Kernel Accommodations for I/O Intensive Workloads

Nathan Wan (advised by Professor Dave Andersen)

Recent advances in storage technologies have lead to widespread availability of solid state storage devices. These devices are extremely different from their mechanical counterparts in that they can process I/O commands at a far superior rate. Suboptimal performance of nodes in the FAWN (Fast Array of Wimpy Nodes) project inspired this work, where Intel Atom processors could not saturate the random read rate of the solid state drive like an Intel X25-E. We probe the command rates that can be achieved by Linux drivers and the kernel's I/O stack.

Introduction

There exists a disparity in the performance of modern processors and persistent storage devices. 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. A beefy processor might be an Intel Core i7, which has multiple cores on the same die, powerful processing features, high clock speeds and 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).

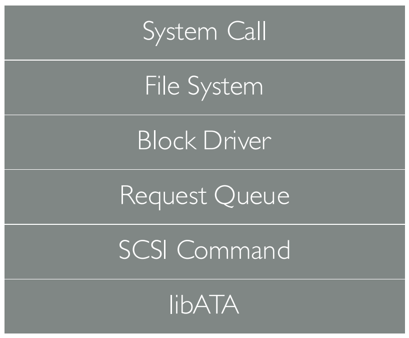
FAWN

In the Fast Array of Wimpy Nodes project, 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/M SSD's, but do not maximize the performance of the SSD. The metric of interest is the IOPS rate. A major bottleneck for the FAWN system is the performance of random reads of small data of disk.

Internally, drives have queues that allow it to handle multiple commands simultaneously. By saturating the drive, we press the hardware IOPS limit but there is difficulty as the commands issued serially by the processor. Even though more powerful processors, like those of the Intel Nehalem Platform, require accessing multiple SSD's to approach the IOPS saturation limit, the Atom processors are much further from that limit. A FAWN node may achieve about 40k random reads per second from an SSD under with some heuristic changes to the kernel, while the same device on an i7 processor can achieve 70k IOPS. The CPU overhead for IO workloads is not exclusive to wimpy processors, but the effects are exacerbated on wimpier processors. Any advantages gained from this work will avail all processors, though the wimpies may reap greater benefit.

The I/O Stack

Today's established kernels were built in the time of rotating hard disk drives and even though SSD's bandwidth is usually comparable, the rate which the drive handles commands deviates considerably. With no moving parts, SSD's gain in power consumption savings and orders of magnitude greater IOPS. Currently, the I/O stack for common storage devices for the Linux kernel looks something like this:



Our focus will revolve around the lower layers, closer to hardware, as changes to kernel wide constructs, like the block layer or file system, would require fundamental changes.

Block Layer

Significant previous work has been done to optimize the performance of rotating disks, particularly the sequential read or write operation. 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

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. For example, we have been very interested in exploiting the Native Command Queue, NCQ, allows the hardware to store multiple commands. Filling and completing commands on this queue in an appropriate manner could provide the desired IOPS rate.

Disk Drivers

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.

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 driver.

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.

Command Multiplier

Currently, the final piece will be another modification to benchmark the system. To continually saturate the queue, *libATA*, the driver for translating SCSI commands to ATA commands as well as issuing them to the device, will 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 garbage 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 overhead.

By issuing more commands to disk, we intend to further gauge the performance of a disk that is more saturated.

N.B. Sorry missing citations