Kernel Accommodations for I/O Intensive Workloads

Abstract

## 1 Introduction

This project was inspired by the disparity in performance of consumer CPU and consumer flash storage device. Historically, processors were the focus of technological advancement. We investigate some current limitations of existing hardware and paradigms of I/O. In doing so, we consider other paradigms for an Operating System to access data on persistent storage.

My project focuses on "wimpy" processors attached to higher performing solid-state disks (SSD's). Let "beefy" and "wimpy" refer to the power and speed of a processor. An example of a beefy processor might be an Intel Core i7, which has multiple cores on the same die, powerful processing features, and high clock speeds, but also greater power consumption. A wimpy processor, like the Intel Atom, may only have one or two cores, a slower clock speed, yet a lower power requirement. Also of interest is the rate at which a storage device can process commands, which is the number of I/O operations per second (IOPS).

## 2 Background

### FAWN

In the FAWN project, a Fast Array of Wimpy Nodes run a datastore program on a dual-core Intel Atom processors with an eye toward energy efficiency. Each node has persistent storage provided by Intel X25-E or M SSD's, but do not maximize the performance of the SSD, a type of flash storage. The goal of the project is to perform equivalent computation at comparable rates while requiring less energy. The metric of interest is operations per joule. In practice thought, the IOPS rate, or the number of I/O commands the device can perform per second has become a major bottleneck. A FAWN node requires high performance of random reads of small data of disk as the datastore program requires lookups into random locations in persistent data.

A primary motivating factor for using SSD's was to save on energy cost while at the same time attaining gains in performance.

| | **HD (SAS)** | **Flash (SATA)** |
|---|---|---|
| **Sequential Throughput** | < 200 | 220 |

| (MBps) | | |
|---|---|---|
| **Random Read IOPS (512B)** | 300 | 70000 |

The advantage of Flash storage devices, like SSD's, lies not in throughput, but in the ability to perform more commands. Rotational devices, with high spindle velocity, can have comparable throughput with SSD's. Even though in the general case, the throughput of rotational device cannot match that of the flash device, the difference is not nearly as staggering as that between the IOPS rate. The difference in this category is orders of magnitude greater, and along with the power consumption reduction, provides an ideal platform for storage on FAWN nodes.

NCQ

The Native Command Queue, NCQ, allows the storage device to store multiple commands. Originally, this technology was designed for rotational SATA drives which can perform the commands in a different order than they were received, optimizing the order in favor of reducing latency. This can be done by trading some seek time to reduce rotational delay. For SSD's, it is a mechanism that allows the storage device to continually process commands while the CPU is performing another task.

Conceivably, filling and completing commands on this queue in an appropriate manner could provide an optimal IOPS rate. The CPU could issue commands collected by the device in the NCQ, and simultaneously as the CPU performs another task, the SSD can fulfill those commands and interrupt the processor as necessary. Knowing how and when the processor should take the interrupt presents an incredibly difficult problem. Even though we have explored this space, it has not been fully solved.

[Possible Diagram]

AHCI Driver

The Advance Host Controller Interface is the primary interface for modern SATA devices. Developed by Intel, it is a relatively recent standard that only more recent iterations of major operating systems. Therefore, there are constant developments in programming the devices. Most programmers resort to open source drivers to reference since the documentation is so sparse, but even those are still evolving. For example, FIS-based switching support for Linux's AHCI driver was not in the mainline until 2.6.34, which was released on May 16th, 2010. Frame information structure (FIS) based switching allows the SATA port multiplier to negotiate more commands simultaneously to its connected devices. Part of my work has revolved around fully understanding the driver and using the driver to interface directly with the hardware.

## 3 Previous Work

Since these problems were not unknown to the members of the FAWN group, there has been work previously to improve the situation. The test platform is a single core Intel Atom processor with Hyper Threading and an attached flash storage device.

### I/O Polling

One of the more recent developments in the Linux kernel was I/O polling for block device drivers. All of the kernel's drivers can be separated into two categories, character and block device drivers. SSD's fall under the category of block devices because the kernel interacts with it in units of "blocks," which are generally akin to sectors on the device. The concept of I/O polling is not new, as it derives from network drivers that poll for incoming packets instead of taking interrupts. Normally polling is considered inefficient, but there are common cases where I/O workloads dictate greater performance when the driver polls.

### I/O Deferring

This work was based on the observation that the CPU is interrupted in an inefficient manner. I/O polling was created to change between interrupt and polling states. The intent would be that the driver is allowed the decision to ask the block layer to stop taking interrupts from the device normally and poll the device instead for I/O. It permits the driver, in times of increased load, to forgo the interrupt when it knows that there is I/O available to complete. In such cases, the driver will poll the device for work to complete that the device would normally interrupt the processor for.

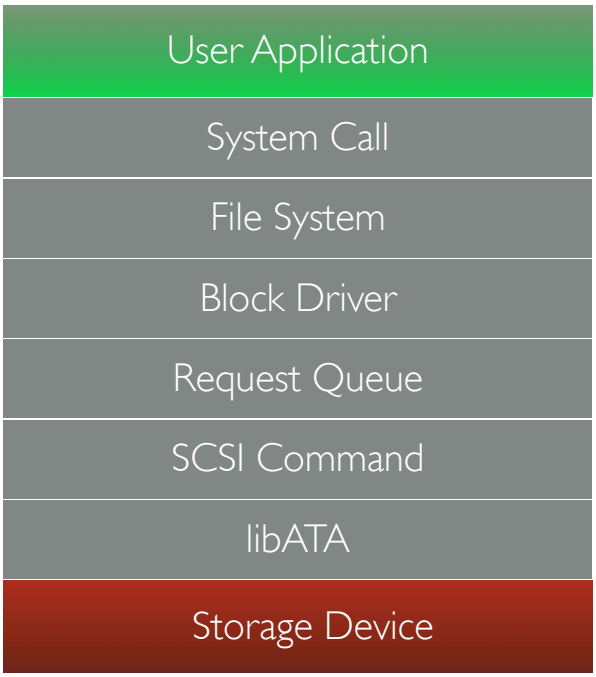
With the test platform, a problem occurs because the driver is constantly switching between interrupt and polling states, when it is unnecessary. It is suspected that the driver will see many commands completed in the NCQ, and thus enable polling because it expects a period of high traffic. Usually this is the case, for there will be commands waiting to be issued to the device, but the polling driver will be starved of work. In such a case, the driver reverts to taking interrupts, at which time the NCQ is again filled enough the driver wants to poll. Members of the group solved this be deferring interrupts for some static amount of time. This is set heuristically and is only changed manually. By deferring interrupts for some specified amount of time, it allows the wimpy processor to issue any awaiting I/O requests so that when polling or interrupting occurs again, the driver as a better concept of what load it is under. That is, if there is a large quantity of requests being issued, the NCQ should saturate and the driver should poll to forgo the unwanted interrupt latency. If there were no more requests, the correct path would be to return to interrupting state, which would occur when there are not a sufficient number of awaiting commands.

Randomness

Randomness is yet another example of inherent differences between SSD and rotational devices. Linux and other kernels will use command completion times to add to the entropy of the pseudo random number generator. Since SSD completion times are generally faster, thus the value is smaller. With lesser variability, the drive is less effective at adding to the entropy pool. The rub comes about when storage device randomness becomes the one of the primary sources of randomness, like on headless servers where input devices are absent. Then, the kernel relies even more on the remaining sources for entropy; without entropy, the kernel would fail.

## 4 Operating System Overhead

It is commonly accepted there exist some kind of software overhead in working with a kernel, but let us quickly categorize that by looking at the I/O stack for the Linux kernel.




Each of these layers represents some kind of abstraction or indirection for the preceding layer. Here are a few things of note, though.

## File System

In very specific cases, the user can directly access blocks with system calls like ioctl. Even though this bypasses at large amount of overhead since the user passes in the sector number of the block to read, it also bypasses the organization of the files and data maintained my the kernel and the file system. This could potentially be destructive and irrecoverable if crucial data is overwritten. Suppose the user had overwhelming knowledge about the organization of the device, which means any overhead created by the file system would occur in the user application. This does not necessarily reduce overhead, as the processing is deferred to the user application. The reason generally disk I/O is handled in the kernel because it can be centralized for the entire system. Suppose another process wanted to access the sectors organized by the current application, then, it would necessitate even more processing to propagate the organization information. For this reason and others, the kernel remains a reasonable place for the file system to exist. In an effort to improve performance, the file system can buffer reads. This can cause unwanted latency so the Linux kernel allows the user to ask for non-buffered access to the disk through system calls.

## I/O Scheduler

The Request Queue, a major component of the block subsystem, encapsulates the I/O scheduler. Historically, there has been significant work to optimize the performance of rotating disks, usually by leveraging the greater performance of sequential read or write operations. The block subsystem makes requests to the hardware through Request Queue structures, which are another scheduling construct to optimize the performance of the rotating unit. By merging and reordering the requests, the kernel trades some processor computation to push the performance limit of rotating drives. To the solid-state disk, which has no moving parts aside from possibly a cooling fan, this scheduler becomes pure overhead. The Request Queue ultimately outputs a serial stream of I/O requests; traditional hardware trades processing time to obtain a more optimal stream of requests. For SSD's, an optimal stream also exists, as within the device, there are different banks of flash memory that could be accessed concurrently, but it is not clear it is worth the CPU's time make an effort to find it; nor is it clear that the CPU would have any knowledge of the organization of the flash memory to utilize concurrent bank accesses.

One of the great features of the Linux kernel is the ability to dynamically change the I/O scheduler for any drive simply by writing to special files in the file system. One scheduler, noop, does not actually perform any reordering, and merely passes the requests from the block layer to the device driver. There was about a 10% performance gain compared to more sophisticated schedulers in the mainline, like Completely Fair Queuing (cfq) scheduler which requires more processing to organize requests.

The block layer remains the interface for the kernel and file system to access storage devices. In a sense, this is unlikely to change soon because the solid-state drives are designed with interfaces that emulate existing paradigms, even if the SSD's are capable of

something more parallel. This requires the operating system to be more intelligent with the given resources.

SCSI Slammer

One way to gauge the maximum possible performance is to completely bypass the larger abstractions of the I/O stack and focus on the drivers. The kernel uses SCSI commands to unify the interface with many attached devices. The device will operate on ATA commands coming from the kernel, but the block layer actually issues SCSI commands to the SCSI disk representation in the kernel; the command is translated from SCSI to ATA within libATA. libATA is the driver library in the Linux kernel that interfaces to all ATA devices. A significant portion of the library is dedicated to translation and interaction between ATA and SCSI, but having is unified interface was probably more attractive to the kernel designers.

Using a kernel module that only issues SCSI commands, we can "slam" the disk with random read requests and ascertain the system's performance, especially with respect to the interrupt rate. The goal is to push the driver and interrupt mechanism to the limit, to explore an upper bound on the IOPS rate. This is useful for measurements on both wimpy and beefy processors.

| IOPS | fio | scsi_slammer | x |
|---|---|---|---|
| **2.6.32** | 22.5k | 44.3k | 2.0 |
| **2.6.35** | 22.4k | 66.4k | 2.9 |

These numbers were taken on a machine with a single-core, hyper-threaded Intel Atom processor with an Intel X25-M. They are running various versions of the Linux kernel with block I/O polling for the AHCI driver enabled. fio, flexible I/O tester, is an application designed to measure the I/O under different workloads specified by the user. The job specified in this case is similar to that of the scsi_slammer, a sustained, random read for a short period of wall time (a few seconds). scsi_slammer is a kernel module that injects SCSI read commands for random sectors. This emulates the fio job because it causes the drive to perform SCSI read commands and actually DMA data from the device to memory. Yet, in both the issue and the interrupt code path, the overhead is reduced – nothing in "above" Request Queue layer is utilized. We were interested in the hardware capability of the smaller processor to handle interrupts. Here, it is clear that even though the Intel Atom processor is not able to achieve a high rate of IOPS from userland, it is feasible for the processor to be interrupted at such a rate to perform IOPS at that rate. {}{}

The case is slightly contrived, for obviously there exists far more work that what the scsi_slammer performs when making such tests. In scsi_slammer, there are several elements in memory that are simply static. Additionally, the interrupt path no longer reaches the block subsystem, nor does it launch a deferred function call to complete extra work, as it would if it were a real read request. Still, it takes interrupts as if it were a real request and commands to device to actually perform the DMA into memory. This does not detract from demonstrating the feasibility of performing IOPS at a greater rate on a wimpy platform.

## 4 Driver Timing

Eventually when issuing a command, the physical hardware device must be accessed. In fact, the driver actually plays an important role in the performance of I/O because in issuing the commands. Using *blktrace*, a system in the Linux kernel to record the I/O events within the block layer, we can measure the time spent in each part of processing. With respect to the issuance of commands, the kernel comes with tracepoints for:

- when a command is started
- when the corresponding request is allocated
- when the request is inserted into the request queue
- when the command leaves the block layer as it is sent to the driver

Looking at the median timings of the period of commands, we see that time spent in the driver dominates. That is, given a stream of commands to issue, the majority of the time is spent outside of the block layer, for the disk driver to handle the request and for the I/O stack to initialize the next command for the block layer to process. Adding tracepoints in the driver allows to further distinguish the time spent in either the driver or the other parts of the I/O stack. This gives us a microbenchmark of the individual disk driver.

This measurement includes the majority of the kernel's I/O stack, which is mostly useful for analysis, but does not show the full potential of the hardware.

## 5 New OS

A possible new paradigm of operating systems might contain some concepts of vectors. Vasudevan, Andersen, and Kaminsky [VOS paper] proposed some ways to parallelize repetitive or redundant work performed in the kernel. The desire was to perform more work per system call, and for I/O, that might result in a greater number of commands issued to a device per system call. If each I/O system call were could be more efficient, then more I/O could be performed per system call.

Currently, the final piece will be another modification to benchmark the system. To continually saturate the NCQ, a modified libATA would expel twice the commands actually

issued by the kernel. For every command issued, the lowest level driver will create an additional command to read or write to some disregarded page. Since commands are issued serially, a wimpy processor will naturally have difficulty issuing commands at the same rate as a beefy processor. By issuing the additional commands at the lowest level, we are able increase the saturation of the disk with minimal software overhead. By issuing more commands to disk, it simulates for the device a greater load from the kernel, allowing measurement of the performance of a the system with a disk whose NCQ is more saturated.

## 6 Future Work

I don't know what the future will bring, perhaps a brighter tomorrow.

Sources


- FAWN paper
- http://en.wikipedia.org/wiki/Native_Command_Queuing
- http://www.serialata.org/technology/ncq.asp
- http://www.seagate.com/content/pdf/whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf
- http://wiki.osdev.org/AHCI
- http://lwn.net/Articles/346187/
- http://lwn.net/Articles/346219/
- Vasudevan's unfair I/O deferral patch
- fio
- Intel Interrupt Mitigation for GigaB Note (??)
- Vasudevan's I/O Efficiency Paper
- blktrace
- Vector OS paper