Draconis: Rethinking the Storage Stack

Jialin Li
lijl@cs.washington.edu

1 Introduction

High performance storage systems like file systems, databases and persistent key value stores have very high storage I/O requirements. To maximize the efficiency of disk I/O, these systems usually make certain assumptions about the performance characteristics of the underlying storage medium, and design their disk read/write patterns accordingly. Some of the common assumptions include: sequential accesses are faster than random accesses, minimum read/write size should be a sector (512 bytes), batching writes into a large chunk is more efficient than writing small pieces, etc. Most of these assumptions are based on characteristics of a spinning disk, and therefore usually give applications efficient storage I/O when actually running on spinning disks.

However, newer storage technologies like NAND SSDs, phase change memory and memristors all have very different performance characteristics from spinning disks. For example, SSDs do not have moving mechanical parts and therefore do not incur the extra seek penalty when accessing random addresses; the minimum read/write size is an SSD page size, which is typically 4KB; pages need to be erased before writing new content, causing in place updates to be very costly. Not surprisingly, applications written for spinning disks will not have the optimal performance on these newer storage technologies. To make the issue more problematic, different products of the same technology may have very different performance properties. SSDs for instance have different Flash Translation Layers (FTLs), and depends on the translation scheme, random writes can be much slower or equally fast as sequential writes. As a result, applications optimized for one SSD may have much worse performance on another SSD.

The problem involves more than just the application layer. Operating systems' storage I/O subsystem are also designed around a slow spinning disk: the disk interface only has simple sector read/write, I/O scheduling and buffer cache design optimize for sequential accesses and batching, etc. Even when applications make the correct assumptions, the operating system may totally disturb the disk access pattern, leads to non-optimal performance. With applications, operating systems, file systems, drivers, disk controllers and disk hardware all optimizing with only local information, it becomes extremely difficult for the application to have any end to end storage performance guarantees.

This project thus aims to revisit storage system design across the stack. The goal of the project is twofold. Firstly, understand the inefficiencies caused by application's wrong assumptions about the storage hardware and by the interference from multiple storage layers. This will involve measurements of several storage applications on different storage technologies (spinning disks and SSDs) and performance analysis against raw hardware capabilities. Secondly, base on the observations from the measurement study, propose a new storage stack design that optimizes applications' I/O performance across different storage hardware. This may involve redesign of the OS I/O subsystem, storage API, disk driver/controller or the application.

2 Diverse Hardware

2.1 Spinning Disks

Magnetic spinning disk, or hard disk drive, is still the most widely used persistent storage technology. As the name suggested, magnetic disks have spinning platters and moving disk heads that read and write data by sensing or magnetizing platter surfaces. The minimum read/write size on a hard disk drive is a sector, which is usually 512 bytes. To read or write to a particular sector, the disk has to first move the disk head to the desired track (seek time) and then wait for the platter to spin until the target sector is under the head (rotation time). Average seek times are typically around 5-10 ms and average rotation latencies are around 4 ms for 7,200 HDDs and 2 ms for 15,000 HDDs.

Due to the costly seek and rotation latencies, spinning disks perform badly when doing small and random accesses. It is not surprising that applications and operating systems try to optimize performance running on a hard disk drive by amortizing the access latencies. One common technique is to buffer small writes in memory and only writes to disk in large sequential chunks. Another strategy is to prefer sequential accesses over random accesses. OS has i/o scheduler that minimizes random seek time and the native command queue (NCQ) enables the disk drive to serve multiple outstanding requests in a sequential order. Many storage applications use append only update schemes to avoid the costly random writes altogether. Example scheme includes log-structured file system [15], log-structured merge tree [13] and persistent data structures [7].

2.2 SSDs

NAND flash based SSD is a newer storage technology that has significantly higher performance than magnetic spinning disks. SSDs do not have any moving mechanical parts like their hard disk counterparts: data are read and written using electrical circuits. As a result, random accesses on an SSD are much faster. However, due to the physical property of flash memory, a flash page has to be erased before writing new data. Erase operations are much slower compare to reads and writes, and SSDs can only erase flash memories in large chunks called erasure blocks which typically range from 128 KB to a few MB. On the other hand, the minimum read/write size on an SSD is a flash page which is typically 2KB, 4KB, 8KB or 16KB. Flash memory has a limited program/erase (P/E) cycles before it becomes wear out. The number of maximum P/E cycles depends on the flash technology, for example SLC flash memory usually have more than a hundred thousand P/E cycles while MLC flash memory only have a few thousands.

To address the challenges of slow erase operation and limited P/E cycles, SSDs employ a flash translation layer (FTL). FTL maps logical flash page addresses to physical page addresses. When writing to a (logical) flash page, the FTL simply marks the old physical page as invalid, writes the data to an already erased physical page and creates the new mapping. The invalid pages are later garbage collected in the background. Consider the same flash page write situation without an FTL: not only the SSD needs to erase the whole erasure block that covers the page, it also needs to copy and rewrite all the other flash pages in the same erasure block. This translation scheme also helps with wear-leveling: writes to hot logical pages are spread across different physical pages.

The physical property of SSDs also enables bigger internal parallelism opportunities than hard disk drives. An SSD may contain multiple flash memory packages, each package consists of one or more dies. Within each memory die, there are two or more planes. All these components can be accesses in parallel or interleaved, creating large potentials of parallelism. More advanced FTL creates mappings that take full advantages of internal parallelism: logical addresses are stripped across packages, dies and planes. Coupled with Native Command Queuing (NCQ), modern SSDs are able offer unprecedented throughput and bandwidth.

3 Measurements

In this section, we will show some preliminary measurement results as a basis for our storage proposal. We first measure the baseline hardware performance of a high-end SSD and explore some of the characteristics of the device. We then measure the performance of a popular key-value store, LevelDB, on the SSD as well as on a HDD.

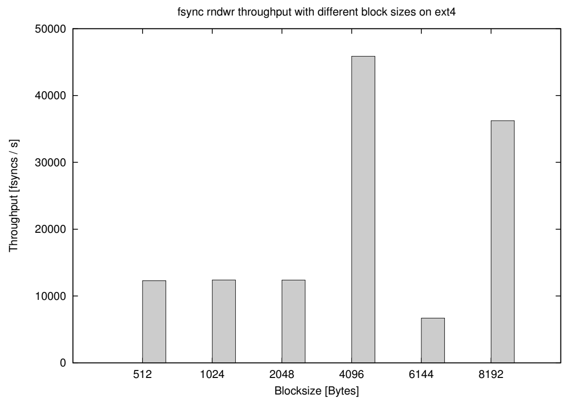

Figure 1: SSD write throughput with varies block sizes. The throughput is significantly higher with 4KB and 8KB blocks which demonstrates that the page size of the device is 4KB: writes not of (multiple) page size will incur expensive Read-Modify-Write.

3.1 SSD Characteristics

As described in section 2, SSDs have a very different hardware architecture from HDDs. To explore the performance implications of the SSD architecture, we conduct some I/O benchmarks on an Intel DC P3700 SSD attached through the PCIe bus. Intel DC P3700 SSD is a high-end storage device targeting high performance data center environment. The product specification indicates that the drive supports up to 460K random 4KB reads and 175K random 4KB writes. Both sequential read and write latency are 20us. All the benchmarks are using the i/o test from sysbench.

3.1.1 Page Size and RMW

As mentioned in section 2, the minimum read/write size of an SSD is a flash page, and the flash page size varies between different SSDs. To determine the page size of our Intel DC P3700, we measure the throughput of writing different size blocks (multiples of 512 byte sectors) to the drive. If the block size is smaller than the page size, the SSD has to perform a Read-Modify-Write(RMW): reads the old page into the register, modifies the block within the page and writes the new page back to the drive. RMW is more costly than writing an aligned page, and as a result writing blocks smaller than page size will yield lower throughput. Figure 1 shows that write throughput is much higher with 4KB and 8KB blocks. This demonstrates that the page size of the device is 4KB. Any writes not of multiples flash page size will incur the expensive Read-Modify-Writes.

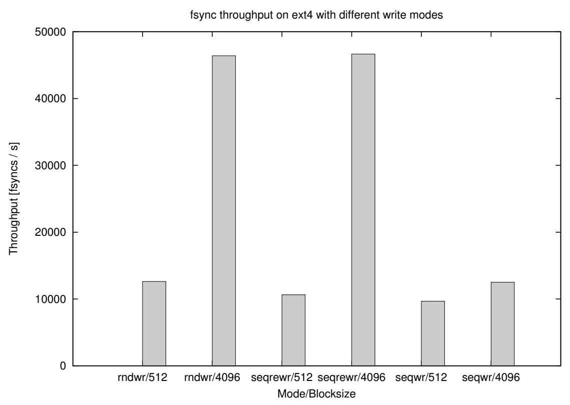

Figure 2: SSD write throughput with 3 different write modes. Sequential and random access pattern do not have significant differences. Sequential write (append) has worse throughput because the file system writes meta data which is less than flash page size, causing RMW.

3.1.2 Write Mode

We then test the throughput of different write modes. The three write modes are random write (rndwr), sequential rewrite (seqrewr) and sequential write (seqwr). Sequential writes write to blocks in numerical sequential order while random write picks blocks in random order. The difference between sequential rewrite and sequential write is that sequential rewrite writes to an already created file with the target size, which means no new blocks are allocated and the length of the file is not changed(same for random write). Figure 2 shows that random write and sequential rewrite have roughly the same throughput. This is a direct result of SSDs not having moving mechanical parts and random accesses are as fast as sequential accesses. Throughput of sequential write is much lower because the file system has to write small meta data to the drive. These meta data are smaller than page size and thus incur the expensive RMW.

3.1.3 Parallelism

To explore the internal parallelism of the SSD, we measure the throughput of concurrently writing flash page size blocks to the drive. Figure 3 shows that the SSD write throughput scales to 8 threads. The throughput does not increase beyond 16 threads, however this is mostly due to the hardware maximum throughput (175K from the specification) is already reached. We believe the experiment demonstrates that our SSD has at least 8-16 way parallelism.

3.2 LevelDB Performance

We then measure the application performance on our Intel SSD as well as a HDD. The application we pick is LevelDB, a popular persistent key value store employed by existing systems including Google Chrome's IndexedDB, Bitcoin and Riak. We will first describe the design of LevelDB and then the measurement results.

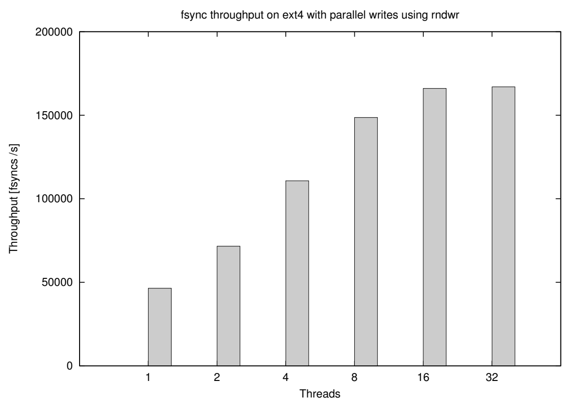

Figure 3: SSD write throughput with increasing concurrency. Write throughput scales to 8 threads. Beyond 16 threads, the hardware throughput limit is reached (175K).

3.2.1 LevelDB

LevelDB is a popular persistent key value store employed by existing systems including Google Chrome's IndexedDB, Bitcoin and Riak. The design principle of LevelDB highly resembles an LSM-tree. The in-memory component is named *memtable* and on-disk components are referred to as *SSTables*. All updates go directly to the *memtable*, and LevelDB writes out the *memtable* as an *SSTable* to disk when it reaches a predefined size. Entries in an *SSTable* are sorted by keys and LevelDB organizes *SSTables* in a hierarchical structure, called levels. Level 0 holds all the newly written out *SSTables*. When level 0 tables grow to a threshold, LevelDB compacts a subset of the tables into level 1 *SSTables*. The same compaction process applies to higher level tables as well, with logarithmically increasing threshold sizes (10MB for level 1, 100MB for level 2 etc.). *SSTables* in any levels other than level 0 have non-overlapping key ranges. To preserve this property, the compaction process for level N picks one (or more in the case of level 0) *SSTable* and merges with all *SSTables* with overlapping key ranges in level N+1.

LevelDB provides atomicity and durability by Write-Ahead-Logging(WAL). All updates are appended to an on-disk log in addition to the *memtable*. In the face of a system crash, LevelDB recovers recent updates from the log (durability), and discards in-complete or corrupted updates (atomicity). LevelDB deletes the old log when writing the *memtable* to disk as an *SSTable*. WAL appends can be configured as asynchronous or synchronous. Asynchronous mode commits the operation without waiting for the log entry to persist on disk, thus gives better performance but could lead to data losses in the presence of power failures or system crashes. Applications with strong durability requirements have to go with the synchronous configuration.

The LSM-tree like architecture offers LevelDB an efficient disk I/O scheme. All key-value updates apply directly to *memtable*, and LevelDB occasionally write large *SSTables* to disk sequentially. In addition, once *SSTables* are written to disk, they are never modified inplace. During compaction, obsolete *SSTables* are simply deleted with newly merged *SSTables* appended to the disk. This large-sequential-append only write scheme is believed to be a good fit for both magnetic spinning disks and SSDs. The synchronous WAL mode however remains to be costly due to the long disk access latency on each update operation. LevelDB also provides fast read operations. LevelDB stores all *SSTable* indexes in memory. All reads thus are either served from memory (key-value resides in *memtable*), or require one disk access (retrieve on-disk location by searching in-memory indexes). Caching techniques like OS buffer cache and application caching could further reduce the number of disk accesses.

3.2.2 Asynchronous Writes Performance

We start with a single thread handling all LevelDB requests and use the asynchronous WAL configuration. The workload is write-only, with 16 Byte keys and 512 Byte values. Each experiment run issues a total of 8GB key-value write requests to LevelDB. We compare the throughput of two write patterns: sequential and skewed random. The sequential workload writes keys in strict numerical order. The skewed random pattern is a more realistic workload: 90% of the keys are picked randomly from 10% of the overall key space. This skewed distribution with small "popular" regions is very common in production workloads. We also vary the size of the overall key space.

Figure 4 shows the throughput of the two write patterns running on a hard disk drive and an SSD. Both hardware show similar performance results. The sequential workload has a much better throughput than skewed random writes. To understand the reason behind this performance divergence, we plot several LevelDB statistics in Figure 5 and Figure 6. Figure 5 shows that LevelDB creates the same number of level-0 *SSTables* regardless of the workload and hardware. This is due to LevelDB appending updates to *memtable* and writing out *memtable* as level-0 *SSTable* at fixed threshold size, making the number of *memtables* depends solely on the total write size. With the same reasoning, the WAL overhead also only depends on the total write size which is the same across all experiments. The only remaining factor is the LevelDB compaction overhead, and Figure 6 demonstrates compaction is indeed the reason behind the throughput divergence. The sequential workload creates zero LevelDB compaction as compared to over thirty thousands compactions during the skewed random workload. Compaction adds two significant overhead to the system: CPU computation to merge multiple tables and storage I/O to write the newly compacted *SSTables*. It is therefore no surprise that the throughput of sequential writes is much higher. The higher number of compactions also explains the lower throughput as the total key space increases in the skewed random case. The reason sequential writes create zero compaction is that the sequential pattern creates sorted, non-overlapping key ranges in all level-0 *SSTables* and thus requires no compaction.

The sequential write pattern essentially gives the best possible write performance for LevelDB: with zero compaction, each key-value pair is written to disk only once (plus WAL), and with the asynchronous configuration, LevelDB writes to storage in large sequential chunks which is efficient for both hard disk drives and SSDs. However, in a more realistic skewed workload, LevelDB's write throughput drops by more than 4X. The question thus follows: can we do any better? Many persistent data stores, including LevelDB, use an append only update scheme to avoid the expensive in-place updates. An side-effect of this update scheme is that it leaves obsolete data on disk and thus requires garbage collection. LevelDB uses compaction to garbage collect invalid key-value entries, and many other key-value stores employ a similar scheme. Though this GC overhead is inevitable, SSDs have already implemented very efficient garbage collection in hardware. Therefore, it is possible to utilize SSD hardware garbage collection to improve the performance of LevelDB.

3.2.3 Synchronous Writes Performance

Next, we configure LevelDB to use synchronous WAL. Synchronous WAL is required to guarantee durability. Figure 7 depicts the throughput of synchronous LevelDB with the same set of workloads. In contrast to asynchronous mode, the difference in throughput between the two write patterns is very small for both hardware, although Figure 8 clearly shows that random writes still produce significantly more compactions (the hard disk drive is too slow we were not able to run enough operations to get compactions). Turns out with the synchronous configuration, LevelDB's throughput is bottlenecked by the storage access overhead for each write operation. We verified it by measuring the throughput of synchronously appending 512 Bytes data to a file. Figure 9 shows that LevelDB's write throughput is indeed close to that of file appending.

Synchronous mode gives LevelDB 250X more throughput running on the SSD than on the hard disk

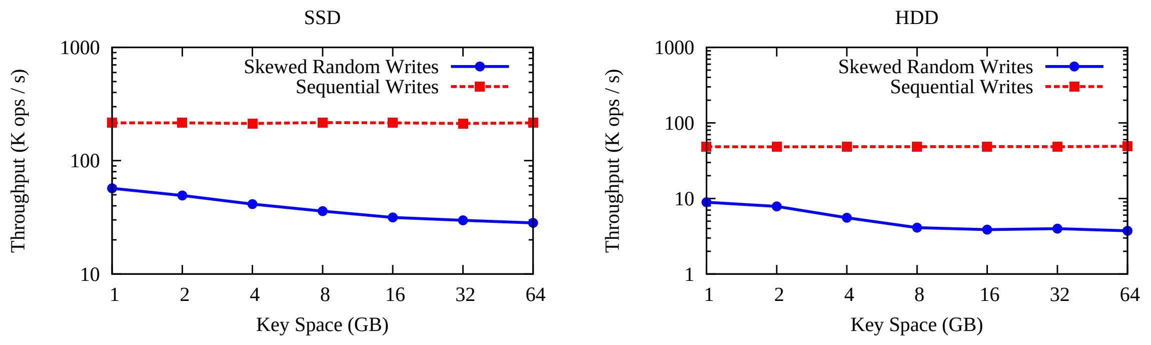

Figure 4: Comparing sequential and random 90/10 write throughput with increasing key space

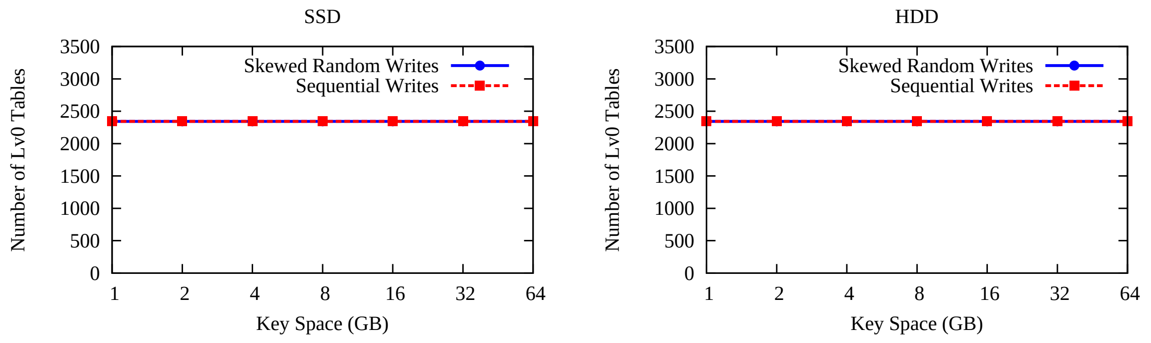

Figure 5: Number of level-0 *SSTables* created during each experiment run.

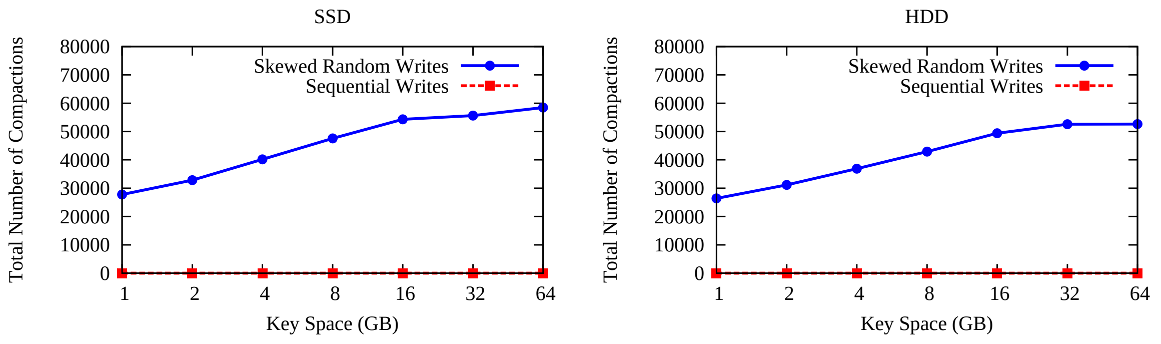

Figure 6: Total number of LevelDB compactions during each experiment run.

drive, as compared to the mere 4X difference in the asynchronous mode. The 3 orders of magnitude difference in access latency between the disk and the SSD justifies the result in the synchronous configuration, while the asynchronous mode is mostly in-memory. It is exactly this miserable performance number with synchronous writes on HDD that lead to people opt for the faster but less durable asynchronous mode. However, the continuous improvement on storage device speed is making synchronous writes more feasible. On the other hand, many of the OS storage stack design options are around a magnetic spinning disk and are not optimized for the newer technologies. SSDs have a minimum writing size (page size) of more than 2KB. When writing less than a page size, an SSD has to do a read-modify-write which is much more costly than writing a whole page. However, the OS issues writes to storage devices in sectors which are 512 Bytes. With LevelDB writing small key-value pairs, the SSD has to incur the expensive read-modify-write overhead all the time. Figure 9 demonstrates that writing to a file with exact flash page size (4KB) gives more than 4X the throughput, but only when updating not appending. The reason is that the file system has to write additional meta data to the SSD when doing file appends, and with those meta data less than a page size, it will also incur the read-modify-write penalty.

4 Problem Statement and Proposal

We believe there are 3 problems in the current storage stack design that lead to sub-optimal storage performance. In this section, we will describe the problems and propose a new stack design option.

4.1 Problems

Firstly, storage applications are explicitly making hardware-level decisions. For example, applications are specifying the actual data layout on storage hardware, including addresses, alignments, SSD erasure blocks etc.; the writing scheme depends on hardware characteristics, for instance, avoid expensive in-place updates by doing append-only writes; grouping writes into hardware specific blocks like sectors (512 Bytes) or pages (4KB). However, it is very hard for applications to get the exact hardware details it is running on, which is further hindered by storage hardware vendors who extensively hide hardware details. Consequently, applications have to make assumptions and when their assumptions are inaccurate (most of the time), the storage performance is compromised.

Secondly, as shown in section 2, different storage technologies have very different performance characteristics. Even within the same technology, different products differ in their hardware details. As a result, the aforementioned hardware-level optimizations, even if accurate on a particular hardware, may be inefficient on other products. Application designers are thus forced to redesign their system for the different hardware they are running on, or be settled with less optimal performance.

Thirdly, the current deep storage stack interferes with the hardware optimizations applications make. As described in section 1, the operating system, file system, drivers, disk controllers and disk hardware are all optimizing storage performance with limited local information. Many of these optimizations unfortunately interfere destructively with those made by the application, leading to sub-optimal performance.

4.2 Proposal

To address these three problems, we are proposing a new storage stack design. As shown in Figure 10, current applications have to specify storage hardware logic in their code. Without modifying the application, the same piece of logic will be applied to different storage devices, leading to potential sub-optimal performance. Figure 11 shows our proposed stack design. Leveraging Arrakis [14], we provide applications direct access to storage hardware, bypassing the operating system kernel and the file system, essentially solve the third problem. To deal with the first two problems, we add a middle layer between the application and the hardware. The layer exposes an object interface to the application. Applications can create, read, write, append and delete objects, and specify ordering dependencies and atomicity among objects. Essentially, we allow applications to be written in a hardware neutral way, without specifying on-disk data layout, consistency implementation, update pattern, etc. The layer then has a hardware dependent component that implements the interface for each storage device. The implementation is hardware specific, and optimizes for the target device. For example, a HDD will have a LSF like implementation while an SSD may require a more efficient COW implementation. We envision the hardware vendor who understand the device best to implement this hardware dependent component to maximize performance.

5 Related Work

5.1 Flash Characteristics

Flash memory and flash memory based solid state drives have been a hot research area in recent years. SSDs have very different hardware characteristics compare to spinning disks [6]: the minimum read/write size is a flash page which is different on different SSDs; flash pages are grouped into erasure blocks, and the hardware needs to erase the entire block before writing new data; there is a Flash Translation Layer that deals with wear leveling and hides costly erasures. SSDs also contain multiple memory chip packages that can be accessed concurrently, cre-

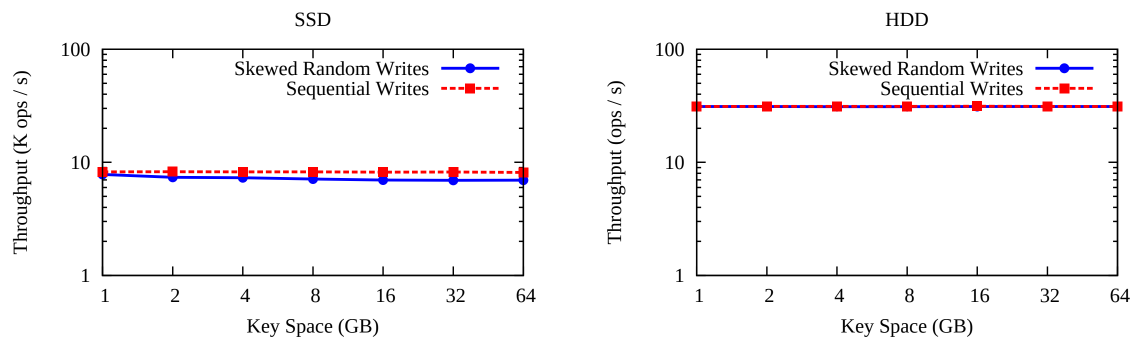

Figure 7: Comparing write throughput using synchronous write mode.

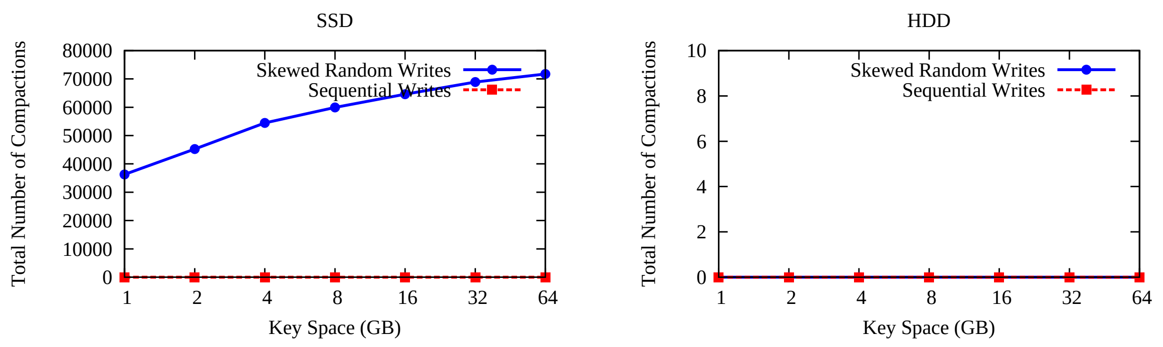

Figure 8: Total number of LevelDB compactions in synchronous mode. We did not run the HDD experiment long enough to get compactions due to its slow speed

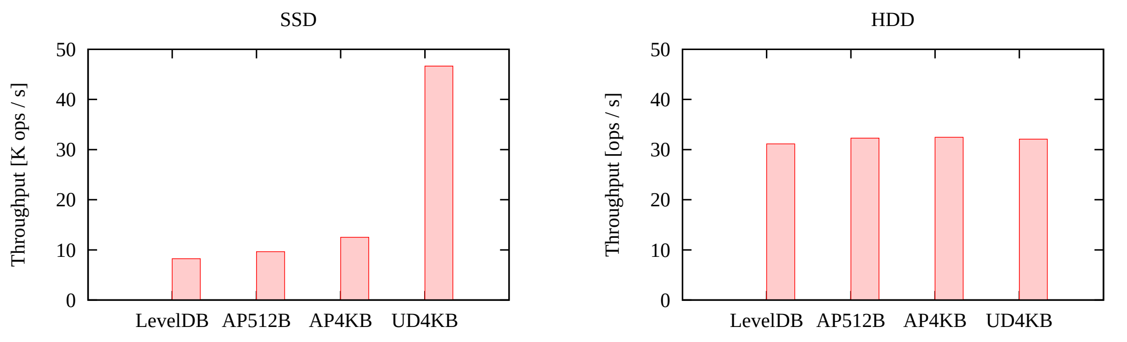

Figure 9: Comparing LevelDB synchronous throughput to file writes with different write sizes. AP denotes appends and UP is updates.

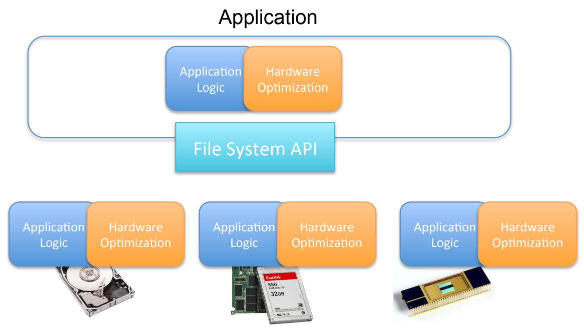

Figure 10: Current storage stack. Application programmers have to specify storage hardware logic in their application code, including data-layout, write pattern, block sizes etc.

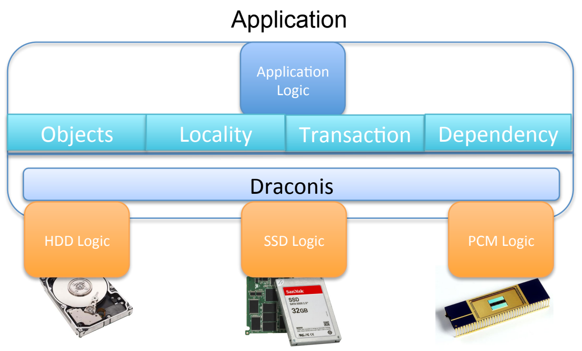

Figure 11: Our proposed stack design. Applications can use the Draconis interface to specify machine independent logic without constraining storage hardware specific decisions. The interface will have a hardware dependent component for each individual storage device.

ating chances of internal parallelisms [4]. Performance characteristics vary significantly across SSDs [3] [10]. For example, sequential and random access patterns have very different latency and throughput effects on different SSDs.

5.2 Storage Systems Built for SSD

We are not the first one who identify that applications written for spinning disks are not efficient on SSDs. NVMKV [12] directly uses FTL in SSDs to maintain key value mappings, minimizing meta data at the KV layer and bypassing the file system. DFS [9] similarly uses FusionIO SSD's virtualized flash storage layer for file block allocations and reclamations. Both FlashStore [5] and Silt [11] avoid the expensive random updates on SSDs by writing all updates as a log.

5.3 Storage Stack Redesign

Arrakis [14] reduces storage stack overhead by giving applications direct accesses to the storage hardware, bypassing the OS kernel and the file system. However, Arrakis does not solve the issue of efficient use of the hardware with different performance properties. There is a also long line of research that provides applications a customized storage stack, like Exokernel [8], Nemesis [1] and SPIN [2]. However, they do not go beyond the mechanism and propose any actual stack design that optimizes I/O performance on different hardware.

References

[1] P. R. Barham. Devices in a multi-service operating system, 1996.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.

[3] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[4] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 266–277, Washington, DC, USA, 2011. IEEE Computer Society.

[5] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, Sept. 2010.

[6] P. Desnoyers. What systems researchers need to know about nand flash. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, pages 6–6, Berkeley, CA, USA, 2013. USENIX Association.

[7] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, Feb. 1989.

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[9] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. Dfs: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association.

[10] M. Jung and M. Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 203–216, New York, NY, USA, 2013. ACM.

[11] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[12] L. Mármol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, pages 8–8, Berkeley, CA, USA, 2014. USENIX Association.

[13] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[14] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.

[15] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM.