Flash Drive Emulation

Eric Aderhold & Blayne Field
aderhold@cs.wisc.edu & bfield@cs.wisc.edu
Computer Sciences Department
University of Wisconsin, Madison

Abstract

Flash drives are becoming increasingly more common. In order to be able to effectively make use of these devices, it is necessary to be able to design systems that are optimized for ideal flash drive performance. To that end, we have developed a flash drive emulator. This emulator can allow software (such as file systems or any other system that would need to interact directly with the disk) to be tested for flash drive performance. The ability to use an emulator for this purpose should be able to make this testing more cost-effective, as there would no longer be as much of a need to buy expensive hardware that is prone to wearing out after a finite number of writes.

1 Introduction

Storage devices based on flash memory are becoming more common. Their capacities have increased dramatically in recent years, and their price has fallen to the point where flash storage devices have started to replace disks in a few applications, particularly in notebook computers and other mobile devices.

Since flash-based storage devices are a fundamentally different technology than traditional rotating disks, it makes sense to design file systems that are optimized for flash memory's different hardware properties. Some research has already been done in this area, [2, 3] but the field could benefit from more work, especially as the underlying hardware technology evolves over time.

Unfortunately, there can be significant costs to this research. Flash devices, especially those with capacities in the tens of gigabytes (or higher), can be expensive. Furthermore, the tendency of flash hardware to wear out after a few hundred thousand writes to any block means that when such drives are used intensively for research purposes, they may wear out fairly quickly and need to be replaced.

Because of this problem, we decided to design and implement a Flash drive emulator. Our emulator is built on top of the Linux RAM disk driver, and designed to closely approximate the read and write performance of an existing and has been tested to closely match the performance of an existing flash drive for both read and write operations. A related project has been undertaken for hard drives in the past [1]. However, as flash hardware does not have the complex rotational delays and seek delays of a traditional hard drive, a flash drive emulator is easier to get to run at speeds such that the timing delays can be accurately calculated and mod-

eled faster than the emulated drive would return data.

2 Approach / Methods

2.1 Basic Approach

On a high level, there are four main layers to the Linux storage system. On the top layer is the system calls which call into the file system and the applications that interact with it. The file system in turn interacts with the device through the VFS layer (which virtualizes the hardware to appear as an abstract device). Depending on the type of device, these calls are routed down to the appropriate device driver. Underneath the device driver is the lowest layer, the actual hardware device.

Since our goal is to be able to emulate system performance on flash hardware, our basic approach is to modify the lower layers of the storage system in such a way that a file system or other program that is accessing the device cannot notice any material difference between the emulated device and actual hardware. In order to do this, we are decided to modify the VFS and driver levels of the system.

In particular, the device driver we decided to modify is that of the RAM disk. A RAM disk is essentially a portion of memory that is treated by the system as a block storage device. This allows a file system or anything else that would be stored on a disk to be stored on a RAM disk volume instead. RAM disks are commonly used within the Linux kernel to assist with the boot process, though most users make little use of them after the computer has been started. By adding appropriate timing delays to the RAM disk driver and the VFS layer above it, we are able to closely emulate the read and write performance of a flash drive, while preserving the set of system calls and operations that would normally be available with any device.

2.2 Emulating Writes

In our initial performance measurements, we realized that an important factor in the emulation of flash disk writes is the distinction between random writes and sequential writes. These different workload types perform significantly differently on flash hardware, so it is necessary to be able to quickly and correctly classify a write being done to our system as either being a sequential write or a random write.

Our first step in classifying these writes was to simply observe the patterns that were present in sequential writes that were not present in random writes. We ran some random write tests and sequential write tests (described in more detail in the Results section), and noticed that the byte offset being accessed in the sequential write calls would increase in a constant amount before eventually rolling back over to zero and starting again. Furthermore, the starting offset for a sequential write operation was identical to the ending offset for the previous operation, for obvious reasons. Thus, in order to detect sequential writes, we simply added a variable to the RAM disk driver to keep track of the previous ending offset, and compared it to the current beginning offset.

An important point to note in this approach is that in a sequence of sequential writes, the first of these writes will necessarily be classified as a random write and will incur the cost penalty associated with random writes. Intuitively, this makes perfect sense since the first of a series of sequential writes will necessarily be reading from a different section of the disk than the previous write. However, our

empirical data indicates that the full random write delay is not typically present at the beginning of a series of sequential writes, so we needed to be able to compensate for this over-waiting. We did this by storing an additional parameter that estimates how much time we have waited more than the observed initial sequential delay, and we reduce future sequential delays in subsequent writes to compensate for this extra delay. We set this time whenever we see a sequential write that was preceded by a random write, since this means that we must have mis-classified the random write.

Our approach was intentionally kept as simple as possible. We only needed to add a couple of variables to the driver to keep track of state. One advantage to this approach is that since the amount of information stored is kept to a minimum, any necessary computations can be done very quickly. This means that any computations have little danger of taking longer than the corresponding action on the actual flash drive. Furthermore, keeping the calculations simple leaves more CPU time for other tasks, and also having a small set of factors makes the process of reconfiguring the emulator for a different computer or different flash drive much simpler. Finally, this design greatly simplifies the future task of automatically determining these parameters from a set of experimental data.

However, this is not to say that our system is without flaws. Our experimental tests of the sequential write functionality resulted in a fairly complex performance model, with some linear portions and some discontinuities. As a result, our emulated results (as shown in more detail in the Performance Analysis section) were slightly faster or slower than the flash hardware for most individual write sizes, though the emulator matches the overall trend fairly well. Even so, a slightly more complex emulator model, involving an additional one or more state variables, would likely improve results even more.

We store the following four pieces of information within the modified RAM disk driver:

- intra-step delay
- inter-step delay
- step locations (writes between steps)
- excess delay to burn

The two delays can be easily modified through the `ioctl` interface to the emulator device. The exact ways in which these delays are used can be seen in more detail in the code samples in Appendix A.

One issue we have with sequential writes is that function of time taken appears like a step function as the number of blocks we write. This delay should correspond to moving to a new block on the flash device. There is also an increase that happens between the steps, the cause of which is not as immediately clear (no such delay exists for reads).

Each time we do a write to a RAM disk, the following three functions are called every time, in order: `ramdisk_prepare_write`, `ramdisk_commit_write`, and `ramdisk_set_page_dirty`. We do all of our delays in the `ramdisk_prepare_write` function, since this one is called first, and it also has all of the information that we need in order to calculate the delays passed in as parameters.

2.3 Emulating Reads

One important difference between emulating reads and emulating writes with the RAM disk code is the flow of control through the kernel. While in a write,

there are three functions in the RAM disk driver (`rd.c`) that get called for every write operation, the same is not true for reads. This perplexed us for some time. It turns out that when a read occurs from a RAM disk, the only code called in the RAM disk driver is the function `rd_open`, and this only happens on the first read operation. This function apparently initializes an inode to point to the appropriate part of the memory page table for the RAM disk, but the actual reading does not occur in the device driver code.

Thus, we needed to move one level up in the storage stack, to the VFS layer, in order to add read delays. We needed to catch calls for reading from the RAM disk in the `vfs_read` function, and do the appropriate timing delays in that function.

However, this added a new complication that was not a factor when emulating writes. At the `vfs_read` level, we cannot just indiscriminately apply delays to each call made, since this function will be called for reads from many more devices than just the flash disk emulator. Thus, we needed a good way of detecting if the call is intended for the emulator or for some other device. This fortunately turns out to be a very simple test. By following the function pointer to its inode, we can retrieve the id of the device associated with that file. If that device id matches that of the RAM disk we are using for our emulator, we can then apply the appropriate delay. This reliance on the device id can unfortunately be somewhat system-dependent, though not any more so than other portions of our emulator.

The delays in the read function turn out to be a bit simpler than the delays needed to emulate writes. Again, for random reads we get a very linear performance pattern from the flash drive, whereas for sequential reads, we once again get a stepwise function. However, the slope within a single step is nearly nonexistent in the sequential reads, which means that it is not necessary to use the additional linear slope parameter that is required for the write emulation code.

The main difficulty for emulating reads, then, lies in finding the correct size of the jumps, and where they occur. For some reason, the size between jumps isn't constant; they rather seem to increase for a while until reaching a fixed size. While we are uncertain about the root cause of this behavior, we were able to successfully emulate it by modeling the width of these steps and updating the time until the next jump based on the current step that we were on.

Similarly to in the emulation of writes, we once again need to be able to distinguish between random and sequential access. Our method for doing this is essentially the same as in the write emulation section.

One improvement that could be made for emulating reads would be to store some additional information for each file that is read (at least the ones accessing the RAM disk). This is because it may be possible for concurrent access to the RAM disk to cause everything to be classified as random (if two processes are alternating reads, for example). However, in reality, caching may in fact be lessening the performance hit of such behavior. We do not know if this is an issue.

3 Flash Emulator Performance Analysis

The flash emulator was tested for performance against a 32MB generic-branded flash drive with a

USB interface. For major sets of tests were run: sequential read tests, sequential write tests, random read tests, and random write tests. All of these tests were implemented using the Linux `dd` command, which is able to read from and write to the disk directly, without any higher-level file systems that could potentially confound the results.

3.1 Sequential reads

For the sequential read test, we tested both drives using various sizes of reads. All reads were done with the Linux `dd` program. For each read amount, the `dd` program was run with a parameter stating how much data was to be read. In each case, the read began from the beginning of the device and continued until the requested amount of data had been read. With both devices, the time required for a given sized read scaled up in a fairly stepwise fashion. Certain ranges of data amounts took essentially the same amount of time to read, and the time would jump sharply upward once that range was surpassed. We attribute this behavior to the architecture of the underlying hardware; we assume that the flash drive is partitioned into a certain number of blocks, each of which must be read all at once.

The important bit of information for our purposes, however, is not why the drive behaves as it does, but is instead how closely our emulator is able to approximate the drive's behavior. On this measure, the emulator did mostly quite well. The similarity in sequential read performance between the flash drive and the emulator is shown in Figure 1. With the exception of the first two very small tests, the performance of the flash drive was always within 20 ms of the simulator. The difference in the small-size write performance is an area where our model could use some improvement.

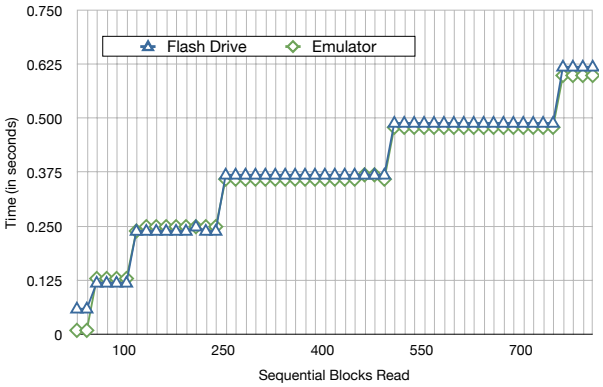

Figure 1: **Sequential read performance**

3.2 Random reads

For the random read test, we again tested both drives using various sizes of reads. All reads were again done with the Linux `dd` program. The main difference between the sequential read test and the random read test is that the `dd` program was invoked multiple times for each random test, reading only one block per call to `dd`, up to the total number of blocks to be read. The location of the block to be read was computed randomly before invoking `dd`. The pattern for these random read tests was very linear. The graph in Figure 2 shows this pattern, with the flash drive and emulator again having very similar performance. This time, the similarity was even closer than that for the sequential reads. This can be attributed to the relative simplicity of the model used to emulate random reads as compared to sequential reads.

3.3 Sequential writes

The sequential write performance of the flash drive was a bit more challenging to model than either type of read performance. Instead of being a purely linear graph (like the random reads) or a purely stepwise graph (like the sequential reads), the sequential

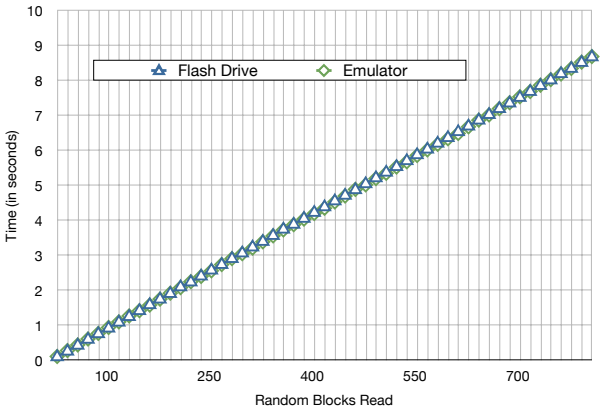

Figure 2: **Random read performance**

writes seem to be a combination of both. Using a very similar test to the sequential read test described above, our emulator was able to approximate the performance of the true flash drive fairly closely. The exact difference is shown in Figure 3. Because this combination linear/stepwise performance was a bit more difficult to model, the two devices aren't quite as close together in performance as either of the read performance graphs. Nevertheless, the average discrepancy between the flash drive and the emulator was a mere seven ms.

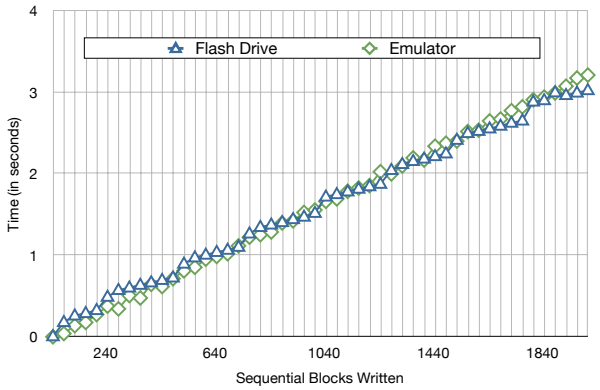

Figure 3: **Sequential write performance**

3.4 Random writes

The random write performance was very similar to the random reads. Both random reads and writes resulted in a very linear distribution, and the emulator was able to very closely approximate the performance of the flash drive. The results from this set of tests are shown in Figure 4.

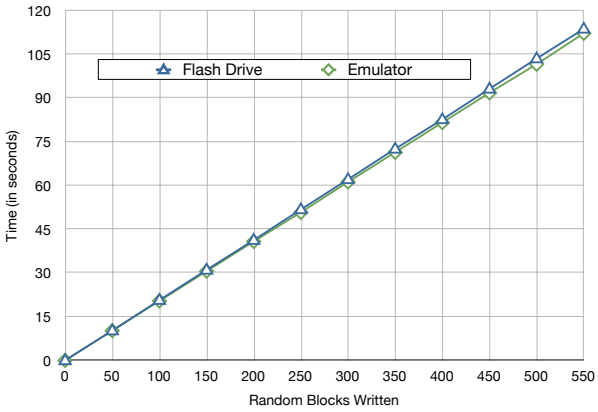

Figure 4: **Random write performance**

4 Conclusions

In conclusion, our flash drive emulator is a good step toward a tool that could be useful for researchers studying file systems for flash drives or other applications that would need to optimize their workload for a flash-based disk device. Our results show that the basic performance of a flash drive can be emulated fairly closely without the need for an overly complex model. This can even be done to reasonable accuracy without a detailed understanding of the operational details of the underlying physical flash hardware.

References

[1] J. Griffin, J. Schindler, S. Schlosser, J. Bucy, and G. Ganger. Timing-accurate storage emulation. *Proceedings of the Conference on File and Storage Technologies (FAST)*, 2004.

[2] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. *Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings table of contents*, pages 13–13, 1995.

[3] D. Woodhouse. JFFS: The Journalling Flash File System. *Ottawa Linux Symposium*, 2001, 2001.

A Write Delay Code

```
static int ramdisk_prepare_write(struct file *file, struct page *page,
                                 unsigned offset, unsigned to)
{
    static unsigned last_to = 9999;
    static uint64_t delay = 0;
    static int last_state = RANDOM_STATE;
    static uint64_t extra_cycles = 0;
    static int threshold = 195;
    static int seq_writes = 0;

    // Sequential
    if(offset == last_to % 4096) {
        if (last_state == RANDOM_STATE && rand_delay > seq_delay) {
            extra_cycles = (rand_delay - seq_delay);
        }

        if ((seq_writes = (seq_writes + 1) % threshold) == 0) {
            threshold = 256;
            wait_ticks(seq_spike);
        }

        delay = (extra_cycles > seq_delay ? 0 : seq_delay - extra_cycles);
        extra_cycles -= (extra_cycles > seq_delay ? 0 : extra_cycles);

        if (last_state == RANDOM_STATE && rand_delay < seq_delay) {
            delay += (seq_delay - rand_delay);
```

```
        }

        last_state = SEQUENTIAL_STATE;
    }
    else {
        delay = rand_delay;
        last_state = RANDOM_STATE;
        extra_cycles = 0;
        seq_writes = 0;
        threshold = 195;
    }
    // resume ramdisk_prepare_write ...
```

B Read Delay Code

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    static uint64_t last_pos = 128647;
    static int seq_reads = 0;
    static int my_threshold = 32;
    static int state = 0;

    if (file->f_mapping->host->i_rdev == RAM_DISK_DEV_ID) {
        if (*pos == last_pos) {
            state = 1;
            if (++seq_reads >= my_threshold) {
                    wait_ticks(rd_seq_read_delay);
                    if (my_threshold < 256) {my_threshold *= 2;}
                    seq_reads = 0;
            }
        } else {
            wait_ticks(rd_rnd_read_delay);
            state = 0;
            seq_reads = 0;
            my_threshold = 32;
        }
        last_pos = *pos + count;
    }
    // continue vfs_read function ...
```