2016 IEEE 40th Annual Computer Software and Applications Conference

Designing SSD-friendly Applications for Better Application Performance and Higher IO Efficiency

Zhenyun Zhuang, Sergiy Zhuk, Haricharan Ramachandra, Badri Sridharan
2029 Stierlin Ct, Mountain View, CA 94043, USA
{*zzhuang, szhuk, hramachandra, bsridharan*}*@linkedin.com*

***Abstract*—SSD (Solid State Drive) is being increasingly adopted to alleviate the IO performance bottlenecks of applications. Numerous measurement results have been published to showcase the performance improvement brought by SSD as compared to HDD (Hard Disk Drive). However, in most deployment scenarios, SSD is simply treated as a "faster HDD". Hence its potential is not fully utilized. Though applications gain better performance when using SSD as the storage, the gains are mainly attributed to the higher IOPS and bandwidth of SSD.**

The naive adoption of SSD does not utilize SSD to its full potential. In other words, application performance improvements brought by SSD could be more significant if applications are designed to be SSD-friendly. In this work, we propose a set of 9 SSD-friendly design changes at application layer. Applying the design changes by applications can result in three types of benefits: (1) improved application performance; (2) increased SSD IO efficiency; and (3) longer SSD life.

***Keywords*-SSD; Application design; Performance; IO efficiency**

I. Introduction

SSD (Solid State Drive) is seeing increasingly wider deployments in various scenarios, thanks to the higher IO performance compared to its HDD (Hard Drive Disk) counterpart. Many people treat SSD as the silver bullet to cure the IO bottlenecks faced by many applications [21], [25], [31]. Numerous measurements and reports have showcased the performance improvement of various types of applications brought by replacing HDD with SSD. These application-level performance improvements are directly attributed to the higher IOPS and bandwidth performance of SSD when compared to HDD.

However, in most of these deployments, SSD is simply treated as a "faster HDD", hence its potential is not fully utilized. In other words, the application performance improvements brought by SSD can be even more significant. SSD has its unique internal working mechanisms that differ from HDD. By explicitly taking advantage of these characteristics, various tiers of today's computing infrastructure can be optimized for using SSD. Though many SSD-friendly design changes have been seen in the software tiers of File System, Database, data structure and data infrastructure, relatively little work has been done to fully examine the required application-tier changes to accommodate the use of SSD.

In this work, we focus on the design changes at application tier that are brought by SSD. We propose totally 9 such application-tier design changes specifically for SSD being the storage. These "SSD-friendly" design changes, if adopted appropriately, are expected to significantly improve the performance of the particular applications that follow the designs.

The benefits of these design changes can go beyond the particular application that adopts the changes. As we will demonstrate later, SSD-friendly applications can result in higher IO efficiency at storage layer, hence allowing more applications to share the same storage tier. Such denser deployments will help save business costs by more efficiently utilizing the storage tier. Moreover, SSD is costly and wears out due to writing. SSD-friendly applications can also help save SSD life, which further reduce operation cost.

The remaining paper is organized as follows. We first motivate the SSD-friendly application designs by providing concrete data in Section II. We then present the internal working mechanisms of SSDs as necessary backgrounds in Section III. By being SSD-friendly, many software tiers (e.g. Database, File System) can gain benefits, we will describe them in Section IV. We propose the set of SSD-friendly application designs in Section V and discussions in Section VI. Section VII presents related works. Finally, in Section VIII, we conclude the work.

II. Motivation

Compared to the naive adoption of SSD by as-is applications, SSD-friendly application designs can can gain three types of benefits:

- **Better application performance;** For achieving higher throughput or lower response latency of the particular applications; these are the results of fully utilizing the IO capacity of SSD storage.
- **More efficient storage IO;** For achieving more efficient IO of SSD storage; if the underlying SSD can be more efficiently used, the same SSD can support more IO requests, possibly from multiple co-located applications. In other words, it allows denser deployment of applications.
- **Longer SSD life** For saving SSD life; SSD wears out with IO writing. Running SSD-friendly applications

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.94

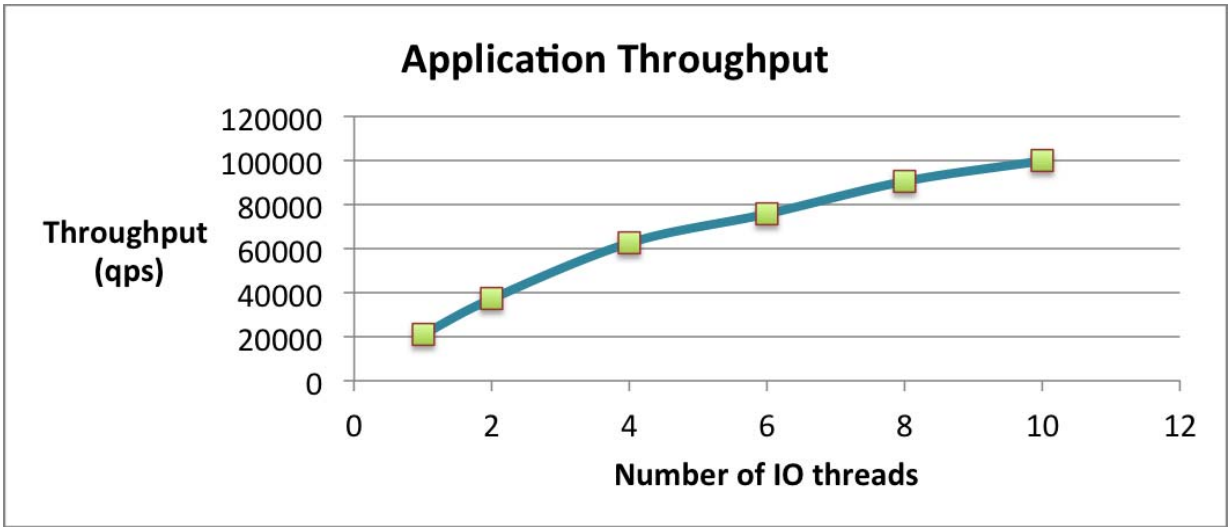


Figure 1. **Naive adoption of SSD is sub-optimal**

also help save SSD life, which reduces the business cost and saves a lot of trouble caused by dead SSD.

A. Better application performance

The first reason why SSD-friendly software designs are needed is: software application could have sub-optimal performance if not being SSD-friendly. Though moving from HDD to SSD typically mean better application performance, thanks to the better IOPS performance of SSD, a naive adoption of SSD (without changing application designs) may not achieve optimal performance.

We have an application that writes to files to persist data and is IO-bound. The maximum application throughput when working with HDD is 142 qps (queries per second). This is the best result we can get, irrespective of various changes/tunings on the application design. When moving to SSD with the as-is design, the throughput is increased to 20 K qps. The 140X performance improvement mainly comes from the higher IOPS provided by SSD. Though the application throughput is significantly improved when compared to HDD, it is not the best it can potentially achieve.

After optimizing on the application design and make it SSD-friendly, the throughput is increased to 100 K qps, a 4X further improvement when compared to the naive SSD adoption. The secret source for this particular example is by using multiple concurrent threads to perform IO, which takes advantage of SSD's internal parallelism that we will describe later, as shown in Figure 1.

B. More efficient storage IO

The minimum internal IO unit on SSD is page (e.g., 4KB [1]), hence a single byte's access (read/write) to SSD has to happen at page level. Partly due to this, application's writing to SSD can result in larger physical write size on SSD media, an undesirable phenomenon referred to "write amplification (WA)" [8]. The ratio is referred to as "write amplification factor" or WA factor.

The extent of write amplification is affected by many mechanisms, such as FS (file system) factor. For instance, a simple command of *echo "SSD" > foo.txt* has only three effective bytes, but SSD needs to write at least 44 KB to the storage. Because of the WA factor, SSD may be substantially under-utilized if the data structure or the IO issued by applications is not SSD-friendly. Though a typical SSD can do up to 1GB/s IO throughput, if the application IO pattern is not SSD-friendly, in reality, a mere few MB/s application IO rate may saturate the SSD.

This problem is more apparent in case of multiple co-located applications sharing the same SSD storage. If the applications are IO-bound, how effectively the storage IO is used directly affects how many applications can be co-deployed.

C. Longer SSD life

SSD cells wear out, and each cell can only sustain limited number of P/E (Program/Erase) cycles. For off-the-shelf mainstream MLC, it only has about a few thousands of P/E cycles. Practically, the life of a SSD depends on four factors: (1) SSD size, denoted by S; (2) Maximum number of P/E cycles, denoted by W; (3) Write amplification factor, denoted

[1]Most SSD products use 4KB as the page size; others use even larger size.

by F; and (4) Application writing rate, denoted by R. With these, the life of a SSD can be calculated by $\frac{SW}{FR}$.

Specifically, Table I shows the life of a SSD in different scenarios. Assuming SSD has 1 TB size and application write rate is 100 MB/s. Consider a MLC (Multi-Level Cell) SSD with the number of P/E cycles being 10K, a typical value for today's mainstream MLC SSDs. When the write amplification factor is 4, it only lasts for 10 months. If the WA factor is increased to 10, the same SSD can only sustain 4 months. For a TLC (Triple-Level Cell) SSD, with reduced P/E cycles of 3K, WA of 10, it only lasts for 1 month. Given the high cost of SSD, the applications are desired to be SSD-friendly for the purpose of saving SSD life.

III. Background

To fully appreciate the later-presented SSD-friendly application designs, let's visit some necessary backgrounds with regard to how SSD works internally.

A. Overview

The mainstream SSD today is NAND based, which stores digital bits in cells. Each SSD cell can store 1 bit (SLC, Single Level Cell), 2 bits (MLC, Multiple Level Cell), 3 bits (TLC, Triple Level Cell) or even 4 bits. Cells can only sustain certain numbers of erasures. The more bits a cell stores, the less manufacturing cost, but significantly reduced endurance (e.g., number of erasures).

A group of cells compose a page, which is the smallest storage unit to read or write. A typical page size is 4KB. Once erased (i.e., initialed or reset), a page can only be written once. So essentially there is no "overwriting" operation on SSD. Pages are grouped into blocks. A block is the smallest unit to erase. The typical size of blocks is 512KB or 1MB, or about 128 or 256 pages.

The brain of a SSD is the controller, which manages the pages/blocks of SSD. SSD controller contains all the intelligence to make SSD work efficiently. For instance, it maintains a FTL (Flash Translation Layer) which maps physical pages to logical pages. FTL is used for various benefits including wear leveling and garbage collection.

B. IO operations and GC

There are three types of IO operations: reading, writing and erasing. SSD Reading is in the unit of pages. Reading is typically faster than writes, and the random reading latency at microsecond level, and is almost constant. Writing is in the unit of pages, and the writing latency could be varying, depending on the historical state of the disk. Erasing is in the unit of blocks. Erasing is slow, typically a few millisecond.

Garbage Collection (GC) in SSD is needed to recycle used blocks and ensure fast allocation of later writes. Maintaining a threshold of free blocks is necessary for prompt write response because SSD cannot afford on-the-fly block erasures due to the slow erasure process. Typically SSD controller has a threshold of how many free blocks should be present. If the number of free blocks drops below the threshold, SSD needs GC to free more blocks by erasing them. During GC, blocks that have the largest number of unused pages are initialized (i.e., erased). The controller firstly backs up valid pages in a block and then initializes the block to an empty block. GC can happen in background (i.e., offline) or foreground (i.e., online), depending on SSD design and its state. In particular, if the writing activity is so heavy that background GC is unable to catch up, then foreground GC will occur, which will significantly affect the writing performance.

Table I
Expected SSD life (Assuming SSD size is 1TB, Application write rate is 100MB/s)

SSD Type	P/E Cycles	WA Factor	Life (Months)
MLC	10K	4x	10
MLC	10K	10x	4
MLC	3K	10x	1

C. Wear leveling and Write amplifications

SSD blocks have limited erasure times (or P/E cycles). Some blocks may host very active data, hence they are erased frequently. Once the maximum number reaches, the block dies. The typical number of P/E cycles for SLC is 100K, MLC 10K, and TLC a few K. To ensure capacity and performance, the blocks need to be balanced in terms of how many erasures have been done. SSD controllers have such a "wear leveling" mechanism to achieve that. During wear leveling, the data are moved around among blocks to allow for balanced wear out.

D. Internal parallelism

Though a typical SSD can achieve up to several GB/s IO rate and hundreds of thousands of IOPS, a single NAND-flash IO bus has much limited bandwidth. The IO latency of reading is up to 50 microseconds, and IO latency of writing is up to several hundreds of microseconds. To achieve higher bandwidth and lower latency, SSD relies on internal parallelism of different levels. The first level of parallelism is channel-level, the flash controller communicates with flash packages through multiple channels. Also, multiple packages can access the same channel, which is package-level parallelism. A package may contain multiple chips, and a chip may contain multiple planes; these are chip-level and plane-level parallelism.

IV. SSD-friendly design changes at other software layers

There are several computing tiers where SSD-friendly designs have taken place. We now present such design changes that happened at File System tier, Database tier and Data Infrastructure tier.

Table II
THE WORLD OF **HDD**: FAVORING REMOTE MEMORY ACCESS

Performance Metric	Local Disk Access	Remote Memory Access
IO Latency	A few milliseconds	A few microseconds
IO Bandwidth	e.g., 100 MB/s	120 MB/s (Gbit NIC), 1.2 GB/s (10Gbit NIC)

Table III
THE WORLD OF **SSD**: FAVORING LOCAL DISK ACCESS

Performance Metric	Local Disk Access	Remote Memory Access
IO Latency	A few microseconds	A few microseconds
IO Bandwidth	up to 1GB/s	120 MB/s (Gbit NIC), 1.2 GB/s (10Gbit NIC)

A. SSD-friendly designs at File System tier

File system (FS) deals with storage directly, so naturally SSD-friendly design changes are needed to accommodate SSD [22], [29]. Conventional file systems is optimized to work with HDD; they typically don't take into considerations of SSD characteristics, hence resulting in suboptimal performance when working with SSD. Generally speaking, the design changes at file system tier are focused on the three key different characteristics of SSD: (1) Random access is on par with sequential access; (2) Blocks need to be erased for overwriting; (3) SSDs internal wear leveling causes write amplification.

There are two types of SSD-friendly file systems: (1) General FS adapted for SSD. The adaptation mainly supports the new feature of TRIM. Examples include Ext4, JFS [2] and Btrfs [26]. (2) There are also FSs that are specially designed for SSD. The basic idea is to adopt log-structured data layout (vs. B-tree of H-tree) to accommodate SSD's "copy-modify-write" property. Examples are ExtremeFFS from SanDisk, NVFS (Non-volatile file system) [7], JFFS/JFFS2 [5] and F2FS [22].

B. SSD-friendly designs at Database tier

The differences between HDD and SSD are particularly important in Database design, because Database components such as query processing, query optimization and query evaluation have been tuned for decades with the HDD characteristics in mind. For instance, random accesses are considered much lower than sequential access. With SSD, many of the assumptions do not hold, hence SSD-friendly Databases are designed.

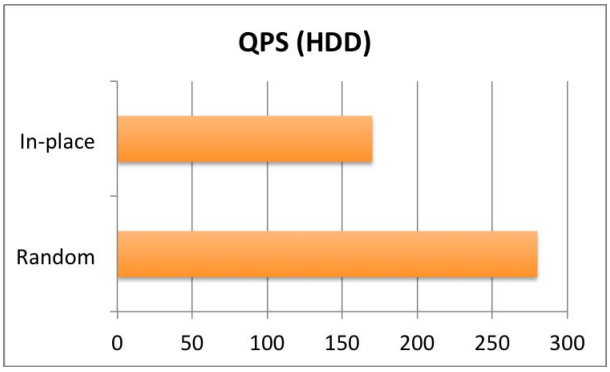
There are mainly two types of Databases that work well with SSD [18], [27], [30]: (1) Flash only database, which benefit greatly from flash-friendly join algorithms. One example is AreoSpike; (2) Hybrid flash-HDD database, which uses flash judiciously to cache data. The key design changes include the following: (1) increased IO concurrency; In the world of HDD, 1 thread per database connection is optimal, but in SSD world, this treatment is suboptimal. (2) data structure change; Similar to File System change, B-tree data structure gives way to Log-structured tree. (3) data layout. With SSD, locality still matters, but matters differently. Page or block aligned locality is much preferred. Hence many database use column-oriented data layout rather than the traditional Row-oriented layout.

C. SSD-friendly design changes at Data Infrastructure tier

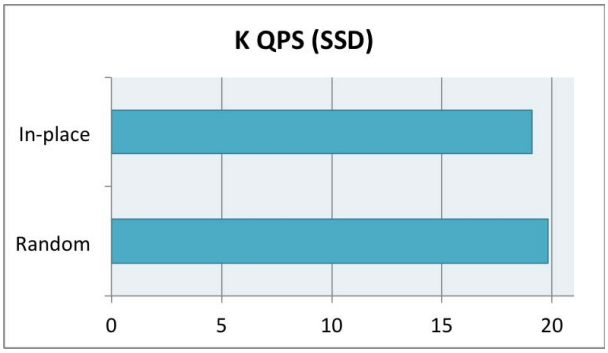
Many of the assumptions in Data Infrastructure tier also need to revisited [12], [17], [20]. For distributed data systems, there is a debate over where to load data: from local disk on the same machine or from remote memory on another machine. The argument used to favor remote memory access from another machine. If data need to be put to RAM, loading from remote memory is more favorable than loading from local disk, even though this design adds additional network hops, deployment complexity and operation cost. The reason for such argument is that remote memory access usually is faster and cheaper than local HDD.

Traditional HDD access latency is in the order of a few milliseconds, while the latency of remote memory access, which includes both RAM access latency and networking transmission latency, is only at microsecond level. For small data size (e.g., 1KB) and 1Gbps NIC card, the networking transmission latency is only a few microseconds; while the RAM access latency is at nanosecond level. Because of this, remote memory access is about 10X faster than local disk access. The IO bandwidth of remote memory access is on par with or faster than local disk access. RAM bandwidth can easily be multiple GB/s, while 1Gbps NIC card can achieve up to 120MB/s, and 10Gbps NIC can achieve 1GB/s IO bandwidth, but local HDD access typically has about 100MB/s. The comparison results are shown in Table II.

With SSD being the storage, local disk access (SSD) is more favorable than remote memory access, as local disk access is cheaper and faster than remote memory access. The IO latency of SSD is reduced to microsecond level, while the IO bandwidth of SSD can be up to several GB/s. The comparison results are shown in Table III. These results motivate new designs at Data Infrastructure tier: co-allocating data with the applications as much as possible to avoid additional nodes and associated network hops. This not only reduces deployment complexity, but also reduces business cost.



(a) HDD



(b) SSD

Figure 2. **In-place updates vs. Random updates**

Let us visit an example scenario where Cassandra uses memcached [16] as the caching layer, a true use case experienced by Netflix [10]. In this case, memcached is used to cache data behind Cassandra layer. Assuming 10TB of Cassandra data need to be cached. If each memcached node holds 100GB of data in RAM, then 10TB of data requires 100 memcached nodes. With 10 Cassandra nodes, it is possible to completely remove memcached layer by equipping each Cassadra node with a 1TB SSD. Instead of having 100 memcached nodes, only 10 SSD are needed - a huge saving on the cost and deployment complexity. The query latency performance, as reported, is on par with memcached being used. What is more, the SSD-based infrastructure is found to be more scalable - the new IOPS capacity is much higher.

V. SSD-friendly Application Design

At application tier, we can also make SSD-friendly design changes to gain the three types of benefits (i.e., better application performance, more efficient IO, and longer SSD life). In this work we present a set of 9 SSD-friendly design changes at application tier. These changes are organized into 3 categories: data structure, IO handling and threading, as listed below.

- *Data structure:* (1) Avoid in-place update optimizations; (2) Separate hot data from cold data; (3) Adopt compact data structure;
- *IO handling:* (1) Avoid long heavy writes; (2) Prefer not mixing write and read;; (3) Prefer large IO aligned on pages; (4) Avoid full disk usage;
- *Threading:* (1) Use multiple threads (vs. few threads) to do small IO; (2) Use few threads (vs. many) to do big IO.

A. Data structure: *Avoid in-place updating optimizations*

Conventional HDD storages are known to have substantial seeking time, hence applications that use HDDs are oftentimes optimized to perform in-place updating that does not require seeking. For instance, as shown in Figure 2(a), a workload that persists data to storages has significantly different IOPS performance for random updating vs. in-place updating. When performing random updating, the workload can only achieve about 170 qps; while for the same HDD, in-place updating can do 280 qps, much higher than random updating. When designing applications to work with SSD, such concerns are not valid anymore. In-place update does not gain any IOPS benefit compared to non-in-place update.

Moreover, in-place updating actually incurs performance penalty on SSD. SSD pages containing data cannot be overwritten, so when updating the stored data, the corresponding SSD page has to be read into SSD buffer first. After the updates are applied, the data then are written to a clean page. The process of "read-modify-write" in SSD is in sharp contrast to direct "write-only" in HDD. By comparison, a non-in-place update does not incur the reading and modifying steps (i.e., only "write"), hence is faster. As shown in Figure 2(b), for the same workload, either random update or in-place update can achieve about 20K qps.

In addition to the performance disadvantage, unnecessary reading from SSD actually hurts due to a symptom referred to as read-distrubance [4]. Reading a cell may affect the neighboring cells that store data, hence increases data corruption rate. Because of this, unless the benefit of in-place updates (e.g., design simplicity) prevails, in-place updating data structures should be avoided.

B. Data structure: *Separate hot data from cold data*

For almost all applications that deal with storage, the data stored on disks are not accessed with equal probabilities. Considering a social network application which needs to track active user's activities. For the storage of users' data, a naive solution would simply compact all users in the same place (e.g., files on SSD) based on trivial properties such as registration time. Such a solution is straightforward to come up with and may simplify the business logic, however it may not be optimal from the SSD's standpoint.

It is well known that user activities differ in their "hotness" (i.e., popularity) - some users are more active than

others. When updating the activities of hot users, SSD needs to access (i.e., read/modify/write) at page level. So if a user's data size is less than a page, the nearby users' data will also be accessed together. If nearby users' data are not needed, the bundled IO not only waste IO bandwidth, but also unnecessarily wear out the SSD.

To alleviate the above performance concern, hot data should be separated from cold data when using SSD as the storage. The separation can be done at different levels or different ways: that is, different files, different portions of an file, or different tables.

The degree of benefits by separating hot data from cold data directly depends on the ratio between hot data and cold data inside SSD pages. Assuming a 1:2 ratio of hot vs cold data, then separating them likely will increase the SSD life by 2X, while improving the IO performance by 2X when accessing the hot data.

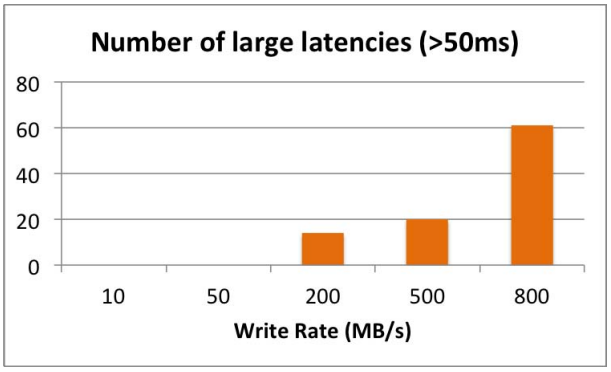
C. Data structure*: Adopt compact data structure*

The smallest updating unit in SSD world is the page (e.g., 4KB), hence, even a single-bit update will results in at least 4KB SSD write. Note that the actually written bytes to SSD could be far larger than 4KB, due to the write amplification. Reading is similar, a single byte read will result in at least 4KB reading. Note again that the actually read bytes could be much larger than 4KB, since OS has read-ahead mechanism to aggressively read in data beforehand in the hope of improving cache-hit when reading files.

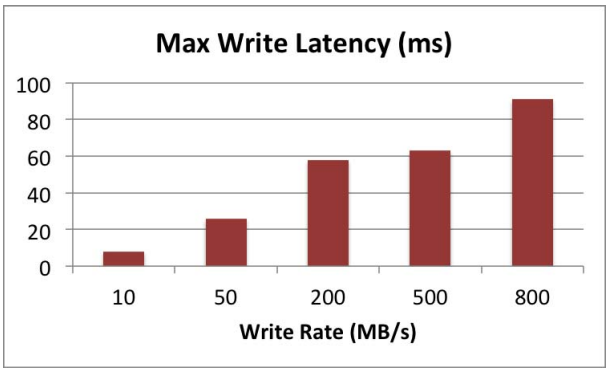
Such read/write characteristics favor the use of compact data structure when persisting data on SSD. A SSD-friendly data structure should avoid scattered updates. Compact updates can significantly increase SSD life by less wearing out the media. For instance, assuming an application updates an user's profile of 4KB size (i.e., page size), a compact data structure which stores the users' profiles in a single file only needs a single SSD page write. On the contrary, a non-compact data structure would scatter the user's profiles in 100 different files, each containing a small field (e.g., age) of the user profile. With such a scattered data structure, each user update will likely need to write to all the 100 files and results in writes to 100 SSD pages. The benefits of using compact data structure are faster application performance, more efficient storage IO, and also saving SSD life.

D. IO handling*: Avoid long heavy writes*

SSD typically features GC mechanisms to collect blocks offline for later use. GC can work in background or foreground fashion. SSD controller typically maintains a threshold value of free blocks. Whenever the number of free blocks drops below the threshold, background GC will kick in. Since background GC happens asynchronously (i.e., non-blocking), it does not affect application's IO latency when writing to SSD. If however, the requesting rate of blocks exceeds the GC rate, then background GC will not be able to keep up, and foreground GC will be triggered.



(a) Number of large write latencies



(b) Maximum write latency

Figure 3. **Impacts of long heavy writes**

During foreground GC, each block has to be erased on the fly for applications to use (i.e., blocking). Since block erasure is much slower than typical write (on free blocks), the write latency experienced by applications which issue writes will suffer. Specifically, a typical write IO operation takes about hundreds of microseconds, but an foreground GC operation that frees a block could take more than several milliseconds, resulting much larger IO latency. For this reason, it is better to avoid long heavy write such that the foreground GC is never kicked in.

We conducted experiments to see how the write rate affects the IO performance, and the results are shown in Figure 3. We vary the write rate from 10MB/s to 800 MB/s and run for 2 hours for each write rate. For each test case, we count the max write latency, as well as the number of "large latencies" (that is, latency larger than 50 ms). When the write rate is light at 10MB/s, the largest IO latency is 8 ms; the latency increases with higher write rates, and it is 92 ms when write rate is 800MB/s. The large latencies that are more than 50ms are not observed when the write rate is 10MB/s or 50MB/s. For 800MB/s write rate, 61 such latencies are observed.

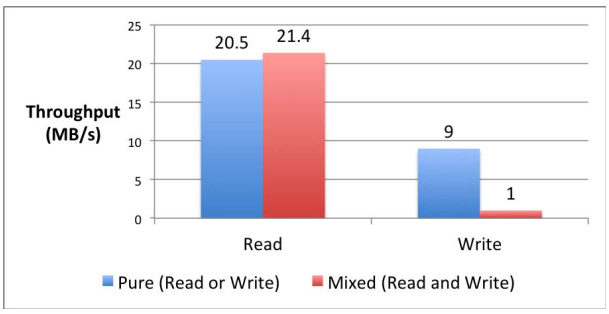


Figure 4. **Write/Read rate in different scenarios (Random IO, 102 bytes each)**

E. IO handling*: Prefer not mixing write and read*

Read and write share some critical resources internally on SSD and may interfere each other. Mixing read and write traffic will render both directions being suboptimal for several reasons. Specifically, first, all IO share certain critical resources, notably the lock-protected mapping table. Second, there are internal optimizations in SSD that is uni-directional. For instance, pipelining of moving data between cache register and controller and moving pages between flash and data register. Mixing read and write will render these pipelines ineffective. Third, multiple IO generate certain background activities such as OS readahead and write-back, these activities will interfere with each other.

IO workload with mixed write and read will hurt either write or read or both, depending on SSD implementations and application workloads. Here we show one set of result with a particular SSD in Figure 4. The workload is random small IO of 102 bytes. When doing purely writing, it can achieve 9MB/s throughput; and pure reading achieves 20 MB/s throughput. When we mix the IO workloads of reading/writing in a half-and-half fashion, the reading throughput is largely the same as in pure-reading case, but writing throughput is much lower, only 1MB/s, or a 9X degradation.

F. IO handling*: Prefer lage IO, aligned on pages*

IO operations of read and write can only happen at page level in SSD, while GC erase happens at block level. So application IO size is preferred to be at least a page size. Internally, some SSDs use "clustered blocks", which constructs an IO unit that spans multiple SSD planes. The use of clustered blocks can increase the internal parallelism, as when read/write happens, multiple planes can be accessed simultaneously. So if possible, applications should do IO at even bigger size than pages or even blocks. The benefits of doing large IO are increased IO efficiency, reduced WA factor, and higher throughput.

Due to the page-level read/write, without aligning on pages, SSD will need to read/write more than necessary pages. For instance, a 4KB IO that is aligned on a page only access one page, while the same IO size will need to 2 pages if not aligned on pages. Such inefficiency will result in longer access time, consume more resources and wear out SSD faster.

G. IO handling*: Avoid full SSD usage*

SSD usage level (i.e., how full the disk is) impacts the write amplification factor and the writing performance caused by GC. During GC, blocks need to be erased to create free blocks. Erasing blocks requires preserving pages that contain valid data in order to obtain a free block. Creating a single free block may require compacting more than one blocks, depending on how "full" the blocks are.

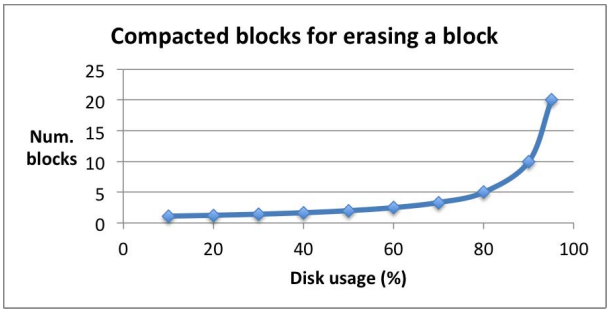
The number of blocks required to be compacted to free a block is determined by how full the disk is. Assuming the full percentage is $A\%$, on average, to free a block, $\frac{1}{1-A}$ blocks will need to be compacted. Apparently, the higher usage a SSD is, the more blocks will be moved around to free a block, which takes more resources and results in longer IO wait. For instance, If $A = 50\%$, only 2 blocks are compacted to free a block. If $A = 80\%$, about 5 blocks data are moved around to free one block. A shown in Figure 5(a), when $A = 95\%$, about 20 blocks are compacted.

Looking at the number of pages that need to be preserved during GC, the impact is even higher. During GC, the live pages that contain valid data need to be copied around. Assuming each block has P pages and all occupied pages contain live data, then each Garbaged Collected block requires copying $\frac{PA}{1-A}$. Assuming $B = 128$, when $A = 50\%$, each block requires copying 128 pages; while when $A = 80\%$, 512 pages. As shown in Figure 5(b), when $A = 95\%$, it is 2432 pages!

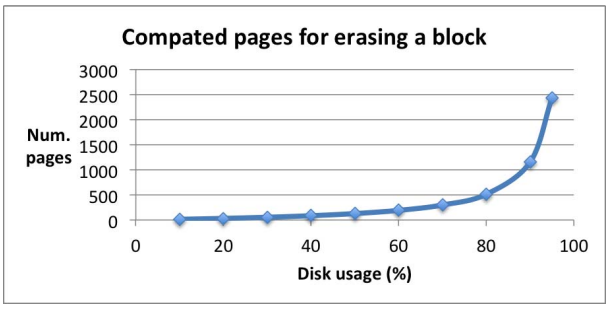
In summary, to ensure GC efficiency, SSD should not be fully occupied by live data. Note that over-provisioning is another way to preserve some free blocks for later use. In long heavy write scenarios, these preserved blocks will serve as a "buffer pool" to absorb the write bursty. Over-provisioning level determines how long and heavy the writes can be before the foreground GC kicks in.

H. Threading*: Use multiple threads (vs. single thread) to do small IO*

SSD has multiple levels of internal parallelism: channel, package, chip and plane. A single IO thread won't be able to fully utilize these parallelism, resulting in longer access time. Using multiple threads can take advantage of the internal parallelism. Wide support of NCQ (Native Command Queuing) on SSD can distribute read and write operations across the available channels efficiently, hence providing high level of internal IO concurrency. Hence it is very beneficial to keep the queue-length significantly large. For instance, we used an application to perform 10KB write IO, the results are shown in Figure 6. Using one IO thread, it achieves 115MB/s. Two threads basically doubles the



(a) Number of blocks



(b) Number of pages

Figure 5. **Impacts of disk usage level (Number of compacted blocks/pages for garbage collecting a single block)**

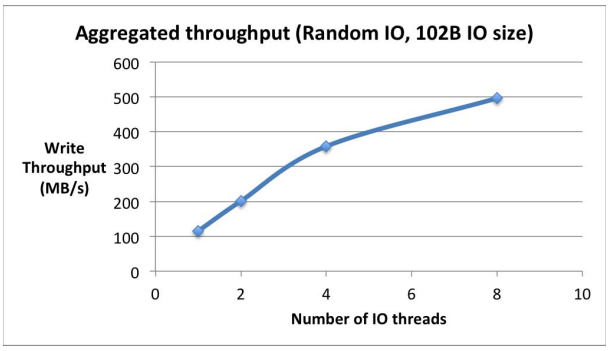


Figure 6. **Preferring multiple threads when doing small IO**

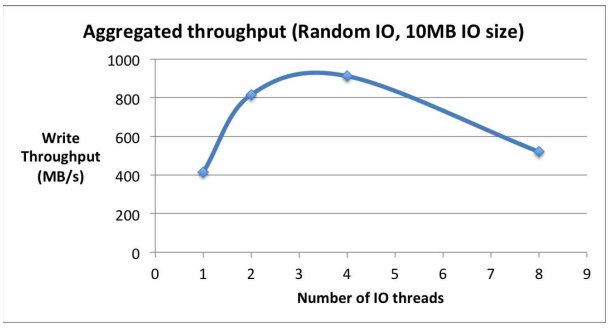


Figure 7. **Preferring few threads when doing big IO**

throughput; and 4 threads doubles again. Using 8 threads achieves about 500MB/s.

A natural question is: how small is "small". The answer is that any IO size that cannot fully utilize the internal parallelism is considered "small". For instance, with 4KB page size and parallelism level of 16, the threshold is 64KB.

I. Threading*: Use few threads (vs. many threads) to do big IO*

This design change is just the other side of the picture of depicted in the previous design change. For large IO, the SSD's internal parallelism can be taken advantage of, hence a few threads (i.e., 1 or 2) should suffice to achieve maximum IO throughput. Throughput-wise, many threads don't see benefits.

Moveover, multiple threads will incur other problems and overheads. For instance, the resource competition between threads (e.g., SSD mapping table) and the interfered background activities such as OS-level readahead and write-back are all examples of complications. For instance, based on our experiments shown in Figure 7, when the write size is 10MB, 1 thread can achieve 414MB/s, 2 threads 816MB/s, while 8 threads actually drops to 500MB/s.

VI. Discussion

A. Application layer transparency

High-end storages such as SSD are critical for supporting high-performance computing. This work presents a set of possible design changes in application development to bring SSD to its higher potential. However, there are tradeoffs regarding applying these design changes with regard to the computing design rationales. From the perspective of layer-separation, it might defeat the layer cleanness by tieing application designs too close with the underlying hardware or specific architecture even for the sake of performance. For such concerns, we argue that such application design changes are more appropriate when other forms of optimizations are not available and the storage layer is the performance bottleneck.

B. Storage management software

Many SSD management software may achieve certain amount of performance gains by performing various types of optimizations and hiding hardware details to applications. There are a fleet of storage layer solution providers supplying all sorts of storage products by striking the performance tradeoffs at different places [9]. We have studied many such products, and believe these storage-layer solutions can complement our application-layer design changes to get the maximum performance out of the storage hardware. We envision many future works along this line.

VII. Related Work

A. Internal Design of SSD

There are dozens of SSD vendors in today's market [6], and most of the internal mechanisms of their respective SSD products are proprietary. The SSD controller [3] , where almost all intelligence such as GC and Wear Leveling [8], [19] resides, is the brain of a SSD. Some work analyzes the design tradeoffs inside SSD design [13], [28]. Though NAND-based SSD dominates current flash market, new types of SSD are being developed to have even better hardware performance. One of the promising types is Intel/Micron's 3D XPoint Technology, which promises 1000X higher IO capacity than today's NAND SSD [11].

B. Performance improvement of adopting SSD

High IO capacity of SSD has caused changes and evolutions at various software tiers. New types of File systems [22], [29], databases [18], [27], [30] and data structures [12], [17], [20] to take better advantage of SSD have been developed. The adoption of SSD also caused changes to data infrastructure [1]. As an example, Netflix published a blog demonstrating how SSD helps remove *memcached* layer in their Cassandra infrastructure [10]. To fully utilize the potential of SSD, software changes, as well as co-designing software and hardware, have been proposed [14], [15], [23], [24].

VIII. Conclusion

SSD is increasingly being adopted. Applications adopting SSD generally see better performance than using HDD, however the naive adoption of SSD without changing application designs cannot achieve optimal performance because SSD works differently from HDD. To fully take advantage of the performance potentials brought by SSD, SSD-friendly application designs are needed.

This work proposes a set of SSD-friendly design changes at application layer; these designs can also help drive storage-layer and cross-layer designs.

References

[1] Ssds and distributed data systems, http://blog.empathybox.com /post/24415262152/ssds-and-distributed-data-systems.

[2] Jfs for linux, http://jfs.sourceforge.net/.

[3] Flash memory controller, https://en.wikipedia.org/wiki /Flash_memory_controller.

[4] Solid state drive, https://en.wikipedia.org/wiki /Flash_memory_controller.

[5] Jffs2, https://en.wikipedia.org/wiki/JFFS2.

[6] Solid state drive manufacturers, https://en.wikipedia.org/wiki /List_of_solid-state_drive_manufacturers.

[7] Non-volatile file system, https://en.wikipedia.org/wiki/Non-Volatile_File_System.

[8] Write amplification, https://en.wikipedia.org/wiki/Write_amplification.

[9] Accelstor, https://www.accelstor.com/.

[10] Netflix tech blog, http://techblog.netflix.com /2012/07/benchmarking-high-performance-io-with.html.

[11] Intel and micron's 3d xpoint technology, http://www.micron.com /about/innovations/3d-xpoint-technology.

[12] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, Aug. 2009.

[13] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference*, ATC'08. USENIX Association, 2008.

[14] 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.

[15] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

[16] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124), Aug. 2004.

[17] F. Havasi. An improved b+ tree for flash file systems. In *Proceedings of the 37th International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'11, pages 297–307. Springer-Verlag, 2011.

[18] J. Jiang, J. Le, and Y. Wang. A column-oriented storage query optimization for flash-based database. In *Proceedings of the IEEE 2010 International Conference on Future Information Technology and Management Engineering*, FITME 2010, 2010.

[19] X. Jimenez, D. Novo, and P. Ienne. Wear unleveling: Improving nand flash lifetime by balancing page endurance. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14. USENIX Association, 2014.

[20] R. Jin, H.-J. Cho, S.-W. Lee, and T.-S. Chung. Lazy-split b+-tree: A novel b+-tree index scheme for flash-based database systems. *Des. Autom. Embedded Syst.*, 17(1):167–191, Mar. 2013.

[21] K. Kambatla and Y. Chen. The truth about mapreduce performance on ssds. In *Proceedings of the 28th USENIX Conference on Large Installation System Administration*, LISA'14, 2014.

[22] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, 2015.

[23] Y. Lu, J. Shu, and W. Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, 2013.

[24] K. Miyaji, C. Sun, A. Soga, and K. Takeuchi. Co-design of application software and nand flash memory in solid-state drive for relational database storage system. *Japanese Journal of Applied Physics*, 53(4S), Mar. 2014.

[25] S. Moon, J. Lee, and Y. S. Kee. Introducing ssds to the hadoop mapreduce framework. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing*, CLOUD '14, Washington, DC, USA, 2014.

[26] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[27] M. Saxena and M. Swift. Revisiting database storage optimizations on flash. *Tech Report of Computer Science at University of Wisconsin Madison*, 2010.

[28] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu. Ftl design exploration in reconfigurable high-performance ssd for server applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, 2009.

[29] J. Suk and J. No. hybridfs: Integrating nand flash-based ssd and hdd for hybrid file system. In *Proceedings of the 10th WSEAS International Conference on Systems Theory and Scientific Computation*, ISTASC'10, 2010.

[30] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, 2009.

[31] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 6:1–6:11, 2015.