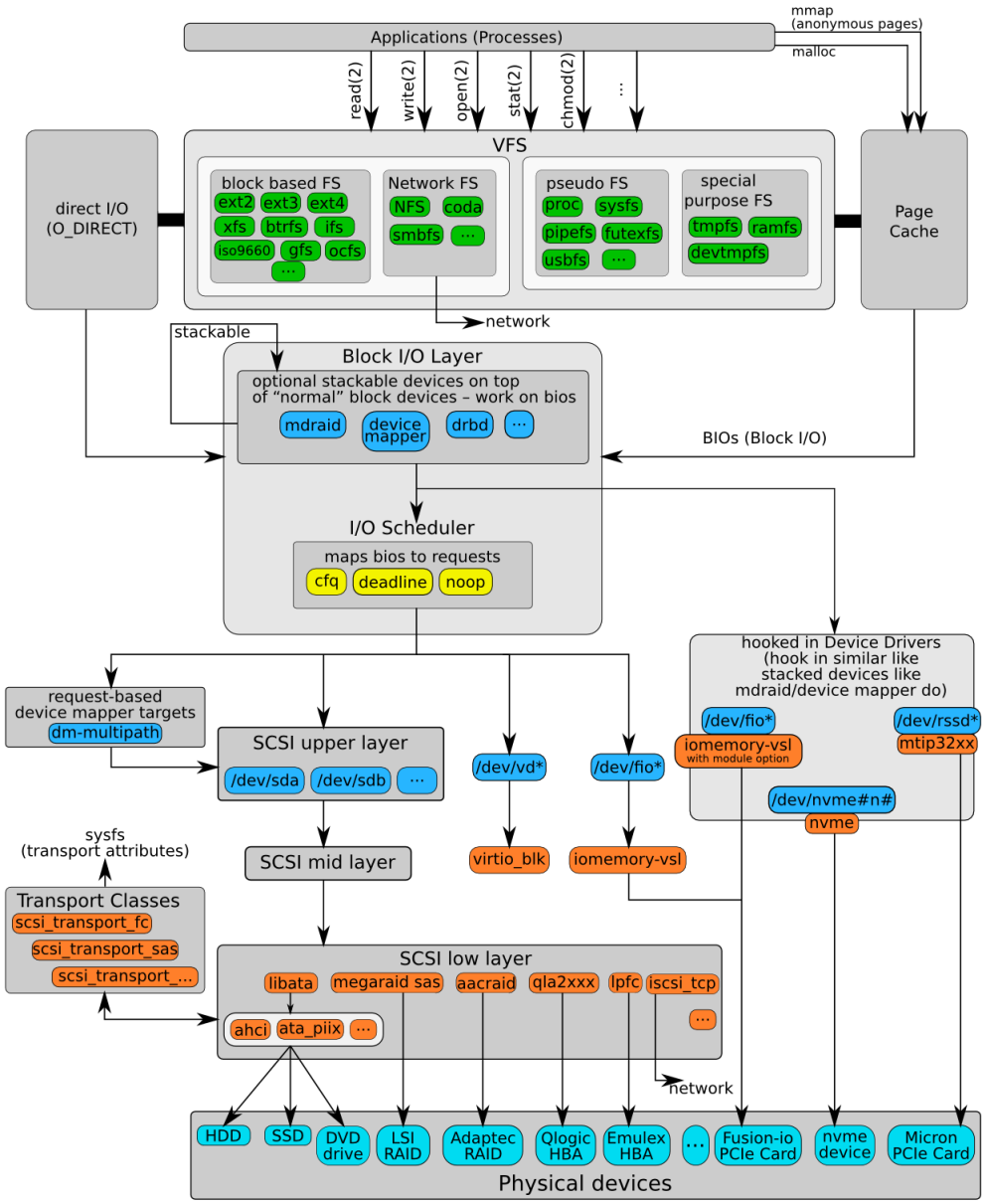

Figure 1.1: The Linux I/O Stack. The figure outlines Linux i/o stack as of kernel version 3.3 ***Figure created by Werner Fischer and Georg Schönberger and used under the Creative Commons License.***

1.1.1 The Application Layer

Files and directories are the basic abstractions that applications use to store and retrieve persistent data from storage devices. Linux kernel enhances this abstraction by introducing concepts of file system and I/O system calls. The file system encapsulates implementation details of storing and retrieving data. The system calls provide consistent interface to interact with file and directory objects.

To start working with files, user needs to build file system layout on top of storage devices (`mkfs`) and then to mount newly created file system somewhere in the hierarchy of the root file system (`mount`). For example `/dev/sda` is the special block device that represents first SCSI disk in the system. Using set of shell commands (1.1) the user creates new file system on the physical device and mounts it into the root tree. The root file system is the first file system which is mounted during the boot process at location ‘‘/’’. All other file systems can be later mounted somewhere in the hierarchy below ‘‘/’’.

After creating file system each application can access, modify and create files and directories. These actions can be performed using set of system calls which are exposed by the Linux kernel and are listed in Algorithm 1.2.

Algorithm 1.1 Creating a new ext4 file system on physical device and mounting it in.

```
# create file system
mkfs.ext4 /dev/sda

# mount new file system into the system root tree
mkdir /mnt/disk2
mount /dev/sda /mnt/disk2
```

Algorithm 1.2 Partial list of files, directories and I/O system calls.

```
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int unlink(const char *pathname);
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int readdir(unsigned int fd, struct old_linux_dirent *dirp,
            unsigned int count);
```

1.1.2 The Virtual File System Layer

To support different types of file systems Linux introduced common layer called the Virtual File System layer (VFS) [7]. This layer consists of set of objects and callbacks which must be implemented by any new file system. The VFS layer hooks into Linux I/O stack and redirects any I/O operation coming form user space into specific implementation of the current file system. The implementation of the I/O operation manipulates VFS objects and interacts with the block device to read and write persistent data.

The most important objects of the VFS layer are the superblock, the inode, the dentry, the file, and the address space. They are each explained in the next sections.

1.1.2.1 The Superblock

A superblock object, represented by the `super_block` structure, is used to represent file system. It includes meta data about the file system such as it's name, size and state. Additionally it contains reference to the block device on which the file system is stored and to the lists of other file system objects (inodes, files, and so on). Most of the file system store the superblock on the block device with redundancy to overcome corruptions. Whenever we mount file system the first step is to retrieve the superblock and build it's representation in memory.

1.1.2.2 The Index Node (Inode)

The index node or inode represents an object in file system. Each object has a unique id. This object can be regular file, directory, or special objects like symbolic links, devices, FIFO queues, or sockets. The inode of a file contains pointers to data blocks with file contents. Directory inodes contain pointers to blocks that store the name-to-inode associations. Operations on an inode allow to modify it's attributes and to read/write its data.

1.1.2.3 Directory Entries (Dentry)

The VFS implements I/O system calls that take a path name as argument (for example: `/root/docs/1.txt`). This path name needs to be translated into specific inode by traversing path components and reading

next component inode info from block device. To reduce the overhead of translating path component to inode, the kernel generates dentry objects that contain pointer to the proper inode object. These dentries stored in global file system cache and can be retrieved very quickly without additional overhead of reading from block device.

1.1.2.4 The Representation of Open Files

A file object represents file opened by user space process. It connects the file to the underlying inode and has reference to implementation of file operations, such as read, write, seek and close. The user space process holds an integer descriptor which is mapped by the kernel to the actual file structure whenever file system call performed.

1.1.3 Page Cache

The VFS layer operates on its objects to implement file system operation and eventually decides to read/write data to disk. Unfortunately disk access is relatively slow. This is why Linux kernel uses page cache mechanism to speed up storage access. All I/O in the system performed using page-size blocks. On read request, we search for the page in the cache and bring it from the underlying block device if it is not already in the cache. On write request we update the page in the cache with new data. The modified block may be sent immediately to the block device ("sync-ed"), or it may remain in the modified (dirty) state in the cache, depending on the synchronization policy of the kernel for the particular file system and file. Another advantage of the page cache is the ability to share same pages between different processes without the need to duplicate data and make redundant requests to block device.

The page cache is manipulated using `address_space` structure. This structure contains radix tree of all pages in specific address space and many address spaces can be used to isolate pages with similar attributes. Inode object contains link to `address_space` and to set of operations on address space which allow manipulation of pages.

Practically all I/O operations in Linux rely on page cache infrastructure. The only exception to this rule is when user opens file with O_DIRECT flag enabled. In this case the page cache is bypassed and the I/O operations pass directly to the underlying block device.

1.1.4 Block I/O Requests

Eventually the page cache layer needs to write/read pages from the block device. For this purpose special structure called `bio` created (see Figure 1.2). The structure encapsulates an array of entries that point to individual memory pages that the block device needs to read/write.

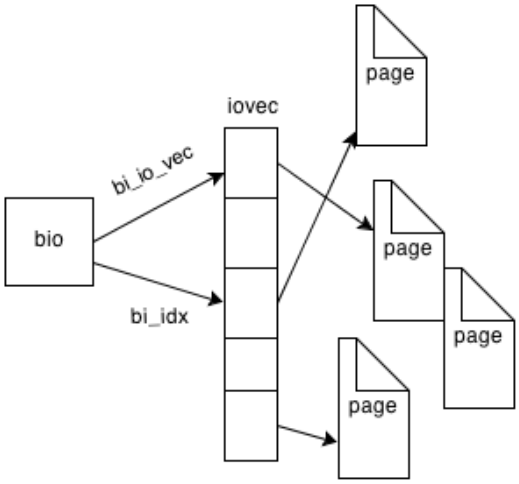

Figure 1.2: struct bio.

1.1.5 Requests and Request Queues

`Bio` structures are embedded into `request` structures that are placed on the request queue of the block device. The kernel rearranges and merges requests to achieve better I/O performance. For example, requests for adjacent blocks may be merged, or requests may be sorted by an elevator algorithm to minimize disk-arm movement, etc. These rearrangements are performed by an I/O scheduler; the Linux kernel provides several such schedulers.

1.1.6 The Block Device and the SCSI Layer

Requests placed on a queue are handled by a block device driver. In modern versions of Linux, the block device driver responsible for almost all persistent storage controllers is the SCSI driver. The SCSI layer in Linux is a large subsystem that is responsible to communicate with storage devices that use the SCSI protocol (which means almost all storage devices). Using this layer I/O request is submitted to the physical device in a form of SCSI command. When a device completes handling a request, its controller will issue an interrupt request. The interrupt handler invokes a completion

callback to release resource and return data to the higher layers (the file system and eventually the application).

1.2 The SCSI protocol

SCSI (Small Computer Systems Interface) has emerged as a dominant protocol in the storage world [8]. It is constructed from numerous standards, interfaces and protocols which specify how to communicate with a wide range of devices. SCSI is most commonly used for hard disks, disk arrays, and tape drives. Many other types of devices are also supported by this rich standard. The Linux kernel implements a complex SCSI subsystem [9] and utilizes it to control most of the available storage devices.

1.2.1 History

SCSI is one of the oldest protocols that still evolve today. SCSI came to the world around 1979 when computer peripheral manufacturer Shugart Associates introduced the "Shugart Associates System Interface" (SASI). This proprietary protocol and interface later evolved into the SCSI protocol, with SCSI-1 standard released in 1986 and SCSI-2 [10] in 1994. Early versions of SCSI defined both the command set and the physical layer (so called SCSI cables and connectors). The evolution of SCSI protocol continued with the SCSI-3 standard [11], which introduced new modern transport layers, including Fibre Channel (FC), Serial attached SCSI (SAS) and SCSI over IP (iSCSI).

Today SCSI is popular on high-performance workstations and servers. High-end arrays (RAID systems) almost always use SCSI disks although some vendors now offer SATA based RAID systems as a cheaper option.

1.2.2 Architecture

The SCSI protocol implements client-server architecture (Figure 1.3). The client called “initiator” sends commands to the server which is called “target”. The target processes the command and returns response to the initiator. Usually we see Host Base Adapter (HBA) device in the role of initiator and we have several storage devices (disk, tape, CD-ROM) which perform as targets.

Each target device can also be subdivided into several Logical Units (LUNs). The initiator starts by sending REPORT_LUNS command which returns map of the available LUNS that the target exposes. Later the initiator can send commands directed to specific LUN. In this way single device can expose several functional units. For example a tape library which is used for data backup can expose a robotic arm on the first LUN and the tape devices which used to read/write cartridges on the rest of the LUNs.

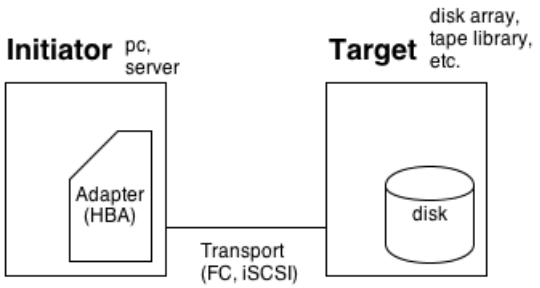

Figure 1.3: SCSI Architecture.

1.2.3 Commands

A SCSI command and its parameters are sent as a block of several bytes called the Command Descriptor Block (CDB). Each CDB can be a total of 6, 10, 12, or 16 bytes. Later versions of the SCSI standard also allow for variable-length CDBs but this format is not frequently used in storage systems.

The first byte of the CDB is the Operation Code. It is followed by the LUN in the upper three bits of the second byte, by the Logical Block Address (LBA) and transfer length fields (Read and Write commands) or other parameters. Example of CDB can be seen at Figure1.4.

At the end of command processing, the target returns a status code byte (0x0 - SUCCESS, 0x2 - CHECK CONDITION, 0x8 - BUSY and etc'). When the target returns a Check Condition status code, the initiator will usually issue a SCSI Request Sense command in order to obtain a key code qualifier (KCQ) which specify the reason for failure.

SCSI commands can be categorized into groups using data transfer direction criteria: non-data commands (N), writing data to target commands (W), reading data from target commands (R) and bidirectional read/write commands (B).

Other option is to separate SCSI commands based on the device they apply to: block commands (disk drive), stream commands (tape drive), media changer commands (jukebox), multi-media commands (CD-ROM), controller commands (RAID) and object based storage commands (OSD).

bits / bytes	7	6	5	4	3	2	1	0
0	op code 0x28							
1	LUN			DOA	FUA	Reserved		RelAddr
2-5	LBA							
6	Reserved							
7-8	TransferLength							
9	Control							

Figure 1.4: Read CDB (10 bytes length).

Examples of popular commands include: REPORT LUNS that returns LUNs exposed by the device, TEST UNIT READY that queries device to see if it is ready for data transfers, INQUIRY that returns basic device information and REQUEST SENSE that returns any error codes from the previous command that returned an error status.

1.3 Linux SCSI Drivers

The SCSI subsystem (Figure1.5) in Linux kernel is using a layered architecture with three distinct layers [12]. The upper layer represents kernel interface to several categories of devices (disk, tape, cdrom and generic). The mid layer is used to encapsulate SCSI error handling and to transform kernel request which come from upper layer into matching SCSI commands. And finally, the lowest layer controls a variety of physical devices and implements their unique physical interfaces to allow communication with them.

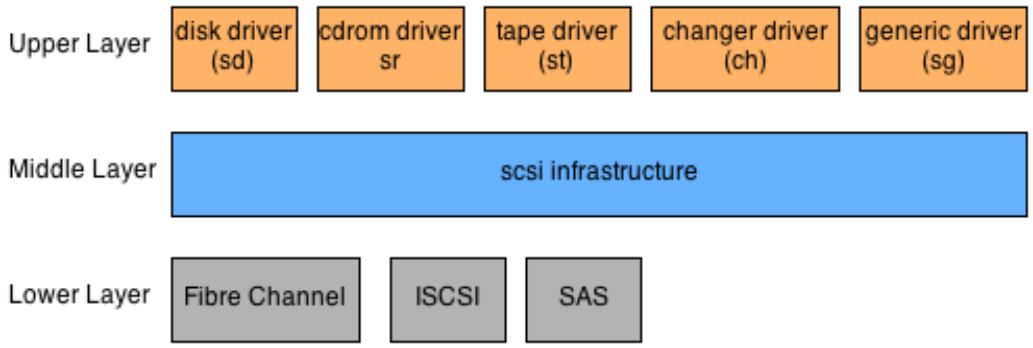

Figure 1.5: SCSI subsystem.

1.3.1 The SCSI Upper Layer

The purpose of the SCSI upper layer is to expose high level interfaces that introduce different families of scsi devices into user space. With these interfaces (special device files), user space application can mount file system on top of block devices or control SCSI device by using specific ioctl[1] commands. There are 5 main modules implemented in this layer. These modules allow to control, disk, cdrom/dvd, tape, media changer and generic SCSI devices.

1.3.1.1 Disk Block Device

The SCSI disk driver is implemented in `drivers/scsi/sd.c`. It registers itself as kernel driver which will handle any SCSI disk that kernel detects. Then when existing or new SCSI disk is detected by the kernel, the driver will create block device (usually called sda, sdb and so on). This device can be used to read/write to the disk by invoking `sd_prep_fn` which gets pending block request from the device queue and builds SCSI request from it. The request will be sent to the target device using the low level driver which controls specific storage device.

1.3.1.2 Cdrom/DVD Block Device

The SCSI cd-rom driver is implemented in `drivers/scsi/sr.c`. This is again block driver which works in similar fashion as the `sd.c` disk driver. Here again the `sr_prep_fn` will transform kernel block requests into scsi media commands which can be sent to the cd-rom device.

1.3.1.3 Tape Char Device

The SCSI tape driver is implemented in `drivers/scsi/st.c`. The tape driver is used to backup and restore data from sequential tapes, therefore it is implemented as character device. The `st_read` and `st_write` functions in this driver will build READ/WRITE CDB and dispatch it to the tape device. Other tape device commands such as ERASE (erase the tape) or SEEK (advance tape head to some position) implemented by the `st_ioctl` function which can be triggered by `ioctl` system call from user space.

[1] An ioctl interface is a single system call by which userspace programs may communicate with device drivers

1.3.1.4 Media Changer Char Device

The media changer driver is implemented in `drivers/scsi/ch.c`. This device used to control a media changer (robotic arm in tape library for example). It is very similar to the tape device and can be controlled by ioctl from user space which invokes `ch_ioctl` kernel function.

1.3.1.5 Generic Char Device

The generic SCSI device driver is implemented in `drivers/scsi/sg.c`. This is a catch all driver. It creates char device files with pattern `/dev/sgxx` to every SCSI device. Then it allows sending raw SCSI commands to the underlying SCSI devices using `ioctl` system call and specially crafted parameters. The popular `sg3_utils` package uses this driver to build extensive set of utilities that send SCSI commands to devices.

1.3.2 The SCSI Middle Layer

The SCSI middle layer is essentially a separation layer between the lower scsi layer and the rest of the kernel. It's main role is to provide error handling and to allow low level drivers registration. SCSI commands coming from the higher device drivers will be registered and dispatched to the lower layer. Later when the lower layer sends commands to the device and receives response, this layer will propagate it back to the higher device drivers. If the command completes with error or if the response doesn't arrive within some customizable timeout, retry and recovery steps will be taken by the mid layer. Such steps could mean asking to reset device or disabling it completely.

1.3.3 The SCSI Lower Layer

At the lowest layer of SCSI subsystem we have a collection of drivers which talk directly to the physical devices. Common interface to all these devices is used to connect them to the mid layer. On the other side the interaction with the device is performed by controlling physical interfaces using using memory mapped I/O (MMIO), I/O CPU commands and interrupts mechanisms. The lower layer drivers family contains for example drivers provided by the leading fibre channel HBA vendors (Emulex, QLogic). Other drivers control RAIDs (for example LSI MegaRAID) SAS disks and etc.

1.4 Linux SCSI Target Frameworks

Linux kernel has rich support for connecting to various SCSI devices. When using such device Linux needs to implement the initiator role of the SCSI architecture. More and more new SCSI devices were introduced to the Linux kernel throughout the years. This lead to rapid improvement in the layers that supported SCSI initiator behavior. On the other hand the other side of the SCSI client-server architecture was neglected for many years. Since SCSI target code was build into the physical devices (e.g. disk drives) there was no particular reason to implement this layer in the Linux kernel.

With the introduction of virtualization technologies things started to change. Developers wanted to create custom SCSI devices and to expose existing storage elements on the local machine through the SCSI protocol. These requirements lead to creation of several SCSI target frameworks with unique design and implementation. Of these frameworks, LIO was eventually selected and on January 2011 [13] was merged into the kernel main line, effectively becoming the official framework for introducing SCSI target support into the Linux kernel.

1.4.1 SCST

SCST target framework [14] is one alternative to the main line LIO framework. It (See figure1.6) was developed with main focus on performance. Since it is not present in the main kernel, the framework can be used by applying set of kernel patches to specific versions of Linux kernel. The patches introduce three groups of modules:

- SCSI engine - implementation of protocol independent engine to process SCSI commands
- Target drivers - These drivers connect to the different transport layers used to transfer SCSI commands (Fibre Channel, iSCSI, etc') coming from the initiator
- Storage drivers - These drivers allow to expose local SCSI devices, disks that use files as their back-end and custom SCSI devices which can be implemented in user space. The exposed target devices react to the initiator SCSI commands

The initiator (for example SCSI machine with emulex HBA) connects to the target where SCST is running (target can be another Linux machine with qlogic HBA which is connected to the first

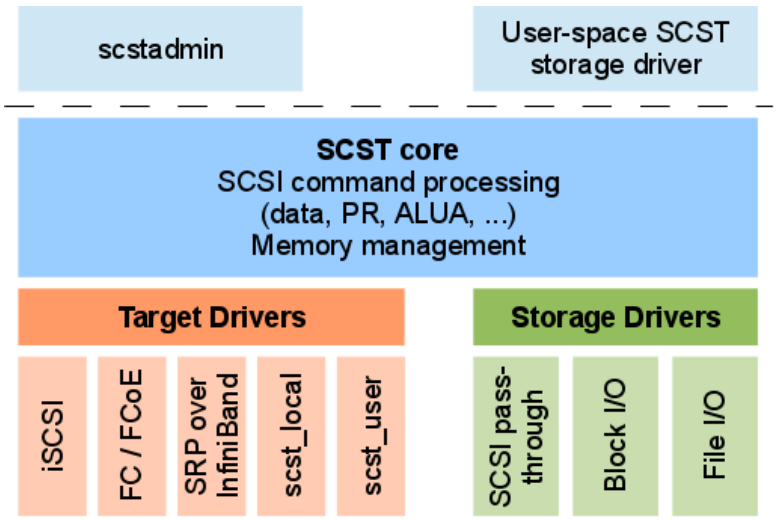

Figure 1.6: SCST Architecture.
(taken from wikipedia)

machine through fibre channel network). The target driver for qlogic extracts SCSI command and transfers it to the SCSI engine. SCSI engine interprets the SCSI command and forwards it to storage handler which emulates SCSI disk that stores it's contents inside regular file on the file system. The file I/O storage drivers reacts to the command and transfers data using the SCSI engine back to the target driver and through it to the initiator over the transport channel.

SCST proved itself as stable and fast infrastructure and was adopted by many storage companies which were interested in implementation of smart storage raids, De-duplication and backup solutions. Today it remains leading SCSI target framework even though it lost the battle of entering mainline kernel to the newly emerged LIO framework [15].

1.4.2 TGT

Unlike SCST framework which was focused on performance, the TGT framework [16] tried to focus on simplicity. To achieve this goal, small modification to the kernel code was used to forward SCSI requests from the kernel into user space. Various SCSI devices were implemented in user space including virtual disk, tape, cdrom, jukebox and object store device.

The TGT architecture contains three main components:

- Target drivers (kernel) - similar to SCST the target drivers is used to manage transport connection (for example fibre channel)

- TGT core (kernel) - This is a simple connector between target drives and user space daemon (tgtd). It allows driver to send commands to user space and forwards responses from user space back to the target driver.

- TGT daemon (user space) - Responsible for all SCSI commands processing. The daemon runs in user space and implements different SCSI devices. The devices can storage back-end to store data. This back-end can be some local SCSI device, file or memory.

Recently the kernel part's of TGT were removed from mainline kernel and replaced by the new LIO SCSI framework. Today TGT remains a pure user space project with iSCSI transport support in user space.

1.4.3 LIO

LIO target framework [17] is a recently new framework which was merged into the mainline kernel on 2011. The LIO framework provides access to most of the data storage devices over majority of the possible transport layers.

The architecture (See figure1.7) of the LIO consist of three main components:

- SCSI target engine - Implements the semantics of SCSI target without dependency of fabric (transport layer) or storage type

- Backstores - Backstore implements the back-end of the SCSI target device. It can be actual physical device, block device, file or memory disk

- Fabric modules - Fabric modules implement the front-end of the target. This is the part that communicates with initiator over transport layer. Between the implemented fabrics: iSCSI, Fibre Channel, iSER (iSCSI over infiniband) and etc.

Today LIO is a leading framework which replaced other frameworks in the kernel main line. Nevertheless, time still needs to pass until the framework will become mature and stable for everyday use.

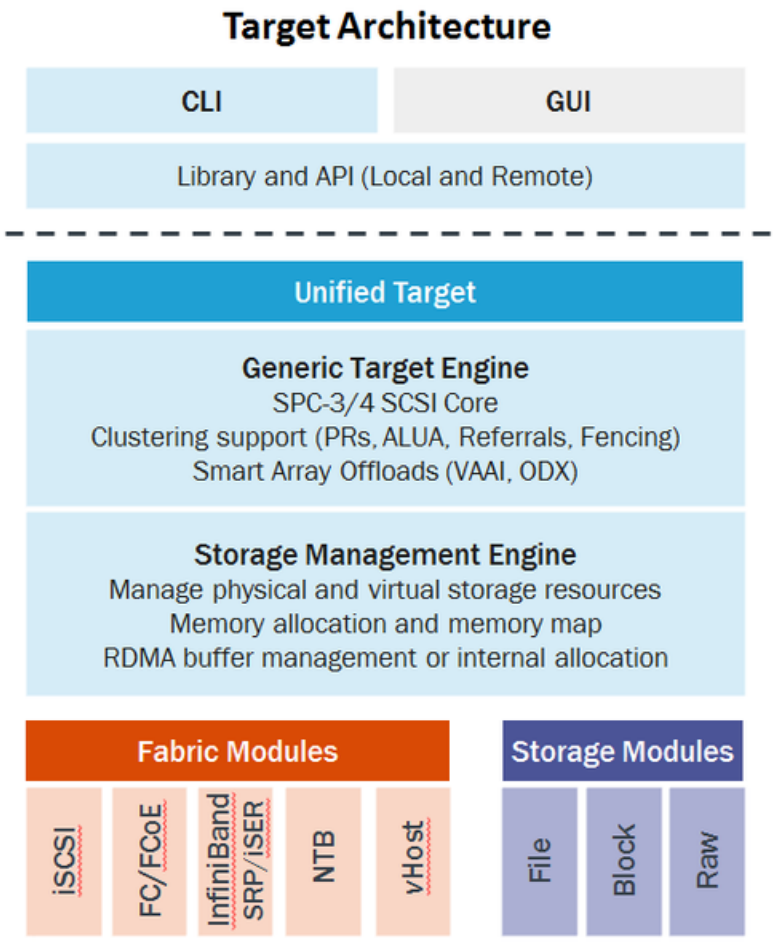

Figure 1.7: LIO Architecture.
(taken from wikipedia)

II Host Assisted Hints for SSD Device[1]

Solid state disks became very popular in recent years. They provide exceptional performance and endurance compared to the traditional hard disk drives. Nevertheless one factor delays their quick adoption - SSD's cost per capacity is is much higher compared to the conventional HD's.

In this chapter we explore solution to the cost problem by reducing the RAM size needed by the solid state disk. A small RAM lowers the cost of SSD but reduces the performance and endurance of the device. The proposed solution makes it possible to construct SSD devices with a little amount of embedded RAM, nevertheless the performance and the endurance of the devices won't be damaged.

2.1 SSD Background

SSD device essentially contains two key elements: controller and flash memory [19]. The controller is responsible for bridging of the NAND flash memory components to the host computer. On the front-end it connects to computer bus using SATA, SCSI or other interface. On the back-end the controller manipulates flash memory chips.

The task of working with flash chips becomes complicated due to some limitations of the flash memory [20] which support limited number of erasure cycles and no fine grained granularity when overwriting data. Therefore the controller implements complicated layer of software called flash translation layer (FTL) whose responsibility is to manage the flash chips and overcome the hardware limitations of the flash technology.

2.1.1 How NAND Flash Works

Flash devices can be made out of NAND and NOR technology. NAND based flash devices have emerged as a more acceptable candidate in the storage market and today almost all SSD's use NAND

[1] This chapter is based on article published in the the 4th Annual International Systems and Storage Conference [18]. My main role in the project was the implementation and testing of the hinting mechanism in the Linux kernel and in the simulated storage device, using TGT SCSI target framework.

technology.

The low level interface used to access flash memory chips differs from other memory technologies such as DRAM, ROM and EEPROM. Non flash memory technologies usually support changing bits from 0 to 1 and vice versa and provide random access.

NAND memories are accessed similarly to block devices. The memory area is broken into blocks. Each block consist of a number of pages (for example 128 pages for block). Pages are typically have size of 512 bytes, 2K or 4K. In the NAND technology the read and programming (changing bits from 0 to 1) performed on a page basis.

One limitation of the NAND memory is that although it can be read or programmed in a random access fashion (using page granularity), it can only be erased a block at a time. The erasure operation will set all bits in the block to 0 and allow new cycle of programming. Another limitation of the flash is that the number of erasure cycles is limited and beyond some threshold the block can not be erased any more (memory wear).

With these limitation a software layer needs to be applied to overcome hardware limitation. This software layer (FTL) has two main functions. First it creates abstraction of continues range of pages which can be read, written and overwritten just like in usual hard disk device. Second, it manages arrangement of data so that erasures and re-writes are distributed evenly across the memory.

2.1.2 SSD Performance Depends on RAM Size

SSD is a purely electronic device with no mechanical parts, and thus can provide lower access latencies, lower power consumption, lack of noise, shock resistance, and potentially uniform random access speed. However, one of the major problems with SSD is the relatively poor random write performance.

The flash memory technology requires erasure of complete block of pages before one of the pages in the block can be overwritten. On the other hand SSD provide abstraction of logical blocks array to the outside world. In this abstraction the os can randomly modify any logical block whose size is 512 bytes. To maintain this abstraction the FTL needs to keep mapping of the logical block into the physical location on the flash chip. When write operation happens, the FTL needs to find some block which was previously erased (or to erase new block) and to copy the newly modified page into it

(read-modify-write). Now it could seem as natural to build this mapping table at the granularity of one entry per logical block address (LBA). But then even typical 128G SSD disk will require (assuming logical block size of 512 bytes) 256 million entries with each entry taking several bytes of information for index and meta data. This amount of memory is simply to expensive for embedded controller to have. If we reduce the size of the mapping so it can enter affordable size of ram, then we will need to relocate (read-modify-write) larger chunks of data when part of the chunk is modified. This will reduce the performance of the device (we need to wait until the data is relocated) and increase the wearing of individual blocks which will be relocated frequently.

To overcome the trade-off between the cost of RAM and the performance of SSD device [21], many smart techniques were applied [22, 23, 24, 25, 26]. In this chapter I will describe host-assisted hinting mechanism that uses RAM on the host to compensate for the small amount of RAM within the SSD. This mechanism is implemented as an enhanced SCSI driver (Linux kernel module). It allows great flexibility to the designer of the FTL layer which can now achieve better performance while using moderate amount of RAM.

2.2 Host-assist Mechanism

When we look on a typical SSD device system (disk connected to some desktop or server). We can quickly spot the asymmetry between the SSD embedded controller with weak CPU and limited resources and the powerful PC with great deal of ram and large amount of computing resources available to it. Building some kind of cooperation interface between the two sides can help us to solve our problem of insufficient memory resources inside the SSD device.

Since SSD device connects to the host using some storage interface such as SCSI, SATA or USB. We decided to exploit this interface to generate an out-of-band channel that is not used for the normal read/write requests. This host-assist mechanism allows the host to assist the SSD in various tasks, and in particular in mapping sectors.

When the SSD receives I/O request from the host it needs to perform some mapping manipulation (read some mapping entries from the flash memory or modify existing mapping). At the end of this operation the SSD device will send back mapping hint. In general hint is a saved result of some computation which needs to be cached. Unlike regular cache entry the hint might contain wrong

information and so the user of the hint must have a way to check it's correctness (without repeating the computation of course). Hinting is an old caching technique used to speedup computations [27]. Most of the time the hint will be correct and no repeating computation will be performed which will lead to increase in performance.

In our case the SSD device will send hint with mapping computation result related to particular LBA. Th host part of host-assist mechanism will cache this hint. Next time when the host issues I/O request with LBA matching one of the hints stored in the host, it will send the hint using out of band channel and then invoke the I/O request itself. The SSD then can validate the hint information and use it to process incoming request without overhead of mapping computation.

Together with Aviad Zuck and Sivan Toledo we prototyped this idea [18]. Sivan and Aviad were working on new high-performance FTL for SSD. One of the design elements for this translation layer was to store the mapping between logical sectors and the flash pages in two level structure. The first level of the mapping is sitting in small data structure in the SSD's RAM and the second layer is stored on the flash memory. For each I/O request the STL layer needs to locate the actual physical address on the flash chip. This is done by using the first layer of the mapping in the RAM to locate the second layer on the flash chip. Then reading mapping entry from flash memory where second layer of mapping is stored. The host-assist mechanism allows to avoid the extra read from flash overhead, effectively increasing the performance of the FTL layer.

We have prototyped the mechanism over iSCSI. The SSD, which we prototyped using the TGT user space SCSI target framework [16], exposes one disk LUN and one custom LUN that is used for the assist mechanism. The host sends hints to the SSD by writing to the custom LUN. The host also keeps at least one read request pending on this LUN at all times, allowing the SSD to send mapping chunks to the host whenever it chooses. When the SSD reads or modifies a mapping chunk, it sends the chunk to the host as a response to the outstanding read request on the custom LUN. On the host, a kernel thread receives the chunk and caches it. This cache of chunks on the host is used by an interceptor function. The interceptor is invoked by the SCSI driver on every request to the SSD. For every request directed at the hint-enabled SSD, the interceptor attempts to find relevant mapping chunks. If it finds any, it writes them to the custom LUN, concatenated with the original read or write SCSI request. This constitutes a hint. The interceptor then sends the original request to the data LUN. If the host can't find the chunk, the interceptor sends only the original request. This structure is illustrated in

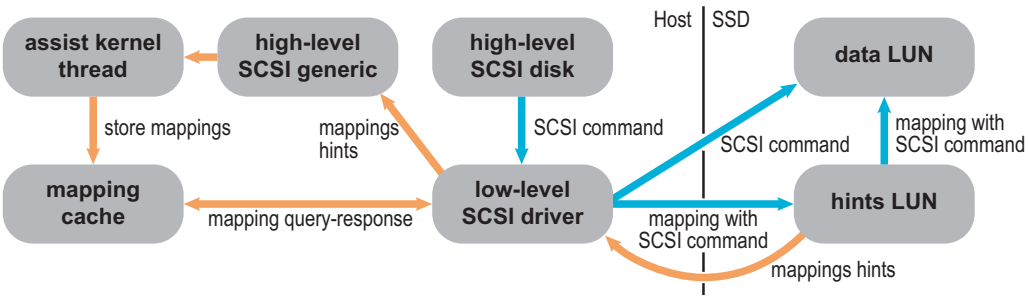

Figure 2.1: The hinting mechanism.

figure 2.1.

When the SSD receives a hint before the actual read/write request, it uses the mapping in the hint instead of reading the mapping from flash. This saves one read latency for most random reads and writes. To work correctly without coupling the SSD and the host too tightly, the SSD attaches a version number to chunk pointers in the root array and to the chunks sent up to the host. It uses a chunk from the host only if its version number is up to date; otherwise it ignores it. The hints are small relative to the data pages; therefore, they do not consume much bandwidth on the SCSI interconnect. If the SSD is used with an existing SCSI driver (on any operating system), the driver will not attach to the custom LUN, will not read mapping chunks, and will not send hints. Random reads and writes would be slower than on a host that implements hinting, but the SSD would work.

2.3 Implementation of the Hints Mechanism

Implementation of the hint-assist mechanism is distributed into two parts. First part is a Linux kernel module that can be loaded on the host. When loaded the module detects SCSI devices (attached locally or through some remote protocol such as iSCSI or fibre channel). When one of the devices contains special hints LUN, the kernel module will start listening on this LUN for incoming hints. It will also intercept any I/O request coming on other LUN's. The hint mechanism will search for hints associated with the LBA of the I/O request and send any hint as an I/O WRITE request to the hint LUN.

Second part of the mechanism is implemented in user space. We use SCSI target framework TGT [16] to emulate hint device that injects hints to the host and receives them back when appropriate. We also implement emulation of SSD SCSI device which integrates with code written by Aviad that perform read/write requests to the FTL layer. The SSD device is co-designed with the hint device by

using hints device interface to receive and inject hints. This way the SSD device offloads mapping entries to the host RAM and receives them back together with relevant I/O request.

For evaluating our solution we use iSCSI transport layer. Linux kernel support SCSI initiator and the TGT framework implements the target side of the protocol. And so our virtual devices exchange SCSI messages over iSCSI transport. The whole system runs inside vm executed by some virtualization technology such as KVM or VirtualBox.

2.3.1 The Linux Kernel Module

The `ssd_hint` kernel module consist of three parts:

- **hint_device** - This is the module which provides api to communicate with hint virtual device. The `hint_device_recv` and `hint_device_send` functions use SCSI upper layer in the kernel to send read/write requests to the hint device. These functions insert the request into the device requests queue and wait for it's completion. Since sending hints to the device needs to happen immediately and if possible before the relevant I/O reaches the device, we provides additional function called `hint_device_send_fast`. This function skips the upper and the mid layers of SCSI subsystem and sends SCSI command immediately through the low level SCSI layer which controls the low level adapter capable to push messages into the transport layer (iSCSI, Fibre Channel, etc.)

- **hint_manager** - This part implements hint assisting logic. It has two handlers which are invoked on different events. The `hint_manager_on_scsi_request` handler intercepts I/O request directed to the SSD device. It checks that the SCSI command is READ/WRITE command and then tries to find matching hint and send it to the hint device before the real command arrives to the SSD device. The `hint_manager_on_idle_time` is invoked from kernel thread which runs in background. This function tries to receive new hint from the hint device and to store it for future uses. This way we have a kernel thread dedicated to receiving hints so that when the SSD device decides to send us hint we will be ready for it.

- **hint_storage** - This module stores the hints. And allows using LBA ranges as key to search for existing hints and to retrieve them.

Finally we have the main routine which is invoked when the kernel loads hints module. The routine uses kernel api function `scsi_register_interface` to enumerate all SCSI devices in the system. For each SCSI device we get reference to it's state (struct `scsi_device`). We check if this is indeed SSD device and follow pointers in the kernel structure to get the state of the underlying transport layer (struct `scsi_host_template`). This state is essentially collection of callbacks which will be invoked by the kernel in attempt to send SCSI request. We replace one callback called `queuecommand` with our one version. This hooking method allows us to intercept every SCSI request that is directed to the SSD device and to apply our hinting logic before executing the request. Finally when we detect a hint device (which declares itself with special type), we initialize the hinting logic and start to listen for incoming hints.

2.3.2 The User Space Part

The TGT framework is based on two types of templates which can be used to implement custom SCSI devices.

- **device_type_template** - Contains set of 256 callbacks (one callback for each of possible SCSI operation codes) and some additional initialization and destruction callbacks used to configure the emulated device. The TGT framework already implements wide range of SCSI devices (disk, tape, raid, osd and so on). We implement two additional devices: ssd and hint device.

- **backingstore_template** - This template used to create abstraction layer for the device to store it's data. The idea is that each device can be configured to store it's persistent data on physical disk, file or at ram. For out implementation we don't use this template, since the hint device stores all it's information in memory and the SSD device integrates with FTL layer which has it's own custom storage implementation.

For the hint device we implement only READ and WRITE SCSI commands and use default implementations for the rest. The hint device allows other SCSI devices to register for hints using `hint_register` function. It also has one queue for pending hints that needs to be sent to the host. Each registered device can invoke `hint_send` function which will add new hint into the pending hints queue. When host initiator send read SCSI command to the hint device the hint will peek one hint

from it's pending queue and transfer it to the host. On the other hand when host invokes write SCSI command. The hint device will enumerate all registered device and invoke `notify_hint` callback which was provided by each device to transfer incoming hint to them. This simple mechanism allows exchanging hints with the host.

The SSD device registers itself to the hint device in order to receive hints. It then implements READ-/WRITE and other SCSI block device commands using interface into the FTL layer which was implemented by Aviad. The FTL layer sends hints to the host after each I/O request using the `hint_send` api. This allows SSD to reserve less memory and to increase performance by quickly resolving logical to physical mappings using the provided hints.

2.4 Evaluation of the Hints Mechanism

We performed two main types of experiments with the SSD prototype. Both experiments used the same setup. A VirtualBox virtual machine ran a Linux kernel into which our hinting device driver was loaded. The SSD prototype ran on the same machine under TGT and it exported an iSCSI disk. The iSCSI disk was used either as a block device by a benchmark program or the ext4 file system was mounted on it. The SSD was configured to include 8 NAND flash chips connected through 4 buses. The total capacity was kept small, 4GB, to allow us to run experiments quickly. In this configuration the amount of RAM required by our controller simulator was less than 1 MB. The small total size of the SSD should not make a significant difference in our experiments. In one set of experiments, we mounted a file system on top of the SSD block device and ran Linux applications from that file system (mainly a kernel build). We repeated the experiments with and without hints. This verified that the prototype works correctly. The main set of experiments is designed to evaluate the performance of the SSD. Each experiment starts by filling the block device sequentially. This brings the SSD to a state in which it must perform garbage collection. We then run vdbench [28] to generate block-device workloads: random or sequential reads or writes. Each request is for 4 KB, or one flash page. During the run of vdbench, the SSD prototype counts the number of pages written to or read from flash, the number of erasures, and the number of bytes transferred on NAND buses. We count separately operations that are triggered by SCSI requests and operations that are triggered by garbage collection. When the SSD writes pages of mapping chunks, the pages include both SCSI-triggered chunks and garbage-collection triggered chunks. We consider all of these page write operations to be SCSI related

(this makes our results seem a little worse than they really are). The metric that we use to evaluate different SSD configurations is the average flash-chip time consumed by each SCSI operation. We compute this average by multiplying the number of pages read by amount of time NAND chips take to read a page, the number of page written by the program time, and so on, and then divide by the total number of blocks requested through the SCSI interface. The graphs below also show a breakdown of the average time to different components. We repeated the main set of experiments 5 times, and the results presented here are the averages. The results were stable, and the largest variance exhibited between the runs was 1.5%.

Figure 2.2 presents the performance benefits of using host-assisted mapping. The time to both read chunks is significantly reduced. This improves the performance of random reads and writes. In particular, the use of hinting with small mapping chunks brings the cost of every read and write close to the minimum dictated by the NAND flash timing; random reads and writes are as fast as sequential ones, and they take essentially no time beyond the time to transfer a page and execute the NAND operation on the flash chip. This is the most important result of this work.

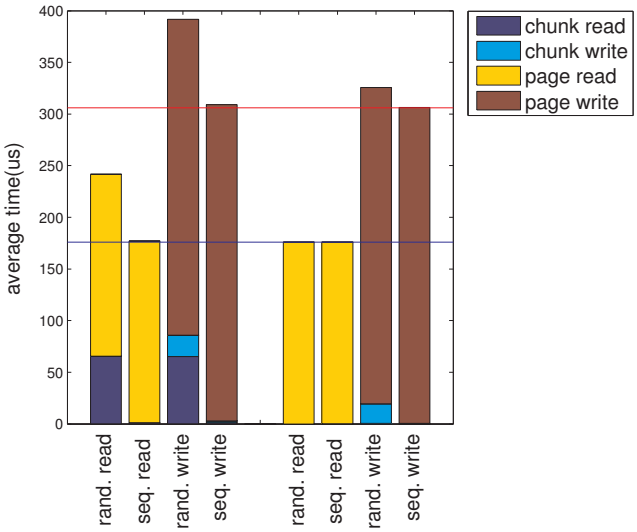

Figure 2.2: The effect of mapping hints on our SSD. The four bars on the left show the performance without hints, and the four on the right show the performance with hints. Each bar is broken into constituents. The average running times are estimates based on the timing of NAND flash operations and on the NAND bus throughput. We simulated each setup 5 times; the variations were 1.5% or less; this also applies to subsequent figures. The time to read hints almost disappears. The red horizontal line shows the cost of transferring a page from the controller to the NAND chip and programming a page. The blue line shows the cost of reading a page and transferring the data to the controller. With small chunks and hinting, performance is almost optimal for all access patterns.

Figures 2.3 and 2.4 explore the effects of hinting on flash devices with alternate timing characteristics.

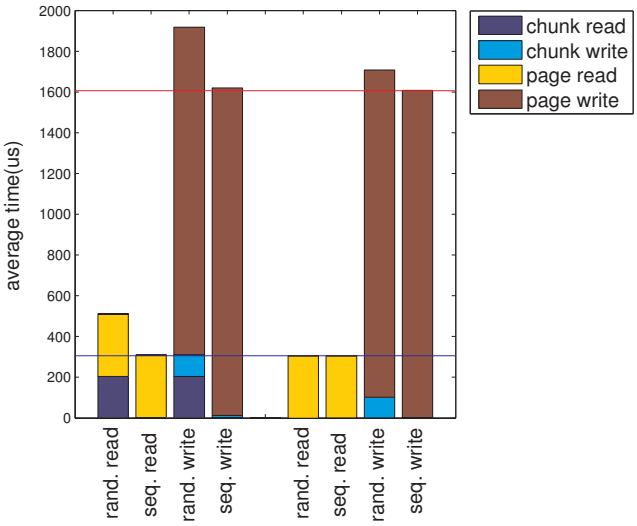

Figure 2.3: The impact of hints on our SSD when flash read and program times are longer (200μs and 1.5ms, typical of some 3Xnm MLC devices) but bus transfer times remain the same. The impact on writes becomes smaller, because of the long latency, but the impact on reads becomes larger.

Figure 8 shows that when flash read operations are relatively slow, which is the case on some MLC NAND flash chips, the impact of the hinting mechanism is even grater. Some 3Xnm chips take about 200us to read a page and about 1.5ms to program a page or erase a block. The bus transfer times are similar to those of older chips. On such chips, the high cost of page read operations makes hinting particularly useful in random-read operations; writes are so slow that the use of hinting does not make a big difference. Figure 2.4 explores a possible future enhancement to NAND flash chips, in which the bus timing improves. In this graph, we assume a hypothetical transfer time of 5ns per byte. Again, the use of hinting becomes more significant, because the faster NAND bus makes reading chunk pages relatively more expensive.

Figure 2.5 demonstrates the effect of limiting the capacity of hints stored on the host. We repeated our random write benchmark with varying amounts of storage allocated to the hints cache on the host. The amounts of hint storage ranged from 20% of the amount required to keep the full mapping on the host to 100% (about 3 MB). We then measured the percentage of SCSI requests that triggered a chunk read from flash, i.e. that were not preceded by a useful hint. The data shows that there is a clear and simple correlation between the effectiveness of the hinting mechanism and the size of memory allocated in the host for storing hints. On workloads with more temporal or spatial locality, a small hints cache would be much more effective than on a random-access workload.

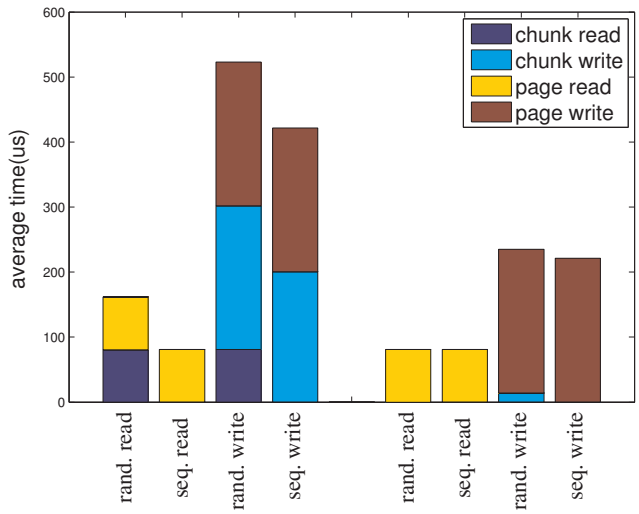

Figure 2.4: The impact of hints when bus transfer times drop to 5ns per byte.

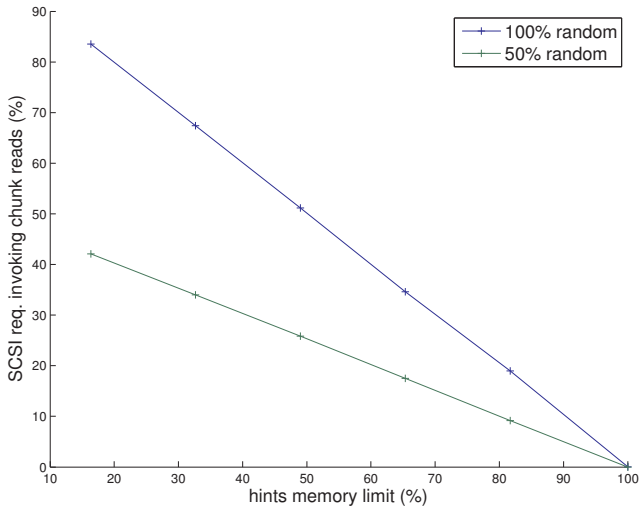

Figure 2.5: The effect of limiting the memory allocated for hints storage on the host-side device driver. The 100% point allows the entire page-level mapping to be stored on the host. The blue line displays the behavior in a purely random workload, and the green line in a workload where only 50% of the requests are random and the rest are sequential.

III End-to-End Tracing of I/O Requests[1]

3.1 Background

Traces of the input-output (I/O) activity of computer system are critical components of research efforts to improve I/O subsystems [29]. Traces record what applications or entire systems are doing, allowing researchers to replay the traces in order to measure the performance of proposed or modified systems. Looking at traces we can discover interesting patterns which can be used to build smarter storage systems [30]. Traces bring repeatability to I/O research and they can be used even when the entire environment in which they were generated cannot be duplicated in the lab (which could be due to privacy issues, cost, use of proprietary application software and so on). Recognizing the importance of I/O traces, Storage Networking Industry Association (SNIA) has established the Input/Output Traces, Tools, and Analysis (IOTTA) repository [31]. The primary goal of this repository is to store storage-related I/O trace files, associated tools, and other information to assist with storage research.

The I/O requests that we trace traverse a complex stack of software and hardware components: database systems, file systems, operating-system caches and buffers, interconnects and their device drivers, array controllers, SSD controllers, etc. Application performance is influenced by all of these components and by the interaction between them. However, almost all tracing is done at one particular point in the I/O stack, typically either the file system interface [32, 33] or the block-device interface [34]. We care about end-to-end performance, but we measure at one spot.

The work in this chapter is focused on building framework to collect end-to-end traces and to exploit them. In particular, we collect traces that allow us to follow individual requests as they make their way from the file system layer, through the Linux buffer cache and the I/O scheduler (the "elevator") to the block device. We collect end-to-end traces using a Linux kernel subsystem called SystemTap, which allows users to instrument the kernel dynamically. Taps at several points in the I/O stack allow

[1] This chapter describes the results of a collaboration with Dmitry Sotnikov, Avishay Traeger, and Ronen Kat from the IBM Haifa Research Lab. The work was part of a larger project which is outside the scope of this thesis; we focus on the tracing aspect, which was my contribution to the project.

us to perform end-to-end tracing that is collected on a separate machine and processed to associate events from different layers with one another. This chapter will explain the concept of end-to-end tracing, how we implement it (including challenges and solutions).

3.2 Design Alternatives

There are different ways to generate end-to-end traces in the Linux kernel. First we can simply modify kernel's I/O layers and enhance them with code that produces I/O traces. However this approach won't be scalable and would be hard to apply since it requires large kernel modifications and recompilation of large parts of the Linux kernel.

Another approach would be to try and modify data structures that are used to perform I/O in the Linux OS. The VFS layer could be extended to propagate inode number through the different layers and till the SCSI request which goes out to the physical device. Then we can place interception functions at these two locations (VFS and gate to physical device) and get traces for both file system and block level. This approach would not work as well because kernel uses many different data structures at different levels (BIO, block request, SCSI request and etc). Modifying them will require again kernel rebuild and might influence memory consumption of the system.

Finally it seems that there are tools already integrated into the kernel which can allow us to achieve what we need without too much effort. The SystemTap utility was build to simplify the gathering of information from running Linux system. SystemTap provides a simple command line interface and scripting language for writing instrumentation for a live running kernel and user-space applications. We will use this tool to collect end-to-end traces avoiding the pitfalls of modifying kernel and its data structures.

3.3 Systemtap

SystemTap is an open source tool which was created by consortium that include RedHat, IBM, Intel, Oracle and Hitachi [35]. It was created to allow kernel developers and regular users to investigate behavior of kernel internals in easy and scalable way.

The user of SystemTap needs to write scripts in c-like language. Each script describes list of event-handler entities. Where event could be any place where kernel can stop or a set of predefined events

such as timers that expire, start and stop of the script invocation and etc. The handler is a set of commands in c-like language which can modify data, aggregate statistics or print data.

The SystemTap script is invoked using `stap` utility which performs the following steps:

1. Translate the `.stp` script into c source code which can be compiled as Linux kernel module
2. Compile the c source code and create kernel module (`.ko` file)
3. Load the module into the kernel
4. The module runs and invokes the handlers associated with events. If any handler output some data, it is collected by the `stap` and printed to standard output.
5. When `Ctrl-C` key sequence is pressed or when one of the handlers invoke exit command, the `stap` unloads the kernel module and the program terminates

Typical SystemTap script can look as in example3.1:

- Line 1: Defines global variable. Variables types in SystemTap inferred at fist use. Variables in SystemTap can be scalars (numbers, strings), arrays, dictionaries or statistics. Statistics are special SystemTap structure. Data can be aggregated in statistics using `<<<` operator and later extracted using range of statistics functions such as `min`, `max`, `average` and etc.
- Lines 3-5: Define `probe` which is a keyword for events in system tap. Probe `begin` is a special event which happens once the script starts running. The handler for event is specified between the curly braces. In our case this just outputs the string to standard output
- Lines 6-8: Probe `end` is a special event which happens just before the script terminates.
- Lines 10-12: This time the probe specifies place in the kernel as a probe point for each the handler will be invoked. The `kfree_skb` is the name of special trace point inside kernel defined with this name. There are many such places which were chosen by kernel developers as strategic locations to gather information. Alternatively `kernel.function` can be specified instead of `kernel.trace`, to invoke handler when function is entered. The handler in our example

aggregates 1 to dictionary which is index using the location parameter. The special `$location` form allows using kernel variables that were defined at the scope of the function in which the probe occur.

- lines 14-22: This time the probe handler will be executed every 5 seconds using built-in timers. Each time we iterate over all locations inside the dictionary and print the sum of aggregated values (number of times the packet was dropped). Finally we delete the locations array, which basically empties it and prepares for next use. This way we discard aggregated statistics every 5 seconds and start over again.

Algorithm 3.1 Example script: Monitoring Network Packets Drops in Kernel

```
1  global locations
2
3  probe begin {
4    printf("Monitoring for dropped packets\n")
5  }
6  probe end {
7    printf("Stopping dropped packet monitor\n")
8  }
9
10 probe kernel.trace("kfree_skb") {
11   locations[$location] <<< 1
12 }
13
14 probe timer.sec(5)
15 {
16   printf("\n")
17   foreach (l in locations-) {
18     printf("%d packets dropped at %s\n",
19            @count(locations[l]), symname(l))
20   }
21   delete locations
22 }
```

The kernel technology behind SystemTap is called `kprobes`. Basically SystemTap can be considered as a high level layer over the `kprobes` which hides from the user the burden of using low level kernel interfaces. Essentially `kprobe` is a low level kernel api which allows to hook any place in the kernel. To allow this hooking the first byte of instruction in the hooking address is replaced by a breakpoint instruction appropriate for the specific CPU architecture. When this break point is hit, `kprobe's` logic takes over and executes the specified handler. After that the overwritten instruction is invoked

and the execution continues from the address after the breakpoint. Another type of `kbprobe` is called ‘‘`return probe`’’. It is used to invoke handler after completion of some function in kernel. This time when the function is called, the return probe gets the return address and replaces it with trampoline code. The trampoline code executes the handler and then resumes the execution at the return address of the original function.

3.4 Implementation

We use SystemTap to implement our end-to-end I/O traces. By placing hooks at strategic points in the I/O stack we are able to intercept I/O request at different level and to output traces to standard output. The two key places to keep our probes is the VFS layer (at the start of I/O system call) and at the bottom of the block layer. Just before the SCSI request is sent to the physical device. We make use of the inode number as an unique id which connects the VFS layer requests to the block layer requests. Our solution is implemented with SystemTap using the following probes:

- Probe `vfs.open` - This is the system call which is called to open file. We will extract the file path and it’s inode number. The handler will log the mapping to standard output.

- Probe `vfs.read` and `vfs.write` - These are the system call which are used to read/write data to file. The handler for these events will extract file system I/O information (time-stamp, inode, offset, size, I/O type and etc.)

- Probe `kernel.function("scsi_dispatch_cmd")` - This is the kernel function which is invoked to send I/O request to physical device. The handler will extract the block request structure. Then it will extract list of BIO structures which identify pages that used for this data request. Finally we will extract the inode number which is associated with each page. The handler will log the time-stamp, offset, size and list of inodes associated with request to the standard output.

- Probe **kernel.function("blk_update_request")** - This is the kernel function which is invoked on completion of the block request. This time we extract time-stamp and use offset and size to relate this completion to block request which was dispatched earlier.

The logs entries are printed to standard output with different fields separated with comas see figure3.1

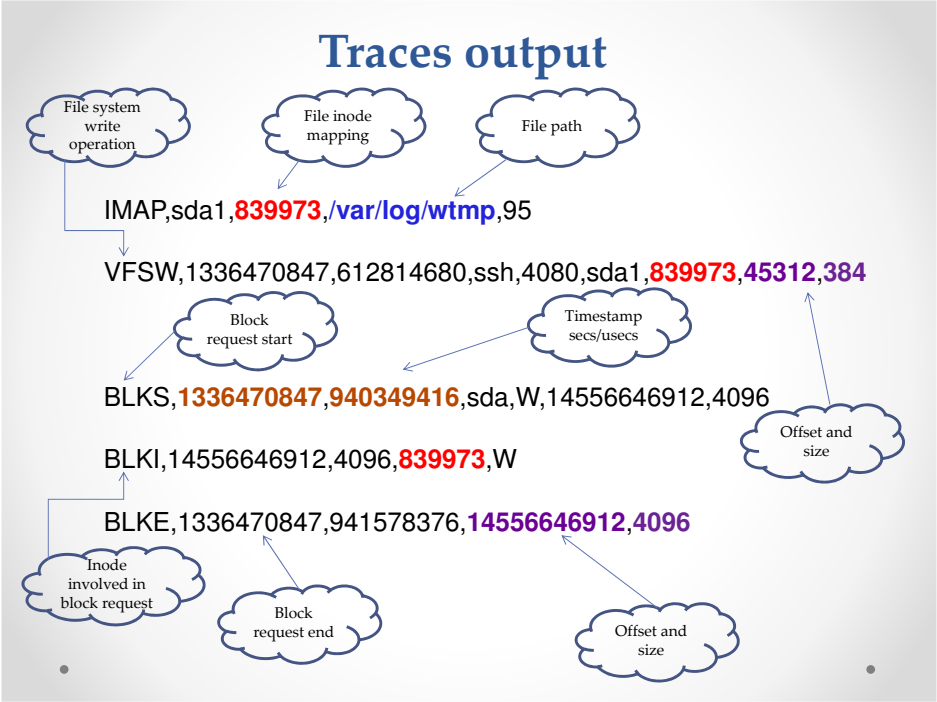

Figure 3.1: I/O traces entries

3.5 Information Extraction

Typical system for collecting ent-to-end traces consist from host and client. Host is the machine with I/O, it runs Linux and needs to have SystemTap scripts running on it. The traces are redirected by stap utility to standard output of the stap process. We pipe the standard output to network utility called netcat (nc) which sends all data over the network to client machine. The client machine is some dedicated PC running some operating system. The client connects to the port exported by the netcat on the host machine and collects all the logs. The logs are stored on the local disk and can be accessed and analyzed.

For analysis we use set of python scripts to dump log entries into database (we use embedded data base called SQLite). The python scripts generates four tables: `vfs_ops`, `block_ops`, `vfs_files` and `block_ops_by_file`. These tables allow us to extract information about I/O patterns on the host machines. For example we can ask what is the access pattern of specific file (random, sequential), what are the files with the most access operations (hot files), what is the level of fragmentation of file on disk and etc.

3.6 Experimental results

We performed several experiments with the end-to-end traces framework. In all experiments the setup consisted from two PCs. On one computer we installed the SystemTap scripts. During the experiment the SystemTap was running and redirecting it's output to `netcat` utility which was listening on port `9999`. The second computer connected to port `9999` with `netcat` and redirected all incoming data to local disk. The data was stored in files with max size limit of 100MB.

In one experiment we wanted to emulate typical user that uses his system and generates common I/O patterns. Unfortunately we didn't have live candidates so instead we used software called BOINC [36]. The BOINC project is a grid computing software. It exploits idle time on user's PC to perform computations that help to solve some important problems that requires great computational power. Basically millions of users around the world can share their computing resources to cure diseases, study global warming, discover pulsars, and do many other types of scientific research.

We installed BOINC client on PC which was running our tracing framework and allowed it to run for 19 days. The results were around 2G of traces logs. We dumped the traces into SQLite database and analyzed the I/O patterns using set of queries. Some quick facts we learned from the traces about BOINC software:

- There are total of 39426 files which were involved with IO during these 19 days
- The BOINC projects first performs sequential WRITES to files inside /home/user/BOINC/projects. This is the area where the data is first downloaded from the web and therefore the sequential access.
- Then most of the READ/WRITE performed inside /home/user/BOINC/slots/xx where xx is number (0,1,2, ...) This is the area of the BOINC framework. The access to files there is both random and sequential

In other set of experiments we tried to estimate the performance overhead of using I/O traces. For this we used simple dd command which allows to read/write data using variable block size. Before running the dd command we generated 1G file containing random data and flushed page cache layer using `sync ; echo 3 > /proc/sys/vm/drop_caches` command. Figure3.2 shows that when I/O

block size is around 1K and above there is no substantial overhead to the system I/O performance. However using small block size for I/O (Figure3.3) substantially reduces system I/O performance. This means that tracing system that uses inefficient I/O techniques (like writing file char by char without buffering) will introduce great overhead to the performance and might inflate the traces logs without control. Likely taking traces from live systems shows that such I/O patterns are not used often.

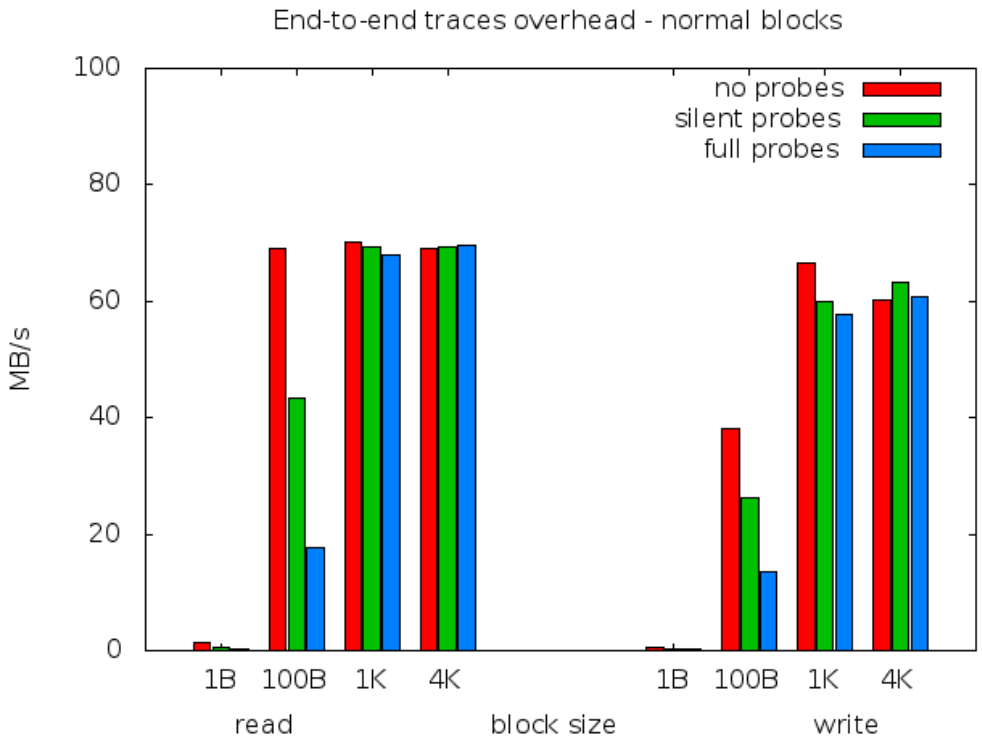

Figure 3.2: The graph show read/write performance using variable size of I/O block. We repeat the test when probes disabled, when probes enabled but do not write to stdout and when the probes write to stdout.

3.7 Pitfalls and How We Addressed Them

While using the end-to-end traces system we encountered several pitfalls. One of the problems was that after analyzing requests, some small percentage of VFS requests didn't match any block request and on the contrary some small percentage of block requests didn't match any VFS request. For example four days of I/O recording leads to the numbers displayed in table3.1

Block request that have no corresponding VFS requests can be partly explained by the fact that some application use `mmap` system call which maps files into user space process address space. Then the

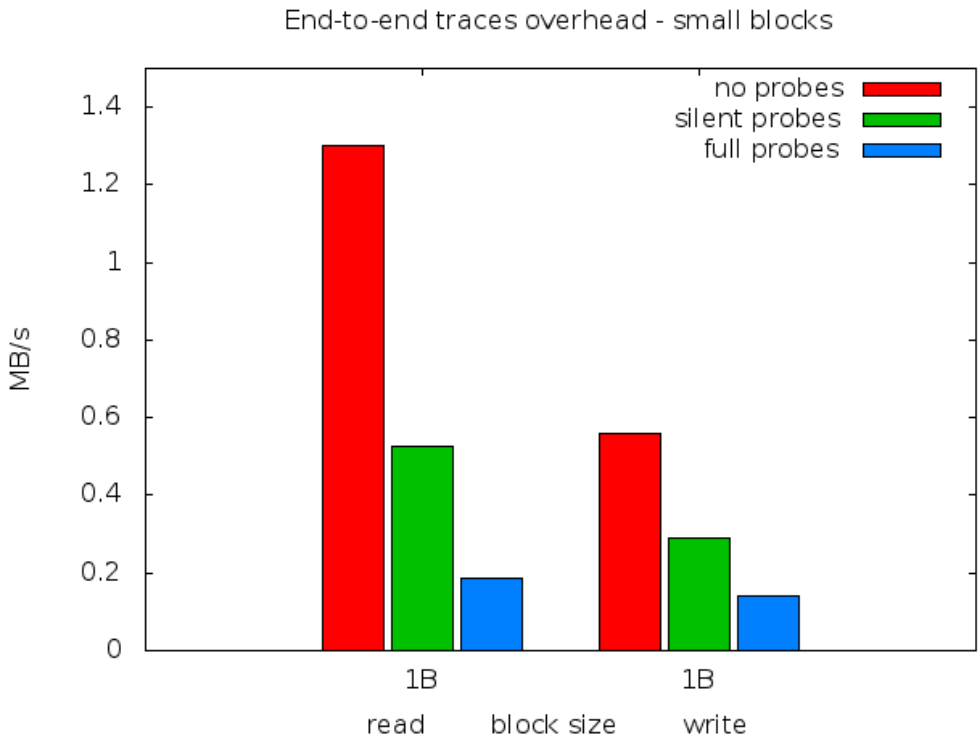

Figure 3.3: The graph shows read/write performance using I/O block of size 1 byte. This small block size emphasizes the overhead introduced by using SystemTap

	without match	total	ratio
block requests	24041	236866	0.10
vfs requests	48285	5135362	0.009
block files	743	13852	0.05
vfs files	156	13265	0.01

Table 3.1: For each block request we try to match corresponding VFS request and vice versa. The "without match" column, contains amount of block/VFS requests that didn't match existing VFS/block requests.

process works with the files as if they are plain memory. The I/O requests generated by these operations go down to the block device but there are no explicit read/write system calls we can catch. After modifying the traces probes to catch call to `mmap` system call we were able to associate most of these block requests with files that were opened using `mmap`.

Even after these actions some small amount of VFS and block requests (around 1-2% of total requests) were remaining without corresponding matches. After some investigation it seems these requests can be explained by the fact that we might loose some events when SystemTap logging kernel buffer becomes full. This can happen when we have high stress on the system. SystemTap reports amount of lost events through `sysfs` interface. Running `cat /sys/kernel/debug/systemtap/*/dropped`

will display the number of events. Experiments with writing to file using small size of blocks shows how more and more events are lost when the I/O block size becomes small and the number of log events increase in system (Table3.2).

block size (bytes)	lost events	total events	ratio of lost evens
1	4117673	4194304	0.98
32	75083	131072	0.57
64	9815	65536	0.14
128	0	32768	0

Table 3.2: We can observe that I/O with small block size leads to massive lost of events.

Bibliography

[1] Timothy Prickett Morgan. Oracle cranks up the flash with Exadata X3 systems. http://www.theregister.co.uk/2012/10/01/oracle_exadata_x3_systems/, October 2012.

[2] Frank B. Schmuck and Roger L. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 231–244, Berkeley, CA, USA, 2002. USENIX Association.

[3] Guangdeng Liao, Danhua Guo, Laxmi Bhuyan, and Steve R King. Software techniques to improve virtualized I/O performance on multi-core systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 161–170. ACM, 2008.

[4] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[5] WU Fengguang, XI Hongsheng, and XU Chenfeng. On the design of a new Linux readahead framework. *ACM SIGOPS Operating Systems Review*, 42(5):75–84, 2008.

[6] Nitin A Kamble, Jun Nakajima, and Asit K Mallick. Evolution in kernel debugging using hardware virtualization with xen. In *Linux Symposium*, page 1, 2006.

[7] Richard Gooch. Overview of the Linux Virtual File System. http://lxr.linux.no/linux+v2.6.38/Documentation/filesystems/vfs.txt, June 2007.

[8] Harry Mason. Standards: SCSI, the Industry Workhorse, Is Still Working Hard. *Computer*, 33(12):152–153, December 2000.

[9] Douglas Gilbert. The Linux 2.4 SCSI subsystem HOWTO. http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/, 2001.

[10] T10 committee. Small Computer System Interface - 2 (SCSI-2). http://www.t10.org/drafts.htm#SCSI2_Family, September 1993.

[11] T10 committee. SCSI Architecture (SAM). http://www.t10.org/drafts.htm#SCSI3_ARCH.

[12] M. Tim Jones. Anatomy of the Linux SCSI subsystem. http://www.ibm.com/developerworks/linux/library/l-scsi-subsystem/, November 2007.

[13] James Bottomley. Re: [ANNOUNCE] TCM/LIO v4.0.0-rc6 for 2.6.37-rc6. http://lwn.net/Articles/420693, December 2010.

[14] V. Bolkhovitin. Generic SCSI Target Middle Level for Linux. http://scst.sourceforge.net/, 2003.

[15] Goldwyn Rodrigues. A tale of two SCSI targets. http://lwn.net/Articles/424004/, January 2011.

[16] T. Fujita and M. Christie. tgt: Framework for storage target drivers. In *Proceedings of the Linux Symposium*, 2006.

[17] Nicholas Bellinger. LIO Unified Target. http://linux-iscsi.org, 2011.

[18] Evgeny Budilovsky, Sivan Toledo, and Aviad Zuck. Prototyping a high-performance low-cost solid-state disk. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 13:1–13:10. ACM, 2011.

[19] Kaoutar El Maghraoui, Gokul Kandiraju, Joefon Jann, and Pratap Pattnaik. Modeling and simulating flash based solid-state disks for operating systems. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 15–26, New York, NY, USA, 2010. ACM.

[20] Woody Hutsell, Jamon Bowen, and Neal Ekker. Flash solid-state disk reliability. *Texas Memory Systems White Paper*, 2008.

[21] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *ACM SIGOPS Operating Systems Review*, 41(2):88–93, 2007.

[22] S.W. Lee, D.J. Park, T.S. Chung, D.H. Lee, S. Park, and H.J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[23] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366–375, 2002.

[24] J.U. Kang, H. Jo, J.S. Kim, and J. Lee. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170. ACM, 2006.

[25] S. Lee, D. Shin, Y.J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, 2008.

[26] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. *ACM SIGPLAN Notices*, 44(3):229–240, 2009.

[27] B.W. Lampson. Hints for computer system design. *ACM SIGOPS Operating Systems Review*, 17(5):33–48, 1983.

[28] Henk Vandenbergh. Vdbench: a disk and tape i/o workload generator and performance reporter. http://sourceforge.net/projects/vdbench, 2010.

[29] Andy Konwinski, John Bent, James Nunez, and Meghan Quist. Towards an I/O tracing framework taxonomy. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, PDSW '07, pages 56–62, New York, NY, USA, 2007. ACM.

[30] Avani Wildani, Ethan L. Miller, and Lee Ward. Efficiently identifying working sets in block I/O streams. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 5:1–5:12, New York, NY, USA, 2011. ACM.

[31] SNIA. SNIA IOTTA Repository.
http://iotta.snia.org/traces.

[32] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: a file system to trace them all. In *Proceedings of the 3rd USENIX conference on File and storage technologies*, FAST'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[33] Eric Anderson. Capture, conversion, and analysis of an intense nfs workload. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 139–152, Berkeley, CA, USA, 2009. USENIX Association.

[34] Tao Huang, Teng Xu, and Xianliang Lu. A high resolution disk i/o trace system. *SIGOPS Oper. Syst. Rev.*, 35(4):82–87, October 2001.

[35] SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems. http://www.redbooks.ibm.com/abstracts/redp4469.html, Jan 2009.

[36] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

תקציר

במסגרת תזה זאת נעסוק במנגנונים שנועדו לשפר את ביצועי קלט/פלט במערכת הפעלה לינוקס. בחרנו להתמקד במערכת הפעלה זאת מכיוון שהיא זוכה לפופולריות עצומה ובעלת קוד מקור פתוח וזמין. בפרק המבוא נציג את מבנה שכבות הקלט\פלט בלינוקס, את הפרוטוקולים העיקריים המשמשים לאחסון מידע ושורה של ספריות עזר עיקריות שמשמשות ליצירה ודימוי של התקני אחסון שונים.

בפרק השני, נתאר מנגנון אשר מאפשר להתגבר על בעיית מחסור בזיכרון בהתקני דיסק SSD. המנגנון עושה שימוש מתוחכם בזיכרון הראשי של המחשב, על מנת לפצות על חוסר הזיכרון בדיסק עצמו. מנגנון זה היווה את אחת התרומות העיקריות למאמר שהתפרסם בכנס השנתי הרביעי של מערכות ואמצעי אחסון. המימוש של המנגנון נעשה ע"י כתיבת דרייבר ללינוקס וע"י אמולציה של התקן אחסון (SSD).

לבסוף, הפרק השלישי עוסק במימוש כלי ליצירת לוגים של מערכת קלט\פלט על לינוקס. הכלי נולד כחלק משיתוף פעולה עם מעבדת מחקר של יבמ בחיפה. הכלי מאפשר לתפוס אירועים חיים מתוך מערכת קלט\פלט ויחודו בכך שהוא עוקב אחרי כל אירוע כזה מתחילתו ועד סופו דרך כל שבכות הקלט\פלט של מערכת הפעלה. שימוש בכלי כזה מאפשר לחוקרים להבין טוב יותר את ביצוע מערכת האחסון ואת צווארי הבקבוק שלה. באופן כזה מתאפשרת לחוקר ההזדמנות לשפר את הביצועים ולתקן את המקומות בהם הביצועים ירודים.

אוניברסיטת תל-אביב

בית הספר למדעי המחשב ע''ש בלבטניק

מנגנונים מבוססי קרנל לשיפור ביצועי I/O במערכת הפעלה לינוקס

חיבור זה הוגש כעבודת מחקר לקראת התואר "מוסמך אוניברסיטה" במדעי המחשב

על ידי

יבגני בודילובסקי

העבודה נעשתה בבית הספר למדעי המחשב

בהנחיית פרופסור סיון טולדו

ניסן תשע"ג