The Unwritten Contract of Solid State Drives [1]

Jun He Sudarsun Kannan Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

Department of Computer Sciences, University of Wisconsin–Madison

Abstract

The "unwritten contract" of SSDs is a set of rules that clients of SSDs should follow to obtain high performance. We formalize the "unwritten contract" and perform a detailed vertical analysis of application performance atop a range of modern file systems and SSD FTLs to uncover application and file system designs that violate the contract. Our analysis, which utilizes a highly detailed SSD simulation underneath traces taken from real workloads and file systems, provides insight into how to better construct applications, file systems, and FTLs to realize robust and sustainable performance.

1. Introduction

In-depth performance analysis lies at the heart of systems research. However, there exists a large and important gap in our understanding of I/O performance across the storage stack. New data-intensive applications, such as LSM-based (Log-Structured Merge-tree) key-value stores, are increasingly common; new file systems, such as F2FS, have been created for an emerging class of flash-based Solid State Drives (SSDs); finally, the devices themselves are rapidly evolving, with aggressive flash-based translation layers (FTLs) consisting of a wide range of optimizations. How well do these applications work on these modern file systems, when running on the most recent class of SSDs? What aspects of the current stack work well, and which do not?

The goal of our work is to perform a detailed vertical analysis of the application/file-system/SSD stack to answer the aforementioned questions. We frame our study around the file-system/SSD interface, as it is critical for achieving high performance. While SSDs provide the same interface as hard drives, how higher layers utilize said interface can greatly affect overall throughput and latency.

Our first contribution is to formalize the "unwritten contract" between file systems and SSDs, detailing how upper layers must treat SSDs to extract the highest instantaneous and long-term performance. Our work here is inspired by Schlosser and Ganger's unwritten contract for hard drives [3], which includes three rules that must be tacitly followed in order to achieve high performance on Hard Disk Drives (HDDs); similar rules have been suggested for SMR (Shingled Magnetic Recording) drives. The SSD rules are naturally more complex than their HDD counterparts, as SSD FTLs (in their various flavors) have more subtle performance properties due to features such as wear leveling and garbage collection.

We present five rules that are critical for users of SSDs. First, to exploit the internal parallelism of SSDs, SSD clients should issue large requests or many outstanding requests (*Request Scale* rule). Second, to reduce translation-cache misses in FTLs, SSDs should be accessed with locality (*Locality* rule). Third, to reduce the cost of converting page-level to block-level mappings in hybrid-mapping FTLs, clients of SSDs should start writing at the aligned beginning of a block boundary and write sequentially (*Aligned Sequentiality* rule). Fourth, to reduce the cost of garbage collection, SSD clients should group writes by the likely death time of data (*Grouping By Death Time* rule). Fifth, to reduce the cost of wear-leveling, SSD clients should create data with similar lifetimes (*Uniform Data Lifetime* rule).

We utilize this contract to study application and file system pairings atop a range of SSDs. Specifically, we study the performance of four applications – LevelDB (a key-value store), RocksDB (a LevelDB-based store optimized for SSDs), SQLite (a more traditional embedded database), and Varmail (an email server benchmark) – running atop a range of modern file systems – Linux ext4, XFS, and the flash-friendly F2FS. To perform the study and extract the necessary level of detail our analysis requires, we build *WiscSee*, an analysis tool, along with *WiscSim*, a detailed and extensively evaluated discrete-event SSD simulator that can model a range of page-mapped and hybrid FTL designs. We extract traces from each application/file-system pairing, and then, by applying said traces to *WiscSim*, study and understand details of system performance that previously were not well understood. *WiscSee* and *WiscSim* are available at `http://research.cs.wisc.edu/adsl/Software/wiscsee/`.

Our study yields numerous results regarding how well applications and file systems adhere to the SSD contract; some results are surprising whereas others confirm commonly-held beliefs. For each of the five contract rules, our general findings are as follows. For request scale, we find that log structure techniques in both applications and file systems generally increase the scale of writes, as desired to adhere to the contract; however, frequent barriers in both applications and file systems limit performance and some applications issue only a limited number of small read requests. We find that locality is most strongly impacted by the file system; specifically, locality is improved with aggressive space reuse, but harmed by poor log structuring practices and

[1] The full paper of this abstract has been published in EuroSys'17, under the same name. Link: `http://pages.cs.wisc.edu/~jhe/eurosys17-he.pdf`

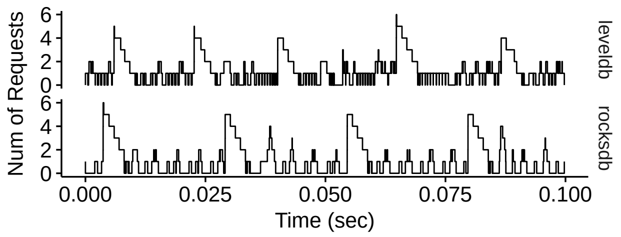

Figure 1: **Number of Outstanding Requests During Compaction.**

legacy HDD block-allocation policies. I/O alignment and sequentiality are not achieved as easily as expected, despite both application and file system log structuring. For death time, we find that although applications often appropriately separate data by death time, file systems and FTLs do not always maintain this separation. Finally, applications should ensure uniform data lifetimes since in-place-update file systems preserve the lifetime of application data.

We have learned several lessons from our study. First, log structuring is helpful for generating write requests at a high scale, but it is not a panacea and sometimes hurts performance (e.g., log-structured file systems fragment application data structures, producing workloads that incur higher overhead). Second, due to subtle interactions between workloads and devices, device-specific optimizations require detailed understanding: some classic HDD optimizations perform surprisingly well on SSDs while some SSD-optimized applications and file systems perform poorly (e.g., F2FS delays trimming data, which subsequently increases SSD space utilization, leading to higher garbage collection costs). Third, simple workload classifications (e.g., random vs. sequential writes) are orthogonal to important rules of the SSD unwritten contract (e.g., grouping by death time) and are therefore not useful; irrelevant workload classifications can lead to oversimplified myths about SSDs (e.g., random writes considered harmful [2]).

2. Observations

We have made dozens of observations based on vertical analysis of applications, file systems, and FTLs. By analyzing how well each workload conforms to each rule of the contract, we can understand its performance characteristics.

As an example, we present one of our observations about request scale (the sizes of I/O requests and the number of outstanding requests) in this section. We evaluate and pinpoint request scale violations from applications and file systems by analyzing block traces, which include the type (read, write, and discard), size, and time (issue and completion) of each request.

We found that Linux buffered I/O may limit the sizes of requests and the number of outstanding requtests of applications, epitomized by LevelDB and RocksDB in our evaluation. Even though LevelDB and RocksDB read 2 MB files during compactions, which are relatively large reads, their request sizes and number of outstanding requests are still small. The number of outstanding requests are shown in Figure 1. The periodic bursts of up to six requests are due to writes; the requests between the bursts are from read. There are only ever up to two outstanding read requests. The median of read request sizes is only about 128 KB (not shown in the figure). One reason for the small request sizes and limited number of outstanding requests is that the LevelDB compaction, as well as the RocksDB compaction from the first to the second level ("the only compaction running in the system most of the time" [1]), are single-threaded and read data by buffered `read()` or `mmap()`.

The buffered `read()` and `mmap()` split and serialize requests before sending to the block layer and subsequently the SSD. If the buffered `read()` is used, Linux will form requests of `read_ahead_kb` (default: 128) KB, send them to the block layer and wait for data one at a time. If the buffered `mmap()` is used, a request, which is up to `read_ahead_kb` KB, is sent to the block layer only when the application thread reads a memory address that triggers a page fault. In both buffered `read()` and `mmap()`, only a small request is sent to the SSD at a time, which cannot exploit the full capability of the SSD. To increase the size and the number of outstanding I/O requests, direct I/O should be used. The direct I/O implementation sends application requests in whole to the block layer. Then, the block layer splits the large requests into smaller ones and sends them asynchronously to the SSD.

3. Conclusions

Due to the sophisticated nature of modern FTLs, SSD performance is a complex subject. To better understand SSD performance, we formalize the rules that SSD clients need to follow and evaluate how well four applications (one with two configurations) and three file systems (two traditional and one flash-friendly) conform to these rules on a full-function SSD simulator that we have developed. This simulation-based analysis allows us to not only pinpoint rule violations, but also the root causes in all layers, including the SSD itself. We have found multiple rule violations in applications, file systems, and from the interactions between them. We believe our analysis here can shed light on design and optimization across applications, file systems and FTLs, and the tool we have developed could benefit future SSD workload analysis.

References

[1] RocksDB Tuning Guide. `https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide`.

[2] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[3] S. W. Schlosser and G. R. Ganger. MEMS-based Storage Devices and Standard Disk Interfaces: A Square Peg in a Round Hole? In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 87–100, San Francisco, California, April 2004.