FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs

Jian Huang, *Georgia Institute of Technology;* Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, and Bikash Sharma, *Microsoft;* Moinuddin K. Qureshi, *Georgia Institute of Technology*

https://www.usenix.org/conference/fast17/technical-sessions/presentation/huang

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs

Jian Huang[†] Anirudh Badam Laura Caulfield

Suman Nath Sudipta Sengupta Bikash Sharma Moinuddin K.Qureshi[†]

[†]Georgia Institute of Technology Microsoft

Abstract

A longstanding goal of SSD virtualization has been to provide performance isolation between multiple tenants sharing the device. Virtualizing SSDs, however, has traditionally been a challenge because of the fundamental tussle between resource isolation and the lifetime of the device – existing SSDs aim to uniformly age all the regions of flash and this hurts isolation. We propose utilizing flash parallelism to improve isolation between virtual SSDs by running them on dedicated channels and dies. Furthermore, we offer a complete solution by also managing the wear. We propose allowing the wear of different channels and dies to diverge at fine time granularities in favor of isolation and adjusting that imbalance at a coarse time granularity in a principled manner. Our experiments show that the new SSD wears uniformly while the 99th percentile latencies of storage operations in a variety of multi-tenant settings are reduced by up to 3.1x compared to software isolated virtual SSDs.

1 Introduction

SSDs have become indispensable for large-scale cloud services as their cost is fast approaching to that of HDDs. They out-perform HDDs by orders of magnitude, providing up to 5000x more IOPS, at 1% of the latency [21]. The rapidly shrinking process technology has allowed SSDs to boost their bandwidth and capacity by increasing the number of chips. However, the limitations of SSDs' management algorithms have hindered these parallelism trends from efficiently supporting multiple tenants on the same SSD.

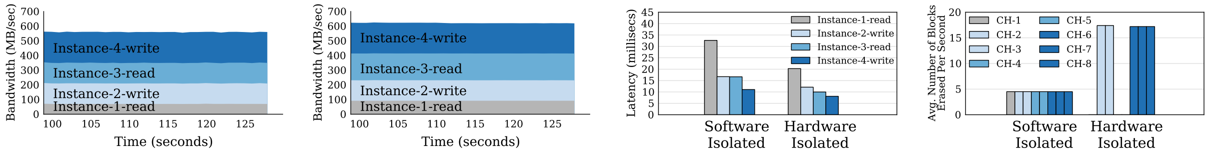
Tail latency of SSDs in multi-tenant settings is one such limitation. Cloud storage and database systems have started colocating multiple tenants on the same SSDs [14, 58, 79] which further exacerbates the already well known tail latency problem of SSDs [25, 26, 60, 78].

The cause of tail latency is the set of complex flash management algorithms in the SSD's controller, called the Flash Translation Layer (FTL). The fundamental goals of these algorithms are decades-old and were meant for an age when SSDs had limited capacity and little parallelism. The goals were meant to hide the idiosyncrasies of flash behind a layer of indirection and expose a block interface. These algorithms, however, conflate wear leveling (to address flash's limited lifetime) and resource utilization (to exploit parallelism) which increases interference between tenants sharing an SSD.

While application-level flash-awareness [31, 36, 37, 51, 75] improves throughput by efficiently leveraging the device level parallelism, these optimizations do not directly help reduce the interference between multiple tenants sharing an SSD. These tenants cannot effectively leverage flash parallelism for isolation even when they are individually flash-friendly because FTLs hide the parallelism. Newer SSD interfaces [38, 49] that propose exposing raw parallelism directly to higher layers provide more flexibility in obtaining isolation for tenants but they complicate the implementation of wear-leveling mechanisms across the different units of parallelism.

In this work, we propose leveraging the inherent parallelism present in today's SSDs to increase isolation between multiple tenants sharing an SSD. We propose creating virtual SSDs that are pinned to a dedicated number of channels and dies depending on the capacity and performance needs of the tenant. The fact that the channels and dies can be more or less operated upon independently helps such virtual SSDs avoid adverse impacts on each other's performance. However, different workloads can write at different rates and in different patterns, this could age the channels and dies at different rates. For instance, a channel pinned to a TPC-C database instance wears out 12x faster than a channel pinned to a TPC-E database instance, reducing the SSD lifetime dramatically. This non-uniform aging creates an unpredictable SSD lifetime behavior that complicates both provisioning and load-balancing aspects of data center clusters.

To address this problem, we propose a two-part wear-leveling model which balances wear within each virtual SSD and across virtual SSDs using separate strategies. Intra-virtual SSD wear is managed by leveraging existing SSD wear-balancing mechanisms while inter-virtual



(a) Software Isolated Instances (b) Hardware Isolated Instances (c) 99th Percentile Latency (d) Wear Leveling

Figure 1: Tenants sharing an SSD get better bandwidth (compare (a) vs. (b)) and tail latency as shown in (c) when using new hardware isolation. However, dedicating channels to tenants can lead to wear-imbalance between the various channels as shown in (d). Note that the number of blocks erased in the first, fourth and fifth channels is close to zero because they host workloads with only read operations. This imbalance of write bandwidth across different workloads creates wear-imbalance across channels. A new design for addressing such a wear-imbalance is proposed in this paper.

SSD wear is balanced at *coarse-time granularities* to reduce interference by using new mechanisms. We control the wear imbalance between virtual SSDs using a mathematical model and show that the new wear-leveling model ensures near-ideal lifetime for the SSD with negligible disruption to tenants. More specifically, this work makes the following contributions:

- We present a system named FlashBlox using which tenants can share an SSD with minimal interference by working on dedicated channels and dies.
- We present a new wear-leveling mechanism that allows measured amounts of wear imbalance to obtain better performance isolation between such tenants.
- We present an analytical model and a system that control the wear imbalance between channels and dies, so that they age uniformly with negligible interruption to the tenants.

We design and implement FlashBlox and its new wear-leveling mechanisms inside an open-channel SSD stack (from CNEX labs [18]), and demonstrate benefits for a Microsoft data centers' multi-tenant storage workloads: the new SSD delivers up to 1.6x better throughput and reduces the 99th percentile latency by up to 3.1x. Furthermore, our wear leveling mechanism provides 95% of the ideal SSD lifetime even in the presence of adversarial write workloads that execute all the writes on a single channel while only reading on other channels.

The rest of this paper is organized as follows: § 2 presents the challenges that we address in this work. Design and implementation of FlashBlox are described in § 3. Evaluation results are shown in § 4. § 5 presents the related work. We present the conclusions in § 6.

2 SSD Virtualization: Opportunity and Challenges

Premium storage Infrastructure-as-a-Service (IaaS) offerings [4, 7, 22], persistent Platform-as-a-Service (PaaS) systems [8] and Database-as-a-Service (DaaS) systems [2, 5, 9, 23] need SSDs to meet their service level objectives (SLO) that are usually outside the scope of HDD performance. For example, DocDB [5] guarantees 250, 1,000 and 2,500 queries per second respectively for the S1, S2 and S3 offerings [6].

Storage virtualization helps such services make efficient use of SSDs' high capacity and performance by slicing resources among multiple customers or instances. Typical database instances in DaaS systems are 10 GB – 1 TB [6, 10] whereas each server can have more than 20 TB of SSD capacity today.

Bandwidth, IOPS [48, 56] or a convex combination of both [57, 74] is limited on a per-instance basis using token bucket rate limiters or intelligent IO throttling [41, 59, 66] to meet SLOs. However, there is no analogous mechanism for sharing the SSD while maintaining low IO tail latency – an instance's latency still depends on the foreground reads/writes [25, 42, 73] and background garbage collection [34] of other instances.

Moreover, it is becoming increasingly necessary to colocate diverse workloads (e.g. latency-critical applications and batch processing jobs), to improve resource utilization, while maintaining isolation [33, 42]. Virtualization and container technologies are evolving to exploit hardware isolation of memory [11, 47], CPU [16, 40], caches [28, 52], and networks [30, 72] to support such scenarios. We extend this line of research to SSDs by providing hardware-isolated SSDs, complete with a solution for the wear-imbalance problem that arises due to the physical flash partitioning across tenants with diverse workloads.

2.1 Hardware Isolation vs. Wear-Leveling

To understand this problem, we compare the two different approaches to sharing hardware. The first approach stripes data from all the workloads across all the flash channels (eight total), just as existing SSDs do. This scheme provides the maximum throughput for each IO, and uses the software rate limiter which has been used for Linux containers and Docker [12, 13] to implement weighted fair sharing of the resources (the scenario for Figure 1(a)). Note that instances in the software-isolated case do not share physical flash blocks with other colocated instances. This eliminates the interference due to

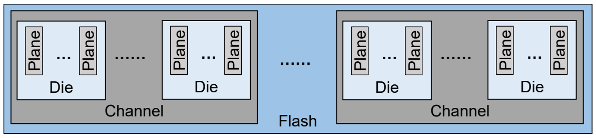


Figure 2: SSD Architecture: Internal parallelism in SSDs creates opportunities for hardware-level isolation.

garbage collection in one instance affecting another instance's read performance [34]. The second approach uses a configuration from our proposed mechanism that provides the hardware isolation by assigning a certain number of channels to each instance (the scenario for Figure 1(b)).

In both scenarios, there are four IO-intensive workloads. These workloads request 1/8th, 1/4th, 1/4th and 3/8th of the shared storage resource. The rate limiter uses these as weights in the first approach, while FlashBlox assigns 1, 2, 2 and 3 channels respectively. Workloads 2 and 4 perform 100% writes and workloads 1 and 3 perform 100% reads. All workloads issue sequentially-addressed and aligned 64 KB IOs.

Hardware isolation not only reduces the 99th percentile latencies by up to 1.7x (Figure 1(c)), but also increases the aggregate throughput by up to 10.8% compared to software isolation. However, pinning instances to channels prevents the hardware from automatically leveling the wear across all the channels, as shown in Figure 1(d). We exaggerate the variance of write rates to better motivate the problem of wear-imbalance that stems from hardware-isolation of virtual SSDs. Later in the paper, we will use applications' typical write rates (see Figure 5) to design our final solution. To motivate the problem further, we must first explore the parallelism available in SSD hardware, and the aspects of FTLs which cause interference in the first approach.

2.2 Leveraging Parallelism for Isolation

Typical SSDs organize their flash array into a hierarchy of channels, dies, planes, blocks and pages [1, 17]. As shown in Figure 2, each SSD has multiple channels, each channel has multiple dies, and each die has multiple planes. The number of channels, dies and planes varies by vendor and generation. Typically, there are 2 - 4 planes per die, 4 - 8 dies per channel, and 8 - 32 channels per drive. Each plane is composed of thousands of blocks (typically 4-9MB) and each block contains 128-256 pages.

This architecture plays an important role in defining isolation boundaries. Channels, which share only the resources common to the whole SSD, provide the strongest isolation. Dies execute their commands with complete independence, but they must share a bus with other dies on the same channel. Planes' isolation is limited because the die contains only one address buffer. The controller may isolate data to different planes, but operations on these data must happen at different times or to the same address on each plane in a die [32].

In current drives, none of this flexibility is exposed to the host. Drives instead optimize for a single IO pattern: extremely large or sequential IO. The FTL logically groups all planes into one large unit, creating "super-pages" and "super-blocks" are hundreds of times larger than their base unit. For example, a drive with 4MB blocks and 256 planes has a 1GB super-block.

Striping increases the throughput of large, sequential IOs, but introduces the negative side effect of interference between multiple tenants sharing the drive. As all data is striped, every tenant's reads, writes and erases can potentially conflict with every other tenant's operations.

Previous work had proposed novel techniques to help tenants place their data such that underlying flash pages are allocated from separate blocks. This helps improve performance by reducing the write amplification factor (WAF) [34]. Lack of block sharing has the desirable side effect of clumping garbage into fewer blocks, leading to more efficient garbage collection (GC), thereby reducing tail latency of SSDs [25, 42, 43, 73].

However, significant interference still exists between tenants because when data is striped, every tenant uses every channel, die and plane for storing data and the storage operations of one tenant can delay other tenants. Software isolation techniques [57, 67, 68] split the the SSD's resources fairly. However, they cannot maximally utilize the flash parallelism when resource contention exists at a layer below because of the forced sharing of independent resources such as channels, dies and planes.

New SSD designs, such as open-channel SSDs that explicitly expose channels, dies and planes to the operating system [44, 38, 49], can help tenants who share an SSD avoid some of these pitfalls by using dedicated channels. However, the wear imbalance problem between channels that ensues from different tenants writing at different rates remains unsolved. We propose a holistic approach to solve this problem by exposing flash channels and dies as virtual SSDs, while the system underneath wear-levels within each vSSD and balances the wear across channels and dies at coarse time granularities.

FlashBlox is concerned only with sharing of the resources within a single NVMe SSD. Fair sharing mechanisms that split PCIe bus bandwidth across multiple NVMe devices, network interface cards, graphic processing units and other PCIe devices is beyond the scope of this work.

3 Design and Implementation

Figure 3 shows the FlashBlox architecture. At a high level, FlashBlox consists of the following three components: (1) A resource manager that allows tenants to al-

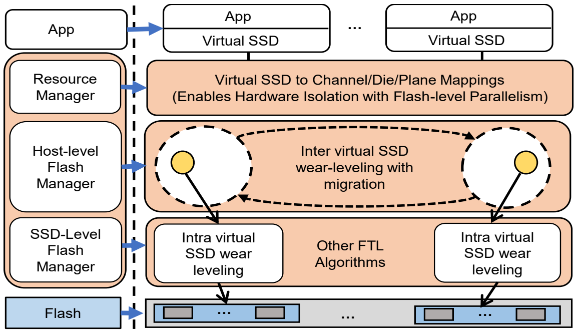

App
App
Virtual SSD
...
App
Virtual SSD
Resource Manager
Virtual SSD to Channel/Die/Plane Mappings (Enables Hardware Isolation with Flash-level Parallelism)
Host-level Flash Manager
Inter virtual SSD wear-leveling with migration
SSD-Level Flash Manager
Intra virtual SSD wear leveling
Other FTL Algorithms
Intra virtual SSD wear leveling
Flash
...

Figure 3: The system architecture of FlashBlox.

Table 1: Virtual SSD types supported in FlashBlox.

Virtual SSD Type	Isolation Level	Alloc. Granularity
Channel Isolated vSSD (§ 3.1)	High	Channel
Die Isolated vSSD (§ 3.2)	Medium	Die
Software Isolated vSSD (§ 3.3)	Low	Plane/Block
Unisolated vSSD (§ 3.3)	None	Block/Page

locate and deallocate virtual SSDs (vSSD); (2) A host-level flash manager that implements inter-vSSD wear-levelling by balancing wear across channels and dies at coarse time granularities; (3) An SSD-level flash manager that implements intra-vSSD wear-levelling and other FTL functionalities.

One of the key new abstractions provided by FlashBlox is that of a virtual SSD (vSSD) which can reduce tail latency. It uses dedicated flash hardware resources such as channels and dies that can be operated independently from each other. The following API creates a vSSD:

```
vssd_t AllocVirtualSSD(int isolationLevel,
          int tputLevel, size_t capacity);
```

Instead of asking tenants to specify absolute numbers, FlashBlox enables them to create different sizes and types of vSSDs with different levels of isolation and throughput (see Table 1). These parameters are compatible with the performance and economic cost levels such as the ones [3, 6] advertised in DaaS systems to ease usage and management. Tenants can scale up capacity by creating multiple vSSDs of supported sizes just as it is done in DaaS systems today. A vSSD is deallocated with `void DeallocVirtualSSD(vssd_t vSSD)`.

Channels, dies and planes are used for providing different levels of performance isolation. This brings significant performance benefits to multi-tenant scenarios (details discussed in § 4.2) because they can be operated independently from each other.

Higher levels of isolation have larger resource allocation granularities as channels are larger than dies. Therefore, channel-granular allocations can have higher internal fragmentation compared to die-granular allocations. However, this is less of a concern for FlashBlox's design for several reasons. First, a typical data center server can house eight NVMe SSDs [46]. Therefore, the maximum number of channel-isolated and die-isolated vSSDs we can support is 128 and 1024 respectively using 16-channel SSDs. Further, SSDs with 32 channels are on the horizon which can double the number of vSSDs which should be sufficient based on our conversations with the service providers at Microsoft.

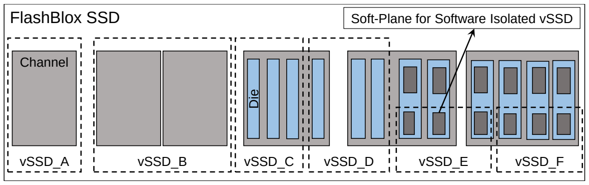


Figure 4: A FlashBlox SSD: vSSD_A and B use one and two channels respectively. vSSD_C and D use three dies each. vSSD_E, and F use three soft-planes each.

Second, the differentiated storage offerings of DaaS systems [3, 6, 10] allow tenants to choose from a certain fixed number of performance and capacity classes. This allows the cloud provider to reduce complexity. In such applications, the flexibility of dynamically changing capacity and IOPS is obtained by changing the number of partitions dedicated to the application. FlashBlox's design of bulk channel/die allocations aligns well with such a model. Third, the differentiated isolation levels match with the existing cost model for cloud storage platforms, in which better services are subject to increased pricing. This is a natural fit for FlashBlox where channels are more expensive and performant than dies.

In DaaS systems, capacity is simply scaled up by creating new partitions. For instance in Amazon RDS and Azure DocumentDB, applications scale capacity by increasing the number of partitions. Each partition is offered as a fixed unit containing a certain amount of storage and IOPS (or application-relevant operations per second). We designed FlashBlox for meeting the demands of DaaS applications. Finally, hardware-isolated vSSDs can coexist with software-isolated ones. For instance, a few channels of each SSD can be used for providing traditional software-isolated SSDs whereby the cloud provider further increases the number of differentiated performance and isolation levels.

Beyond providing different levels of hardware isolation, FlashBlox has to overcome the unbalanced wear-leveling challenge to prolong the SSD lifetime. We describe the design of each vSSD type and its corresponding wear-leveling mechanism respectively as follows.

3.1 Channel Isolated Virtual SSDs

A vSSD with high isolation receives its own dedicated set of channels. For instance, the resource manager of an SSD with 16 channels can host up to 16 channel-isolated vSSDs, each containing one or more channels inaccessible to any other vSSD. Figure 4 illustrates vSSD A and B that span one and two channels respectively.

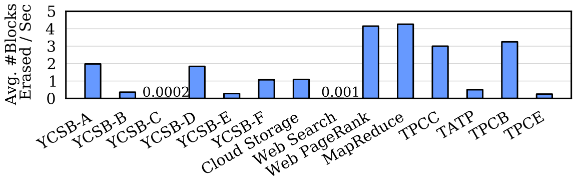


Figure 5: The average rate at which flash blocks are erased for various workloads, including NoSQL, SQL and batch processing workloads.

3.1.1 Channel Allocation

The throughput level and target capacity determine the number of channels allocated to a channel isolated vSSD. To this end, FlashBlox allows the data center/DaaS administrator to implement the `size_t tputToChannel(int tputLevel)` function that maps between throughput levels and required number of channels. The number of channels allocated to the vSSD is, therefore, the maximum of `tputToChannel(tputLevel)` and ⌈`capacity / capacityPerChannel`⌉.

Within a vSSD, the system stripes data across its allocated channels similar to traditional SSDs. This maximizes the peak throughput by operating on the channels in parallel. Thus, the size of the super-block of vSSD_A in Figure 4 is half that of vSSD_B. Pages within the super-block are also striped across the channels similar to existing physical SSDs.

The hardware-level isolation present between the channels by virtue of hardware parallelism allows the read, program and erase operations on one vSSD to largely be unaffected by the operations on other vSSDs. Such an isolation enables latency sensitive applications to significantly reduce their tail latencies.

Compared to an SSD that stripes data from all applications across all channels, a vSSD (over fewer channels) delivers a portion of the SSD's all-channel bandwidth. Customers of DaaS systems are typically given and charged for a fixed bandwidth/IOPS level, and software rate-limiters actively keep their consumption in check. Thus, there is no loss of opportunity for not providing the peak-bandwidth capabilities for every vSSD.

3.1.2 Unbalanced Wear-Leveling Challenge

A significant side effect of channel isolation is the risk of uneven aging of the channels in the SSD as different vSSDs may be written at different rates. Figure 5 shows how various storage workloads erase blocks at different rates indicating that channels pinned naively to vSSDs will age at different rates if left unchecked.

Such uneven aging may exhaust a channel's life long before other channels fail. Premature death of even a single channel would render significant capacity losses ($>$ 6% in our SSD). Furthermore, premature death of a single channel leads to an opportunity loss of never being able to create a vSSD that spans all the 16 channels for the rest of the server's lifetime. Such an imbalance in capability of servers represents lost opportunity costs given that other components in the server such as CPU, network and memory do not prematurely lose capabilities. Furthermore, unpredictable changes in capabilities also complicate the job of load-balancers which typically assume uniform or predictably non-uniform (by design) capabilities. Therefore, it is necessary to ensure that all the channels are aging at the same rate.

3.1.3 Inter-Channel Wear-Leveling

To ensure uniform aging of all channels, FlashBlox uses a simple yet effective wear-leveling scheme:

Periodically, the channel that has incurred the maximum wear thus far is swapped with the channel that has the minimum rate of wear.

A channel's wear rate is the average rate at which it erased blocks since the last time the channel was swapped. This prevents the most-aged channels from seeing high wear rates, thus intuitively extending their lifetime to match that of the other channels in the system.

Our experiments with workload traces from Microsoft's data center workloads show that such an approach works well in practice. We can ensure near-perfect wear-leveling with this mechanism and a swap frequency of once every few weeks. Furthermore, the impact on tail-latency remains low during the 15-minute migration period (see § 4.3.1). We analytically derive the minimum necessary frequency in § 3.1.4 and present the design of the migration mechanism in § 3.1.5.

3.1.4 Swap Frequency Analysis

Let σ_i denote the wear (total erase count of all the blocks till date) of the i^{th} channel. $\xi = \sigma_{max}/\sigma_{avg}$ denotes the wear imbalance[1] which must not exceed 1 + δ; where $\sigma_{max} = Max(\sigma_1, ..., \sigma_N)$, $\sigma_{avg} = Avg(\sigma_1, ..., \sigma_N)$, N is the total number of channels, and δ measures the imbalance.

When the device is new, it is obviously not possible to ensure that $\xi \leq 1 + \delta$ without aggressively swapping channels. On the other hand, it must be brought within bounds early in the lifetime of the server (L = 150–250 weeks typical) such that all the channels are available for as much of the server's lifetime as possible.

SSDs are provisioned with a target erase workload and we analyze for the same – let's say M erases per week. We mathematically study the wear-imbalance vs. frequency of migration (f) tradeoff and show that manage-

[1] The ratio of maximum to average is an effective way to quantify imbalance [45]. This is especially true in our case, as the lifetime of the new SSD is determined by the maximum wear of a single channel, whereas the lifetime of ideal wear-leveling is determined by the average wear of all the channels. The ratio of maximum to average thus represents the loss of lifetime due to imperfect wear leveling.

able values of f can provide acceptable wear imbalance where ξ comes below $1+\delta$ after αL weeks, where α is between 0 and 1.

The worst-case workload for FlashBlox is when all the writes go to a single channel.[2] The assumption that a single channel's bandwidth can handle the entire provisioned bandwidth is valid for modern SSDs: most SSDs are provisioned with 3,000-10,000 erases per cell to last 150–250 weeks. The provisioned erase rate for a 1TB SSD is therefore M=21–116 MBPS, which is lower than a channel's erase bandwidth (typically 64–128MBPS).

For an SSD with N channels, the wear imbalance of ideal wear-leveling is ξ = 1, while the worst case workload for FlashBlox gives a ξ = N: $\sigma_{max}/\sigma_{avg} = M*time/(M*time/N) = N$ before any swaps. A simple swap strategy of cycling the write workload through the N channels (write workload spends $1/f$ weeks per channel) is analyzed. Let's assume that after K rounds of cycling through all the channels, $KN/f \geq \alpha L$ holds true – that is αL weeks have elapsed and ξ has become less than $1+\delta$ and continues to remain there. At that very instant ξ equals 1. Therefore, σ_{max} = MK and σ_{avg} = MK, then after the next swap, σ_{max} = $MK+M$ and σ_{avg} = $MK+M/N$. In order to guarantee that the imbalance is always limited, we need:

$$\xi = \sigma_{max}/\sigma_{avg} = (MK+M)/(MK+M/N) \leq (1 + \delta)$$

This implies $K \geq (N-1-\delta)/(N\delta)$ which is upper bounded by $1/\delta$. Therefore, to guarantee that $\xi \leq (1+\delta)$, it is enough to swap $NK = N/\delta$ times in the first αL weeks. This implies that, over a period of five years, if α were 0.9 then a swap must be performed once every 12 days ($= 1/f$) for a $\delta = 0.1$ ($N = 16$). Table 2 shows how the frequency of swaps increases with the number of channels (shown as decreasing time period). This also implies that $\frac{2}{16}^{th}$ of the SSD is erased to perform the swap once every 12 days, which is negligible compared to the 3,000–10,000 cycles that typical SSDs have. However, for realistic workloads that do not have such a skewed write pattern with a constant bandwidth, swaps must be adaptively performed according to workload patterns (see Table 5) to reduce the number of swaps needed while maintaining balanced wear.

3.1.5 Adaptive Migration Mechanism

We assume a constant write rate of M for analysis purposes, but in reality writes are bursty. High write rates must trigger frequent swaps while swapping may not be needed as often during periods of low write rates. To achieve this, FlashBlox maintains a counter per channel to represent the amount of space erased (MB) in each channel since the last swap. Once one of the counters goes beyond a certain threshold γ, a swap is performed, and the counters are cleared. γ is set to the amount of space erased if the channel experiences the worst-case write workload between two swaps (i.e., M/f).

Table 2: The frequency of swaps increases as the number of channels increase to maintain balanced wear – swap periods shown below for the SSD to last five years.

Number of Channels	8	16	32	64
Swap Period (days)	26	12	6	3

The rationale behind this mechanism is that the channels must always be positioned in a manner to be able to catch up in the worst-case. FlashBlox then swaps the channels with σ_{max} and λ_{min}, where λ_i denotes the wear rate of the i^{th} channel and $\lambda_{min} = Min(\lambda_1, ..., \lambda_N)$.

FlashBlox uses an atomic block-swap mechanism to gradually migrate the candidate channels to their new locations without any application involvement. The mechanism uses an erase-block granular mapping table (described in § 3.4) for each vSSD that is maintained in a consistent and durable manner.

The migration happens in four steps. First, FlashBlox stops and queues all of the in-flight read, program and erase operations associated with the two erase-blocks being swapped. Second, the erase-blocks are read into a memory buffer. Third, the erase-blocks are written to their new locations. Fourth, the stopped operations are then dequeued. Note that only the IO operations for the swapping erase blocks in the vSSD are queued and delayed. The IO requests for other blocks are still issued with higher priority to mitigate the migration overhead.

The migrations affect the throughput and latency of the vSSDs involved. However, they are rare (happen less than once in a month for real workloads) and take only 15 minutes to finish (see § 4.3.1).

As a future optimization, we wish to modify the DaaS system to perform the read operations on other replicas to further reduce the impact. For systems that perform reads only on the primary replica, the migration can be staged within a replica-set such that the replica that is currently undergoing a vSSD migration is, if possible, first converted into a backup. Such an optimization would reduce the impact of migrations on the reads in applications that are replicated.

3.2 Die-Isolated Virtual SSDs

For applications which can tolerate some interference (i.e., *medium* isolation) such as the non-premium cloud database offerings (e.g., Amazon's small database instance [3] and Azure's standard database service [62]), FlashBlox provides die-level isolation. The num-

[2] This worst-case is from a non-adversarial point of view. An adversary could change the vSSD write bandwidth at runtime such that no swapping strategy can keep up. But most data center workloads are not adversarial and have predictable write patterns. We leave it to a security watch dog to kill over-active workloads that are not on a whitelist.

ber of dies in such a vSSD is the maximum of `tputToDie(tputLevel)` (defined by the administrator) and $\lceil$`capacity / capacityPerDie`$\rceil$. Their super-blocks and pages stripe across all the dies within the vSSD to maximize throughput. Figure 4 illustrates vSSD_C, and D containing three dies each (vSSD_D has dies from different channels). These vSSDs, however, have weaker isolation guarantees since dies within a channel must share a bus.

The wear-leveling mechanism has to track wear at the die level as medium-level isolated vSSDs are pinned to dies. Thus, we split the wear-leveling mechanism in FlashBlox into two sub-mechanisms: channel level and die level. The job of the channel-level wear-balancing mechanism is to ensure that all the channels are aging at roughly the same rate (see § 3.1). The job of the die-level wear-balancing mechanism is to ensure that all the dies within a channel are aging roughly at the same rate.

As shown in § 3.1.4, an N channel SSD has to swap at least N/δ times to guarantee $\xi \leq (1+\delta)$ within a target time period. This analysis also holds true for dies within a channel. For the SSDs today, in which each channel has 4 dies, FlashBlox has to swap dies in each channel 40 times in the worst case during the course of the SSD's lifetime or once every month.

As an optimization, we leverage the channel-level migration to opportunistically achieve the goal of die-level wear-leveling, based on the fact that dies have to migrate along with the channel-level migration. During each channel-level migration, the dies within the migrated channels with the largest wear is swapped with the dies that have the lowest write rate in the respective channels. Experiments with real workloads show that such a simple optimization can effectively provide satisfactory lifetime for SSDs (see § 4.3.2).

3.3 Software Isolated Virtual SSDs

For applications that have even lower requirements of isolation like Azure's basic database service [62], the natural possibility of using plane level isolation arises. However, planes within a die do not provide the same level of flexibility as channels and dies with respect to operating them independently from each other: Each die allows operating either one plane at a time or all the planes at the same address offset. Therefore, we use an approach where all the planes are operated simultaneously but their bandwidth/IOPS is split using software.

Each die is split into four regions of equal size called soft-planes by default, the size of each soft-plane is 4 GB in FlashBlox (other configurations are also supported). Planes are physical constructs inside a die. Soft-planes however are simply obtained by striping data across all the planes in the die. Further, each soft-plane in a die obtains an equal share of the total number of blocks within a die. They also receive *fair* share of bandwidth of the die. The rationale behind this is to make it easier for data center/PaaS administrator to map the throughput levels required from tenants to quantified numbers of soft-planes.

vSSDs created using soft-planes are otherwise indistinguishable from traditional virtual SSDs where software rate limiters are used to split an SSD across multiple tenants. Similar to such settings, we use the state-of-the-art token bucket rate-limiter [13, 67, 78] which has been widely used for Linux containers and Docker [12] to improve isolation and utilization at the same time. Our actual implementation is similar to the weighted fair-share mechanisms in prior work [64]. In addition, separate queues are used for enqueuing requests to each die.

The number of soft-planes used for creating these vSSDs is determined similarly to the previous cases: as the maximum of `tputToSoftPlane(tputLevel)` and $\lceil$`capacity / capacityPerSoftPlane`$\rceil$. Figure 4 illustrates vSSDs E and F that contain three soft-planes each. The super-block used by such vSSDs is simply striped across all the soft-planes used by the vSSD. We use such vSSDs as the baseline for our comparison of channel and die isolated vSSDs.

The software mechanism allows the flash blocks of each vSSD to be trimmed in isolation, which can reduce the GC interference. However, it cannot address the situation where erase operations on one soft-planes occasionally block all the operations of other soft-planes on the shared die. Thus, such vSSDs can only provide software isolation which is lower than die-level isolation.

Besides these isolated vSSDs, FlashBlox also supports an **unisolated vSSD** model which is similar to software isolated vSSD, but a fair sharing mechanism is not used to isolate such vSSDs from each other. To guarantee the fairness between vSSDs in today's cloud platforms, software isolated vSSDs are enabled by default in FlashBlox to meet *low* isolation requirements.

For both software isolated and unisolated vSSDs, their wear-balancing strategy is kept the same rather than swapping soft-planes. The rationale for this is that isolation between soft-planes of a die is provided using software and not by pinning vSSDs to physical flash planes. Therefore, a more traditional wear-leveling mechanism of simply rotating blocks between soft-planes of a die is sufficient to ensure that the soft-planes within a die are all aging roughly at the same rate. We describe this mechanism in more detail in the next section.

3.4 Intra Channel/Die Wear-Leveling

The goals of intra die wear-leveling are to ensure that the blocks in each die are aging at the same rate while enabling applications to access data efficiently by avoiding the pitfalls of multiple indirection layers and redundant functionalities across these layers [27, 35, 54, 77].

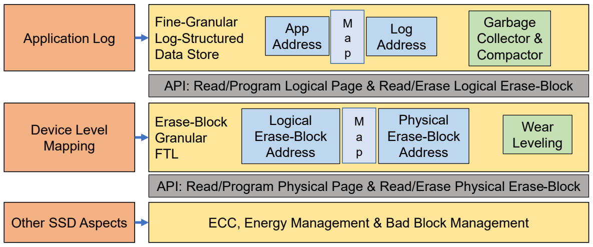

Application Log
Fine-Granular Log-Structured Data Store
App Address
Map
Log Address
Garbage Collector & Compactor
API: Read/Program Logical Page & Read/Erase Logical Erase-Block
Device Level Mapping
Erase-Block Granular FTL
Logical Erase-Block Address
Map
Physical Erase-Block Address
Wear Leveling
API: Read/Program Physical Page & Read/Erase Physical Erase-Block
Other SSD Aspects
ECC, Energy Management & Bad Block Management

Figure 6: In FlashBlox, applications manage a fine-granular log-structured data store and align compaction units to erase-blocks. A device level indirection layer is used to ensure all erase-blocks are aging at the same rate.

With both die-level (see § 3.3) and intra-die wear leveling mechanisms, FlashBlox inevitably achieves the goal of intra-channel wear-leveling as well: all the dies in each channel and all the blocks in each die age uniformly.

The intra-die wear-leveling in FlashBlox is illustrated in Figure 6. We leverage flash-friendly application or file system logic to perform GC and compaction, and simplify the device level mapping. We also leverage the drive's capabilities to manage bad blocks without having to burden applications with error correction, detection and scrubbing. We base our design for intra-die wear-levelling on existing open SSDs [27, 38, 49]. We describe our specific design for completeness.

3.4.1 Application/Filesystem Level Log

The API of FlashBlox, as shown in Table 3, is designed with log-structured systems in mind. The only restriction it imposes is that the application or the file system perform the log-compaction at a granularity that is the same as the underlying vSSD's erase granularity.

When a FlashBlox based log-structured application or a filesystem needs to clean an erase-block that contains a live object (say *O*) then, (1) It first allocates a new block via `AllocBlock`; (2) It reads object *O* via `ReadData`; (3) It writes object *O* in to the new block via `WriteData`; (4) It modifies its index to reflect the new location of object *O*; (5) It frees the old block via `FreeBlock`. Note that the newly allocated block still has many pages that can be written to, which can be used as the head of the log for writing live data from other cleaned blocks or for writing new data.

FlashBlox does not assume that the log-structured system frees all the allocated erase-blocks at the same rate. Such a restriction would force the system to implement a sequential log cleaner/compactor as opposed to techniques that give weight to other aspects such as garbage collection efficiency [37, 38]. Instead, FlashBlox ensures uniform wear of erase-blocks at a lower level.

3.4.2 Device-Level Mapping

The job of the lower layers is to ensure: (1) that all erase-blocks within a die are being erased at roughly the same rate and (2) that erase-blocks that have imminent failures have their data migrated to a different erase-block and the erase-block be permanently hidden from applications; both without requiring application changes.

Table 3: FlashBlox API

`vssd_t AllocVirtualSSD( int isolationLevel, int tputLevel, size_t capacity )` /*Creates a virtual SSD*/
`void DeallocVirtualSSD( vssd_t vSSD )` /*Deletes a virtual SSD*/
`size_t GetBlockSize( vssd_t vSSD )` /*Erase-block size of vSSD: depends on the number of channels/dies used*/
`int ReadData( vssd_t vSSD, void* buf, off_t offset, size_t size )` /*Reads data; contiguous data is read faster with die-parallel reads*/
`block_t AllocBlock( vssd_t vSSD )` /*Allocates a new block; it can be written to only once and sequentially*/
`int WriteData( vssd_t vSSD, block_t logical_block_id, void* buf, size_t size )` /*Writes page aligned data to a previously allocated (erased) block; contiguous data is written faster with die-parallel writes*/
`void FreeBlock( vssd_t vSSD, block_t logical_block_id )` /*Frees a previously allocated block*/

With device-level mapping, the physical erase-blocks' addresses are not exposed to applications – only logical erase-block addresses are exposed to upper software. That is, the device exposes each die as an individual SSD that uses a block-granular FTL, while application-level log in FlashBlox ensures that upper layers only issue block-level allocation and deallocation calls. The indirection overhead is small since they are maintained at erase-block granularity (requiring 8MB per TB of SSD).

Unlike tradtional SSDs, in FlashBlox, tenants cannot share pre-erased blocks. While this has the advantage that the tenants control their own write-amplification factors, write and GC performance, the disadvantage is that bursty writes within a tenant cannot opportunistically use pre-erased blocks from the entire device.

In FlashBlox, each die is given its own private block-granular mapping table, and a IO queue with a depth of 256 by default (it is configurable) to support basic storage operations and software rate limiter for software isolated vSSDs. The out-of-band metadata (16 bytes used) in each block is used to note the logical address of the physical erase-block; this enables atomic, consistent and durable operations. The logical address is a unique and global 8 bytes number consisting of die ID and block ID within the die. The other 8 bytes of the metadata are used for a 2 bytes erase counter and a 6 byte erase timestamp. FlashBlox caches the mapping table and all other out-of-band metadata in the host memory. Upon system crashes, FlashBlox leverages the reverse mappings and timestamps in out-of-metadata to recover the mapping table [24, 80]. More specifically, we use the implementation from our prior work [27].

The device-level mapping layer can be implemented either in the host or in the drive's firmware itself [49] if the device's controller has under-utilized resources; we implement it in the host. Error detection, correction and masking, and other low-level flash management systems remain unmodified in FlashBlox.

Both the application/filesystem level log and the device-level mapping need to over provision, but for different reasons. The log needs to over-provision for the sake of garbage collection efficiency. Here, we rely on the existing logic within log-structured, flash-aware applications and file systems to perform their own over-provisioning appropriate for their workloads. The device-level mapping needs its own over-provisioning for the sake of retiring error-prone erase-blocks. In our implementation, we set this to 1% based on the error rate analysis from our prior work [29].

3.5 Implementation Details

Prototype SSD. We implement FlashBlox using a CNEX SSD [18] which is an open-channel SSD [44] containing 1 TB Toshiba A19 flash memory and an open controller that allows physical resource access from the host. It has 16 channels, each channel has 4 dies, each die has 4 planes, each plane has 1024 blocks, each block has 256 pages with 16 KB page size. This hardware provides basic I/O control commands to issue read, write and erase operations against flash memory. We use a modified version of the CNEX firmware/driver stack that allows us to independently queue requests to each die. FlashBlox is implemented using the C programming language in 11,219 lines of code (LoC) layered on top of the CNEX stack.

Prototype Application and Filesystem. We were able to modify LevelDB key-value store and the Shore-MT database engine to use FlashBlox using only 38 and 22 LoC modifications respectively. These modifications are needed to use the APIs in Table 3. Additionally, we implemented a user-space log-structured file system (vLFS) with 1,809 LoC (only 26 LoC are from FlashBlox API) based on FUSE for applications which cannot be modified.

Resource Allocation. For each call to create a vSSD, the resource manager performs a linear search of all the available channels, dies and soft-planes to satisfy the requirements. A set of free lists of them are maintained for this purpose. During deallocation, the resource manager takes the freed channels, dies and soft-planes and coalesces them when possible. For instance, if all the four dies of a channel become free then the resource manager coalesces the dies into a channel and adds the channel to the free channel set. In the future, we wish to explore admission control and other resource allocation strategies.

4 Evaluation

Our evaluation demonstrates that: (1) FlashBlox has overheads (WAF and CPU) comparable to state-of-the-art FTLs (§ 4.1); (2) Different levels of hardware isolation can be achieved by utilizing flash parallelism, and they perform better than software isolation (§ 4.2.1); (3) Hardware isolation enables latency-sensitive applications such as web search to effectively share an SSD with bandwidth-intensive workloads like MapReduce jobs (§ 4.2.2); (4) The impact of wear-leveling migrations on data center applications' performance is low (§ 4.3.1) and (5) FlashBlox's wear-leveling is near to ideal (§ 4.3.2).

Table 4: Application workloads used for evaluation.

	Workload	I/O Pattern
Key-Value Store	YCSB-A	50% read, 50% update
	YCSB-B	95% read, 5% update
	YCSB-C	100% read
	YCSB-D	95% read, 5% insert
	YCSB-E	95% scan, 5% insert
	YCSB-F	50% read, 50% read-modify-write
Data Center	Cloud Storage	26.2% read, 73.8% write
	Web Search	83.0% read, 17.0% write
	Web PageRank	17.7% read, 82.2% write
	MapReduce	52.9% read, 47.1% write
Databases	TPC-C	mix (65.5% read, 34.5% write)
	TATP	mix (81.2% read, 18.8% write)
	TPC-B	account update (100% write)
	TPC-E	mix (90.7% read, 9.3% write)

Experimental Setup: We used FIO benchmarks [20] and 14 different workloads for the evaluation (Table 4): six NoSQL workloads from the Yahoo Cloud Serving Benchmarks (YCSB) [19], four database workloads: TPC-C [70], TATP [65], TPC-B [69] and TPC-E [71], and four storage workload traces collected from Microsoft's data centers.

YCSB is a framework for evaluating the performance of NoSQL stores. All of the six core workloads consisting of A, B, C, D, E and F are used for the evaluation. LevelDB [39] is modified to run using the vSSDs from FlashBlox with various isolation levels. The open-source SQL database Shore-MT [55] is modified to work over the vSSDs of FlashBlox. The table size of the four database workloads TPC-C, TATP, TPC-B and TPC-E range from 9 - 25 GB each. A wear-imbalance factor limit of 1.1 is used for all our experiments to capture realistic swapping frequencies. The number of dies, channels and planes used for each experiment is specified separately for each experiment.

Storage intensive and latency sensitive applications from Microsoft's data centers are instrumented to collect traces for cloud storage, web search, PageRank and MapReduce workloads. These applications are the first-party customers of Microsoft's storage IaaS system.

4.1 Microbenchmarks

We benchmark two vSSDs that each run an FIO benchmark to evaluate FlashBlox's WAF. Compared to the unmodified CNEX SSD's page-level FTL, FlashBlox deliv-

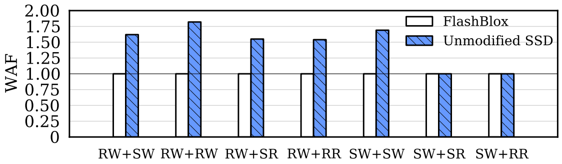


Figure 7: WAF comparison between FlashBlox and a traditional SSD. RW/RR: random write/read; SW/SR: sequential write/read.

ers lower WAFs as shown in Figure 7 because of the fact that FlashBlox's vSSDs never share the same physical flash blocks for storing their pages. As shown by previous work [34], this reduces WAF because of absence of false sharing of blocks at the application level. The different types of vSSDs of FlashBlox have similar WAFs because they all use separate blocks, yet they provide different throughput and tail latency levels (shown in Section 4.2) because of higher levels of isolation.

In addition, FlashBlox has up to 6% higher total system CPU usage compared to the unmodified CNEX SSD when running FIO. Despite merging the file system's index with that of the FTL's by using FlashBlox's APIs which reduces latency as shown by existing open-channel work [38, 49], the additional CPU overhead is due to the device-level mapping layer that is accessed in every critical path. As a future optimization for the production SSD, we plan to transparently move the device-level mapping layer into the SSD.

4.2 Isolation Level vs. Tail Latency

In this section, we demonstrate that higher levels of isolation provide lower tail latencies. Multiple instances of application workloads are run on individual vSSDs of different kinds. In each workload, the number of client threads executing transactions is gradually increased until the throughput tapers off. The maximum throughput achieved for the lowest number of threads is then recorded. The average and tail latencies of transactions are recorded for the same number of threads.

4.2.1 Hardware Isolation vs. Software Isolation

In this experiment, the channel and die isolated vSSDs are evaluated against the software isolated vSSDs (with weighted fair sharing of storage bandwidth enabled). We begin with a scenario of two LevelDB instances. They run on two vSSDs in three different configurations, each using a different isolation level: high, medium, and low; they contain one channel, four dies and sixteen soft-planes respectively to ensure that the resources are consistent across experiments. The two instances run a YCSB workload each. The choice of YCSB is made for this experiment to show how removing IO interference can improve the throughput and reduce latency for IO-bottlenecked applications.

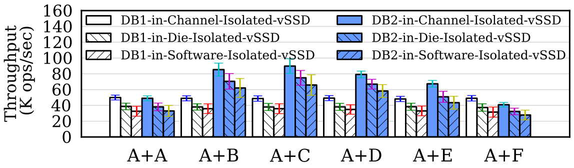
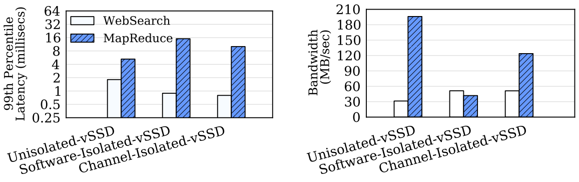


Figure 8: The throughput of LevelDB+YCSB workloads running at various levels of storage isolation.



(a) 99th Percentile Latency

(b) Bandwidth

Figure 11: The performance of colocated Web Search and MapReduce workload traces.

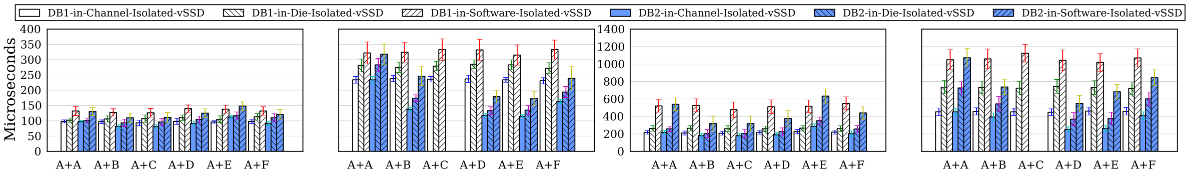
Each LevelDB instance is first populated with 32 GB of data and each key-value pair is 1 KB. The YCSB client threads perform 50 million CRUD (i.e., create, read, update and delete) operations against each LevelDB instance. We pick the size of the database and number of operations such that GC is always triggered. YCSB C is read-only, thus we report results for read operations only.

The total number of dies in each setting is the same. In the channel isolation case, two vSSDs are allocated from two different channels. In the die isolation case, both vSSDs share the channels, but are isolated at the die level within the channel. In the software isolation case, both vSSDs are striped across all the dies in two channels.

Figure 8 shows that on average, channel isolated vSSD provides 1.3x better throughput compared to die isolated vSSD and 1.6x compared to the software isolated vSSD. Similarly, higher levels of isolation lead to lower average latencies as shown in Figure 9 (a) and Figure 9 (b). This is because higher levels of isolation suffer from less interference between read and write operations from other instances. Die isolated vSSDs have to share the bus with each other, thus, their performance is worse than channel isolated vSSDs, which are fully isolated in hardware. Software isolated vSSDs share the same dies with each other, suffering from higher interference.

Tail latency improvements are much more significant. As shown in Figure 9 (c) and Figure 9 (d), channel isolated vSSDs provide up to 1.7x lower tail latency compared to die isolated vSSDs and up to 2.6x lower tail latency compared to vSSDs that stripe data across all the dies akin to software isolated vSSDs whose operations are not fully isolated from each other.

A similar experiment with four LevelDB instances is also performed. Tail latency results are shown in Figure 10 where channel isolated vSSDs provide up to 3.1x lower tail latency compared the software isolated vSSDs.



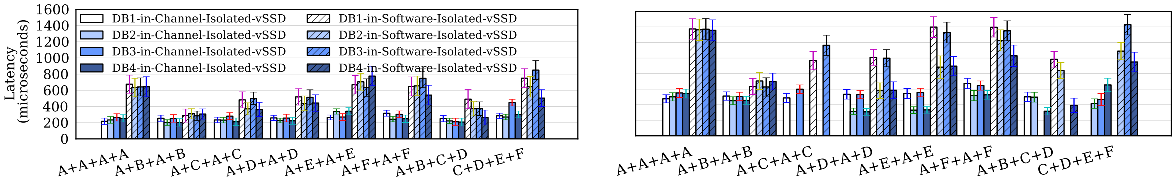
(a) Read (Average)

(b) Update (Average)

(c) Read (99th Percentile)

(d) Update (99th Percentile)

Figure 9: The average and 99th percentile latencies of LevelDB+YCSB workloads running at various levels of storage isolation. Compared to die and software isolated vSSDs, channel isolated vSSD reduces the average latency by 1.2x and 1.4x respectively, and decreases the 99th percentile latency by 1.2 - 1.7x and 1.9 - 2.6x respectively. Note that the update latencies are not applicable for workload C which is read-only.



(a) Read Latency (99th Percentile)

(b) Update Latency (99th Percentile)

Figure 10: The 99th percentile latency of running four LevelDB instances with various levels of storage isolation. The channel isolated vSSD reduces the 99th percentile latency by 1.3 - 2.7x and 1.5 - 3.1x for read and update operation respectively, compared to software isolated vSSD.

4.2.2 Latency vs. Bandwidth Sensitive Tenants

We now evaluate how hardware isolation provides benefits for instances that share the same physical SSD when one is latency sensitive while others are not (for resource efficiency [42]). Channel, software isolated and unisolated vSSDs are used in this experiment. The total number of dies is the same in all three settings and is eight. The workloads from a large cloud provider are used for performing this experiment. Web search is the instance that requires lower tail latencies while MapReduce jobs are not particularly latency sensitive.

Results shown in Figure 11 demonstrate three trends: First, channel isolated vSSDs provide the best compromise between throughput and tail latency: tail latency of the web search workloads decreases by over 2x for a 36% reduction of bandwidth of the MapReduce job when compared to an unisolated vSSD. The fall in throughput of MapReduce is expected because it only has half of the channels of the unisolated case where its large sequential IOs end up consuming the bandwidth unfairly due to the lack of any isolation techniques.

Second, software isolated vSSDs for web search and MapReduce can reduce the tail latency of web search to the same level as the channel-isolated case, but the bandwidth of the MapReduce job decreases by more than 4x when compared to the unisolated vSSD. This is also expected because the work that an SSD can perform is a convex combination of IOPS and bandwidth. Web search takes a significant number of small IOPS when sharing bandwidth fairly with MapReduce and this in-turn reduces the total bandwidth available for MapReduce.

4.3 Wear-Leveling Efficacy and Overhead

Wear-leveling in FlashBlox is supported in two different layers. One layer ensures that all the dies in the system are aging at the same rate overall with channel migrations, while the other layer ensures that blocks within a given die are aging at the same rate overall. Its overhead and efficacy are evaluated in this section.

4.3.1 Migration Overhead

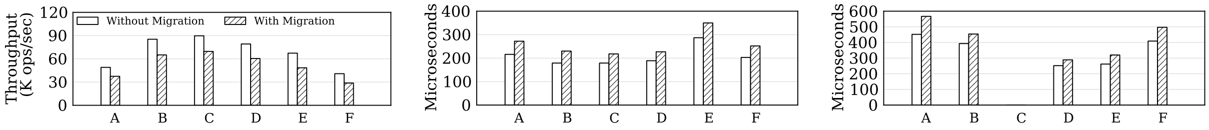
We first evaluate the overhead of the migration mechanism. We migrate one channel and measure the change in throughput and 99th percentile latency on a variety of YCSB workloads that are running on the channel.

The throughput of LevelDB running on that channel drops by no more than 33.8% while the tail latencies of reads and updates increase by up to 22.1% (Figure 12). For simplicity, we show results for migrating 1 GB of the 64GB channel. We use a single thread and the data moves at a rate of 78.9MBPS. Moving all of the 64 GB of data would take close to 15 minutes.

The impact of migration on web search and MapReduce workloads is shown in Figure 13. During migration, the bandwidth of the MapReduce job decreases by 36.7%, the tail latencies of reads and writes of the web search increase by 34.2%. These performance slowdowns bring channel-isolation numbers on par with the software isolation. This implies that a 36.7% drop for 15 minutes when amortized over our recommended swap rate represents a 0.04% overall drop.

4.3.2 Migration Frequency Analysis

To evaluate the wear-leveling efficacy, we built a simulator and used it to understand how the device ages for var-

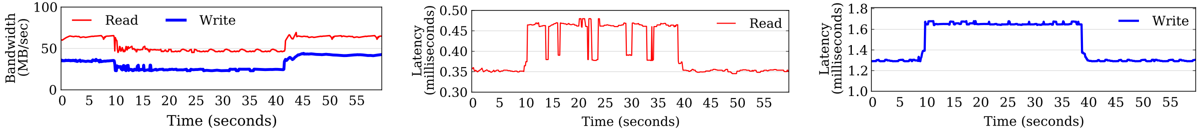


(a) Throughput

(b) Read (99th Percentile)

(c) Update (99th Percentile)

Figure 12: The impact of a channel migration on workloads: LevelDB's throughput falls by 33.8%, its 99th percentile read and update latencies increase by 22.1% and 18.7% respectively.

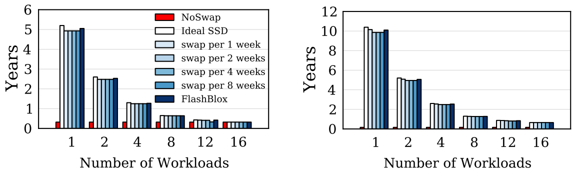


(a) Bandwidth of MapReduce

(b) Read Latency of Web Search

(c) Write Latency of Web Search

Figure 13: The overhead of migrating 1GB of data as MapReduce and web search are running on the channels involved: MapReduce's bandwidth falls by up to 36.7% while web search's latency increases by up to 34.2%.



(a) Channel Killer

(b) Die Killer

Figure 14: SSD lifetime of running adversarial write workloads that stress a single channel or a die.

ious workloads. For workload traces that are not from a log-structured application, we first execute the workload on the log-structured file system vLFS built using FlashBlox and trace FlashBlox API calls. We measure the block consumption rate of these traces to evaluate the efficacy of wear-leveling. For the CNEX SSD, $\gamma = M/f =$ 24 TB (discussed in § 3.1.5). The supported number of program erase (PE) cycles is 10 K in our drive. Our absolute lifetimes scale linearly for other SSDs and factor improvements remain the same regardless of the number of supported PE cycles.

Worst-case workloads. To evaluate the possible worst cases for SSDs, we run the most write-intensive workloads against a few channels (channel killer) and dies (die killer). We gradually increase the number of such workloads to stress the SSD. Each workload is pinned to exactly one channel or one die while keeping other channels or dies for read-only operations.

Figure 14 shows the SSD's lifetime for a variety of wear-leveling schemes. Without wear-leveling (NoSwap in Figure 14), the SSD dies after less than 4 months, while FlashBlox can always guarantee 95% of the ideal lifetime within migration frequency of once per ≤ 4 weeks for both channel and die killer workloads. The adaptive wear-leveling scheme in FlashBlox automatically migrates a channel by adjusting to write-rates.

Mixed workloads. In real-world scenarios, a mix of various kinds of workloads would run on the same SSD. We use all the 14 workloads (Table 4) simultaneously in the experiment, and measure FlashBlox's wear leveling. Fourteen channel isolated vSSDs are created for running these workloads and migrations. Figure 5 shows how the erase rates of these applications vary.

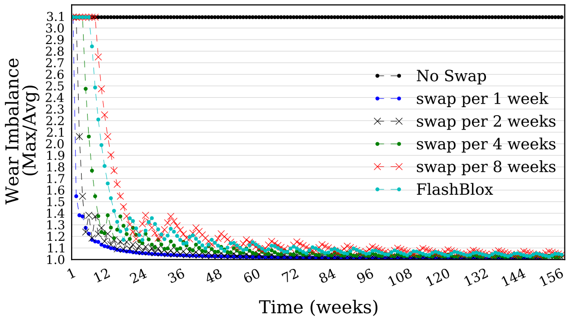


Figure 15: Wear imbalance of FlashBlox with different wear-leveling schemes. The ideal wear imbalance is 1.0.

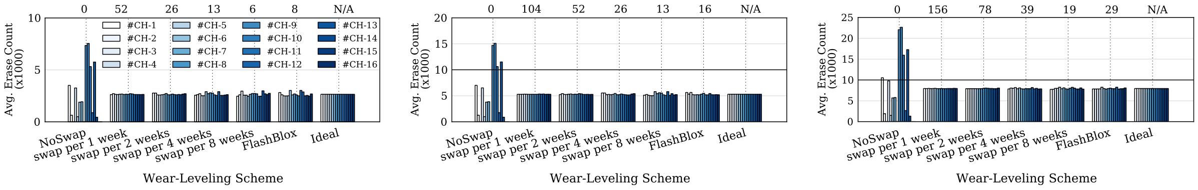
Table 5: Monte Carlo simulation (10K runs) of SSD lifetime with randomly sampled workloads on the channels.

#vSSD	NoSwap Lifetime (Years)		Ideal vs. FlashBlox Lifetime (Years)		Wear Imbalance	Swap Once in Days (Avg)
	99th	50th	99th	50th		
4	1.2	1.6	6.2/6.1	13.8/13.5	1.02	94
8	1.2	1.3	3.7/3.6	6.7/6.6	1.02	22
16	1.2	1.2	2.1/2.1	3.4/3.3	1.01	19

For the scheme without any migrations, the wear imbalance is 3.1, and the SSD dies after 1.2 years. Also, results show that blocks are more or less evenly aged for a migration frequency as high as once in four weeks, as shown in Figure 15. This indicates that for realistic scenarios, where write traffic is more evenly matched, significantly fewer swaps could be tolerated.

Figure 16 shows the absolute erase counts of the channels (including the erases needed for migrations and GC). Compared to the ideal wear-leveling, the absolute erase counts are almost the same with the migration frequency of a week.

To further evaluate FlashBlox's wear-leveling efficacy, we run a Monte Carlo simulation (10K runs) of the SSD lifetime. We create various number of vSSDs and assign



(a) After 1 year

(b) After 2 years

(c) After 3 years

Figure 16: Erase counts over three years for workloads in Table 4. The erase count per block in each channel of FlashBlox is close to that of the ideal SSD. The numbers on the top shows the cumulative migration count.

them uniformly at random to one of the fourteen workloads. The SSD is then simulated to end-of-life.

We report the 99th and 50th percentile lifetime of ideal SSD, SSD without swapping (NoSwap) and FlashBlox in Table 5. For the case of running 16 instances, 99% of the ideal SSDs last 2.1 years, and half of them can work for 3.4 years. With adaptive wear-leveling scheme, FlashBlox's lifetime is close to ideal and its wear imbalance is close to the ideal case. In real world, where not all applications are adversarial (channel/die-killer workloads), the swap frequency automatically increases.

5 Related Work

Open Architecture SSDs. Recent research has proposed exposing flash parallelism directly to the host [38, 49, 61, 76]. This is immensely helpful for applications where each unit of flash parallelism receives more or less similar write workloads. However, this is often not the case in multi-tenant cloud platform where workloads with a variety of write-rates co-exist on the same SSD. FlashBlox takes a holistic approach to solve this problem, it not only provides hardware isolation but also ensures all the units of parallelism are aging uniformly.

SSD-level Optimizations. Recent work has successfully improved SSDs' performance by enhancing how FTLs leverage the flash parallelism [17, 32]. We extend this line of research for performance isolation for applications in a multi-tenant setting. FlashBlox uses dedicated channels and dies for each application to improve isolation and balances inter-application wear using a new strategy, while existing FTL optimizations are relevant for intra-application wear-leveling.

SSD Interface. Programmable and flexible SSD interfaces have been proposed to improve the communication between applications and SSD hardware [15, 50, 53]. SR-IOV [63] is a hardware bus standard that helps virtual machines bypass the host to safely share hardware to reduce CPU overhead. These techniques are complimentary to FlashBlox which helps applications use dedicated flash regions. Multi-streamed SSDs [33] addresses a similar problem with a stream tag, isolating each stream to dedicated flash blocks but sharing all channels, dies and planes to achieve maximum per-stream throughput. OPS isolation [34] has been proposed to dedicate flash blocks to each virtual machine sharing an SSD. They reduce fragmentation and GC overheads. FlashBlox builds upon this work and extends the isolation to channels and dies without compromising on wear-leveling.

Storage Isolation. Recent research has demonstrated that making software aware of the underlying hardware constraints can improve isolation. Shared SSD performance [56, 57] can be improved by observing the convex-dependency between IOPS and bandwidth, and also by predicting future workloads [64]. In contrast, FlashBlox identifies the relation between flash isolation and wear when using hardware isolation, and makes software schedulers aware of it. It solves this problem by helping software perform coarse time granular wear-levelling across channels and dies.

6 Conclusions and Future Work

In this paper, we propose leveraging channel and die-level parallelism present in SSDs to provide isolation for latency sensitive applications sharing an SSD. Furthermore, FlashBlox provides near-ideal lifetime despite the fact that individual applications write at different rates to their respective channels and dies. FlashBlox achieves this by migrating applications between channels and dies at coarse time granularities. Our experiments show that FlashBlox can improve throughput by 1.6x and reduce tail latency by up to 3.1x. We also show that migrations are rare for real world workloads and do not adversely impact applications' performance. In the future, we wish to take FlashBlox in two directions. First, we would like to investigate how to integrate with the virtual hard drive stack such that virtual machines can leverage FlashBlox without modification. Second, we would like to understand how FlashBlox should be integrated with multi-resource data center schedulers to help applications obtain predictable end-to-end performance.

Acknowledgments

We would like to thank our shepherd Ming Zhao as well as the anonymous reviewers. This work was supported in part by the Center for Future Architectures Research (C-FAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. We would also like to thank the great folks over at CNEX for supporting our research by providing early access to their open SSDs.

References

[1] N. Agarwal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. USENIX ATC*, Boston, MA, June 2008.

[2] Amazon Relational Database Service. `https://aws.amazon.com/rds/`.

[3] Amazon Relational Database Service Pricing. `https://aws.amazon.com/rds/pricing/`.

[4] Amazon's SSD Backed EBS. `https://aws.amazon.com/blogs/aws/new-ssd-backed-elastic-block-storage/`.

[5] Azure DocumentDB. `https://azure.microsoft.com/en-us/services/documentdb/`.

[6] Azure DocumentDB Pricing. `https://azure.microsoft.com/en-us/pricing/details/documentdb/`.

[7] Azure Premium Storage. `https://azure.microsoft.com/en-us/documentation/articles/storage-premium-storage/`.

[8] Azure Service Fabric. `https://azure.microsoft.com/en-us/services/service-fabric/`.

[9] Azure SQL Database. `https://azure.microsoft.com/en-us/services/sql-database/`.

[10] Azure SQL Database Pricing. `https://azure.microsoft.com/en-us/pricing/details/sql-database/`.

[11] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proc. USENIX ATC'11*, Berkeley, CA, June 2011.

[12] Block IO Bandwidth (Blkio) in Docker. `https://docs.docker.com/engine/reference/run/#block-io-bandwidth-blkio-constraint`.

[13] Block IO Controller. `https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt`.

[14] Y. Bu, H. Lee, and J. Madhavan. Comparing SSD-placement Strategies to scale a Database-in-the-Cloud. In *Proc. SoCC'13*, Santa Clara, CA, Oct. 2013.

[15] A. M. Caulfield, T. I. Mollov, L. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proc. ACM ASPLOS'12*, London, United Kingdom, Mar. 2012.

[16] CGROUPS. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`.

[17] 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 *Proc. HPCA'11*, San Antonio, Texas, Feb. 2011.

[18] CNEX Labs. `http://www.cnexlabs.com/index.php`.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. SoCC'12*, Indianapolis, Indiana, June 2010.

[20] FIO Benchmarks. `https://linux.die.net/man/1/fio`.

[21] Fusion-io ioDrive. `https://www.sandisk.com/business/datacenter/products/flash-devices/pcie-flash/sx350`.

[22] Google Cloud Platform: Local SSDs. `https://cloud.google.com/compute/docs/disks/local-ssd`.

[23] Google Cloud SQL. `https://cloud.google.com/sql/`.

[24] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proc. ACM ASPLOS*, Washington, DC, Mar. 2009.

[25] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proc. FAST'16*, Santa Clara, CA, Feb. 2016.

[26] J. He, D. Nguyen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Reducing File System Tail Latencies with Chopper. In *Proc. FAST'15*, Santa Clara, CA, Feb. 2015.

[27] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan. Unified Address Translation for Memory-Mapped SSD with FlashMap. In *Proc. ISCA'15*, Portland, OR, June 2015.

[28] Intel Inc. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *White Paper*, 2015.

[29] Iyswarya Narayanan and Di Wang and Myeongjae Jeon and Bikash Sharma and Laura Caulfield and Anand Sivasubramaniam and Ben Cutler and Jie Liu and Badriddine Khessib and Kushagra Vaid. SSD Failures in Datacenters: What? When? and Why? In *Proc. ACM SYSTOR'16*, Haifa, Israel, June 2016.

[30] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Proc. NSDI'13*, Berkeley, CA, Apr. 2013.

[31] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A File System for Virtualized Flash Storage. *ACM Trans. on Storage*, 6(3):14:1–14:25, 2010.

[32] M. Jung and M. K. Ellis H. Wilson III. Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks. In *Proc. ISCA'12*, Portland, OR, June 2012.

[33] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The Multi-Streamed Solid-State Drive. In *Proc. HotStorage'14*, Philadelphia, PA, June 2014.

[34] J. Kim, D. Lee, and S. H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *Proc. FAST'15*, Santa Clara, CA, Feb. 2015.

[35] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving Journaling of Journal Anomaly in Android IO: Multi-version B-tree with Lazy Split. In *FAST'14*, Santa Clara, CA, Feb. 2014.

[36] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *SIGOPS OSR*, 40(3), 2006.

[37] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proc. FAST'15*, Santa Clara, CA, Feb. 2015.

[38] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind. Application-Managed Flash. In *Proc. FAST'16*, Santa Clara, CA, Feb. 2016.

[39] LevelDB.
`https://github.com/google/leveldb`.

[40] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proc. EuroSys'14*, Amsterdam, Netherlands, Apr. 2014.

[41] N. Li, H. Jiang, D. Feng, and Z. Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proc. EuroSys'16*, London, United Kingdom, Apr. 2016.

[42] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proc. ISCA'15*, Portland, OR, June 2015.

[43] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proc. MICRO'11*, Porto Alegre, Brazil, Dec. 2011.

[44] Matias Bjorling and Javier Gonzalez and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proc. USENIX FAST'17*, Santa Clara, CA, Feb. 2016.

[45] H. Menon and L. Kale. A Distributed Dynamic Load Balancer for Iterative Applications. In *Proc. SC'13*, Denver, Colorado, Nov. 2013.

[46] Microsoft's Open Source Cloud Hardware.
`https://azure.microsoft.com/en-us/blog/microsoft-reimagines-open-source-cloud-hardware/`.

[47] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *Proc. MICRO'11*, Porto Alegre, Brazil, Dec. 2011.

[48] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *Proc. EuroSys'12*, Paris, France, Apr. 2010.

[49] J. Ouyang, S. Lin, S. Jiang, Y. Wang, W. Qi, J. Cong, and Y. Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proc. ACM ASPLOS*, Salt Lake City, UT, Mar. 2014.

[50] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proc. HPCA'11*, San Antonio, Texas, Feb. 2014.

[51] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. on Computer Systems*, 10(1):26–52, Feb. 1992.

[52] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. ISCA'11*, San Jose, CA, June 2011.

[53] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A User-Programmable SSD. In *Proc. OSDI'14*, Broomfield, CO, Oct. 2014.

[54] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *Proc. FAST'14*, Berkeley, CA, 2014.

[55] Shore-MT.
`https://sites.google.com/site/shoremt/`.

[56] D. Shue and M. J. Freedman. From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra. In *Proc. EuroSys'14*, Amsterdam, Netherlands, Apr. 2014.

[57] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proc. OSDI'12*, Hollywood, CA, Oct. 2012.

[58] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundama, M. G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. Levandoski, and D. Lomet. Schema-agnostic indexing with azure documentdb. In *Proc. VLDB'15*, Kohala Coast, Hawaii, Sept. 2015.

[59] A. Singh, M. Korupolu, and D. Mohapatra. Server-Storage Virtualization: Integration and Load Balancing in Data Centers. In *Proc. SC'08*, Austin, Texas, Nov. 2008.

[60] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt. Flash on rails: consistent flash performance through redundancy. In *Proc. USENIX ATC'14*, Philadelphia, PA, June 2014.

[61] X. Song, J. Yang, and H. Chen. Architecting Flash-based Solid-State Drive for High-performance I/O Virtualization. *IEEE Computer Architecture Letters*, 13:61–64, 2014.

[62] SQL Database Options and Performance: Understand What's Available in Each Service Tier.
`https://azure.microsoft.com/en-us/documentation/articles/sql-database-service-tiers/#understanding-dtus`.

[63] SR-IOV for SSDs.
http://www.snia.org/sites/default/files/Accelerating%20Storage%20Perf%20in%20Virt%20Servers.pdf.

[64] Sungyong Ahn and Kwanghyun La and Jihong Kim. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems. In *Proc. USENIX HotStorage'16*, Denver, CO, June 2016.

[65] TATP Benchmark.
http://tatpbenchmark.sourceforge.net/.

[66] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proc. SOSP'13*, Farmington, PA, Nov. 2013.

[67] Throtting IO with Linux.
https://fritshoogland.wordpress.com/2012/12/15/throttling-io-with-linux.

[68] Token Bucket Algorithm.
https://en.wikipedia.org/wiki/token_bucket.

[69] TPCB Benchmark.
http://www.tpc.org/tpcb/.

[70] TPCC Benchmark.
http://www.tpc.org/tpcc/.

[71] TPCE Benchmark.
http://www.tpc.org/tpce/.

[72] Traffic Control HOWTO.
http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[73] B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proc. FAST'11*, Santa Clara, CA, Feb. 2016.

[74] H. Wang and P. Varman. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In *Proc. FAST'14*, Santa Clara, CA, Feb. 2014.

[75] J. Wang and Y. Hu. WOLF: A Novel reordering write buffer to boost the performance of log-structured file systems. In *Proc. FAST'02*, Monterey, CA, Jan. 2002.

[76] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An Effective Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proc. EuroSys'14*, Amsterdam, the Netherlands, Apr. 2014.

[77] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. Don't stack your Log on my Log. In *Proc. INFLOW'14*, Broomfield, CO, Oct. 2014.

[78] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. AI-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proc. SOSP'15*, Monterey, CA, Oct. 2015.

[79] N. Zhang, J. Tatemura, J. M. Patel, and H. Hacigumus. Re-evaluating Designs for Multi-Tenant OLTP Workloads on SSD-based I/O Subsystems. In *Proc. SIGMOD'14*, Snowbird, UT, June 2014.

[80] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proc. 10th USENIX FAST*, San Jose, CA, Feb. 2012.