A Parametric I/O Model for Modern Storage Devices

Tarikul Islam Papon
Boston University
Boston, MA, USA

Manos Athanassoulis
Boston University
Boston, MA, USA

ABSTRACT

Storage devices have evolved to offer increasingly faster read/write access, through flash-based and other solid-state storage technologies. When compared to classical rotating hard disk drives (HDDs), modern solid-state drives (SSDs) have two key differences: (i) the absence of mechanical parts, and (ii) an inherent difference between the process of reading and writing. The former removes a key performance bottleneck, enabling *internal device parallelism*, whereas the latter manifests as a *read/write performance asymmetry*. In other words, SSDs can serve multiple concurrent I/Os, and their writes are generally slower than reads; none of which is true for HDDs. Yet, the performance of storage-resident applications is typically modeled by the number of *disk accesses* performed, inherently assuming *symmetric* read and write performance and the ability to perform *only one I/O at a time*, failing to accurately capture the performance of modern storage devices.

To address this mismatch, we propose a simple yet expressive storage model, termed **Parametric I/O Model** (PIO) that captures contemporary devices by parameterizing *read/write asymmetry* (α) and *access concurrency* (k). PIO enables device-specific decisions at algorithm design time, rather than as an optimization during deployment and testing, thus ensuring optimal algorithm design by taking into account the properties of each device. We present a benchmarking of several storage devices that shows that α and k vary significantly across devices. Further, we show that using carefully quantified values of α and k for each storage device, we can fully exploit the performance it offers, and we lay the groundwork for *asymmetry/concurrency-aware* storage-intensive algorithms. We also highlight that the degree of the performance benefit due to concurrent reads or writes depends on the asymmetry of the underlying device. Finally, we summarize our findings as a set of guidelines for designing storage-intensive algorithms and discuss specific examples for better algorithm and system designs as well as runtime tuning.

1 INTRODUCTION

The performance of data-intensive applications is typically bounded by the time needed to transfer data through the storage and memory hierarchy. Hence, when data resides on slow media, *disk I/O* is the primary bottleneck. As a result, measuring and modeling *disk I/O access* is often used as a proxy to performance. The traditional I/O model (also termed EM model) [1] consists of a two-level memory hierarchy with a fast internal memory of size M (which we simply call *memory*) and a slow external memory (*storage*) of unbounded size; both divided into fixed-size blocks. Any computation requires the corresponding data blocks to be in memory, which is typically orders of magnitude faster than storage. Therefore, the EM model measures cost using only the storage I/O cost (number of storage accesses). This modeling is accurate when two key assumptions hold: (i) disk reads and writes have similar cost, and (ii) applications can perform one I/O at a time – *none of which is true* for modern storage devices. The mismatch between the EM model and contemporary devices stems from the fact that it was developed for hard disk drives (HDDs), which have *symmetric* read-write performance dominated by the disk's mechanical movement. Furthermore, the mechanical parts do not allow HDDs to serve multiple concurrent requests; rather, disk accesses happen serially. In contrast, most modern storage devices are composed of electronic components.

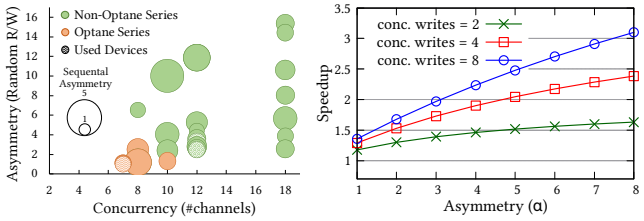


Figure 1: (A) *Asymmetry* (α) and *Concurrency* (k) in recent SSDs; most devices show high α and k. (B) Gain increases as more concurrent I/Os are used, and α dictates the gain.

Modern Devices. The majority of today's secondary storage devices are solid-state drives (SSDs) while HDDs are increasingly used as "archival" storage [48]. NAND flash-based SSDs eliminate the mechanical overhead of HDDs, thus, providing low energy consumption, high chip density, and high random read performance [2, 27, 29, 40]. These benefits can be attributed to the *internal parallelism* of SSDs, which can be exploited to optimize performance [10]. However, due to the physics of NAND flash, writes are generally slower than reads which leads to *read/write asymmetry* [14]. Fig. 1A shows that most recent SSDs have an asymmetry of more than two ($\alpha > 2$). Typically, off-the-shelf SSDs use the traditional SATA interface, while, various high-end latest devices use the PCIe interface. There are multiple alternatives in terms of underlying storage technology [8, 13, 22, 43, 45, 51], commonly known as the emerging Non-Volatile Memory (NVM) class. Most NVMs have similar properties (high asymmetry, concurrency, chip density, and low energy consumption) like SSDs [4, 34]. The vision for NVM is to offer a byte-addressable durable memory medium with performance close to main memory's. The most mature technology to date is 3D XPoint [22, 42], which is already on the market via Intel's Optane series [24]. Optane SSDs generally exhibit lower read/write asymmetry and concurrency than flash-based SSDs [53],

DAMON'21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8556-5/21/06…$15.00
https://doi.org/10.1145/3465998.3466003

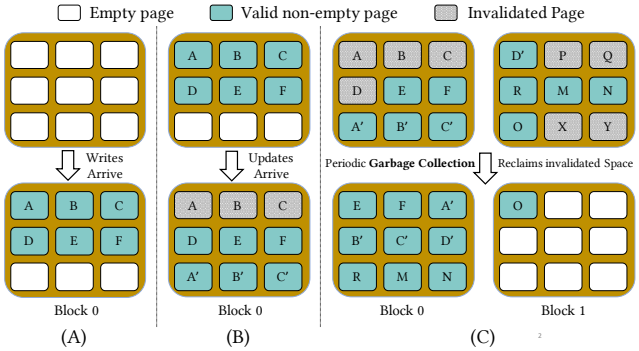


Figure 2: (A) Not all writes are costly; six pages (A-F) are written in a block. (B) Updates cause invalidation of pages which cannot be overwritten unless the whole block is erased. (C) Periodic garbage collection reclaims the invalidated pages.

which is further corroborated by our experiments in Section 2. To summarize, even though there are several variations in terms of different technologies, modern storage devices (SATA/PCIe/Optane SSDs and NVMs) share a few key properties: fast access, high chip density, low energy consumption, and most importantly, *read/write asymmetry* and *internal parallelism*.

Read/Write Asymmetry. Page updates in NAND-based SSDs follow the *erase-before-write* approach. Logical page updates (at the file system level) are always performed as *out-of-place* updates. In order to *physically* update the contents of a flash page, the containing block has to be erased first, which means once a page is written, it cannot be overwritten until that whole block is erased [2]. Fig. 2 highlights three cases for writing a flash page. When a new write request comes in a free location, it can be performed directly which is shown in Fig. 2A for pages A-F. However, once a page is written, it cannot be overwritten until that whole block is erased. So, when a page *update* arrives, the controller has to invalidate the old page and write the updated page in a new location, as shown in Fig. 2B for pages A′-C′. As a result, after a number of writes, the SSD may contain several invalidated pages. To reclaim the invalidated space, the flash controller periodically triggers *garbage collection*, which copies the valid pages of a block, writes them in a new block and then erases the previous block, as shown in Fig. 2C. The overhead of maintaining periodic garbage collection along with the extra writes results in *higher amortized write cost* [7, 14, 19, 35]. The level of asymmetry depends on the specific device and the type of access (sequential/random). Typically, writes can be up to one order of magnitude slower than reads. For example, the advertised random read and write performance of the Intel D5-P4320 SSD is 427K IOPS and 36K IOPS respectively, resulting in an asymmetry of 11.86×. Fig. 1A shows the advertised asymmetry of several recent Intel SSDs. The random read/write asymmetry is plotted along the y-axis, while the radius of the circle shows the sequential read/write asymmetry. The light colored circles correspond to devices that we used for benchmarking and experimentation. More details about these devices are presented in Section 2. The highest advertised random (sequential) asymmetry among those devices is 15.4× (4.86×). Although some devices advertise low asymmetry (e.g., Optane SSDs), in practice, most devices have an asymmetry of 2× or more. In the rest of the paper, we use the generic term

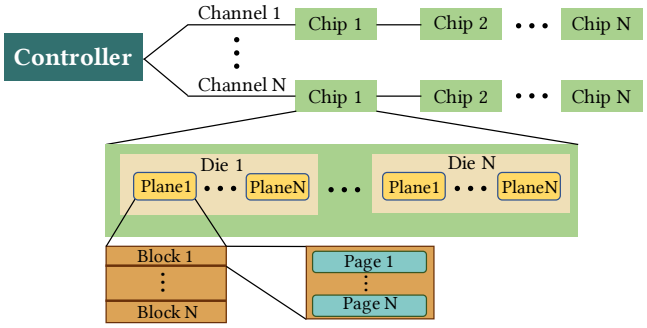


Figure 3: Internal architecture of an SSD.

read/write asymmetry (α) or *asymmetry* to refer to both random and sequential read/write asymmetry.

Concurrency. SSDs exhibit a high degree of **internal parallelism** in their architecture [9, 10, 33]. Fig. 3 shows that multiple channels are connected to the flash controller, and each channel consists of a shared bus with multiple chips. Each chip contains multiple dies, each die comprises multiple planes, and finally, each plane constitutes multiple blocks where the pages reside [2]. Hence, to fully utilize the bandwidth of an SSD, we need to submit multiple concurrent I/Os. Ideally, these I/Os will target different parts of the device, and they will be parallelized by the controller [33, 41, 49]. To quantify the supported concurrency, we refer again to Fig. 1A that shows on the x-axis the number of channels of each device. We use this as a proxy to the supported concurrency which, in most cases, is more than 8. Note that the *observed concurrency* might differ, as it depends on the access patterns of the workload, how much of the parallelism beyond the channels is exploited, and whether the operations are reads, writes, or a mix of the two.

It is essential to know the concurrency (k) and read/write asymmetry (α) of a device to fully utilize and comprehend its benefit. Fig. 1B shows the significance of k and α. We consider an application that can exploit the write concurrency by batching writes. We simulate the speedup of such an application on devices with different α values. The figure shows that as we increase the number of concurrent I/Os, the gain of exploiting parallelism increases. However, the gain closely depends on the device asymmetry – the higher the asymmetry, the higher the gain.

Storage Modeling. Without capturing asymmetry and concurrency of a storage device, we cannot attain its full potential, nor we can tailor the algorithms to the characteristics of the device, resulting in subpar performance and suboptimal device utilization. This raises the need for a new I/O model [38] that can incorporate these device properties. Now, the question we set out to answer is:

How should the I/O model be adapted in light of read/write asymmetry and concurrency?

Parametric I/O Model. We propose a simple yet expressive storage model termed Parametric I/O Model (PIO) that considers *asymmetry* (α) and *concurrency* (k) as parameters. Using the device properties, this richer I/O model can accurately capture contemporary devices. We benchmark different types of state-of-the-art storage devices to quantify their α and k. In addition, we perform an abstract modeling of different types of applications based on our proposed PIO model, and we showcase the impact of utilizing k and modeling

α. Our experiments and analysis reveal that *more informed storage modeling leads to better overall application performance.*

Contributions. Our contributions are as follows.

- We investigate the importance of the key characteristics of modern devices. We seek to answer the question: "*how much are we missing in terms of performance if we do not exploit concurrency and read/write asymmetry?*"
- We benchmark new storage devices (including Intel's 3D XPoint) to quantify k and α, and analyze the impact of the file system.
- We introduce the **Parametric I/O Model** (PIO) which considers both α and k. We show the benefits of adding these properties in our model with respect to device utilization and performance.
- We discuss *when* and *how* to exploit *which* type of concurrency. We highlight that asymmetry is key to quantifying performance.
- We outline five guidelines that should be considered during storage-intensive algorithm design.

2 QUANTIFYING α AND k

We now present a benchmarking methodology to quantify asymmetry (α) and concurrency (k) of modern storage devices. We experiment on several off-the-shelf and high-end devices. We also analyze the file system's impact on α and k as well as on performance.

Benchmarking Setup. We use a machine with two Intel Xeon Gold 6230 2.1GHz processors each having 20 cores with virtualization enabled and with 27.5MB L3 cache, 384GB of RDIMM main memory. Our experimental server has four storage devices: (i) a 375GB Optane P4800X SSD, (ii) a 1TB PCIe P4510 SSD, (iii) a 240GB SATA S4610 SSD and (iv) a 2TB HDD. We refer to these devices as *Optane SSD*, *PCIe SSD*, *SATA SSD*, and *HDD*, respectively. We also experiment with a *virtualized* storage device from Amazon AWS which is a 1.2TB Provisioned IOPS SSD allowing a maximum of 60000 IOPS. We use both Fio [18] and our custom benchmarking tool to have fine-grained experimental control. Our tool supports synchronous and asynchronous I/Os, and optionally, direct I/O, multithreading, varying queue depth, and varying block size. Our device-level benchmark uses a 29GB file on which we issue I/O requests that differ in request type (read/write), access type (sequential/random), request granularity (4K/8K), and number of threads. We enable direct I/O and use a queue depth of 32. For each experiment, we report the average IOPS, bandwidth, and latency per operation where the numbers are averaged over 10 minutes of execution. For every experiment, the standard deviation was less than 1%. The SSDs were pre-conditioned by writing 3 times prior to the experiments ensuring that the devices have stable performance [17]. Without loss of generality, in the rest of the paper, we focus primarily on random requests because (i) random accesses have lower performance with higher asymmetry than sequential, and (ii) in our experiments, we find that the overall trends and observations for sequential accesses are similar to the ones for random accesses.

Computing α & k. We define *concurrency* as the number of concurrent I/Os needed to saturate the device bandwidth. Hence, in these experiments, we increase the number of threads issuing concurrent I/Os for each device to identify the point when the device reaches its maximum bandwidth. It is worth mentioning that for most cases, the increase of the bandwidth is not linear. As a result, we handpick a point where the device bandwidth is close to saturation or where the rate of bandwidth increase drops significantly. In this way, we quantify the concurrency of a device empirically rather than using the specifics of the internal device architecture. Note that most devices do not disclose all the design details as they are proprietary. The interested reader can also find in Appendix A the analytical approximation of the supported concurrency from a device with a known number of channels. The *read/write asymmetry* is calculated by taking the ratio of the maximum read vs. write IOPS (bandwidth) obtained from these experiments.

Table 1: Performance comparison of the tested devices.

Device	Latency (RR)	Latency (RW)
Optane SSD	6.8 μs	7.6 μs
PCIe SSD	12.4 μs	19.6 μs
SATA SSD	100 μs	130 μs
Virtual SSD	180 μs	200 μs
HDD	7000 μs	7300 μs

Optane SSD and PCIe SSD. Fig. 4A and 4B show the read and write throughput of our Optane SSD and PCIe SSD, respectively, as we vary the number of threads issuing concurrent I/Os on the x-axis. The dotted lines indicate the point where the device bandwidth reaches (close to) the saturation point for the corresponding access pattern. These figures highlight the significance of utilizing device concurrency. For example, there is a 40× increase in the PCIe SSD's read bandwidth when using full concurrency compared to no concurrency; for the Optane SSD, this increase is 4×. We also observe a few key differences between the two devices: (i) the Optane SSD is faster than the PCIe SSD, offering 3× higher write bandwidth, (ii) the Optane SSD gets saturated quickly indicating that it has lower concurrency ($k \approx 6$), and (iii) the Optane SSD exhibits a very small degree of asymmetry ($\alpha \approx 1.1$). Fig. 4C shows the latency profile and the impact of concurrency for the Optane SSD. As we increase the number of concurrent I/Os, initially there is almost no increase in latency while the bandwidth increases. When the device bandwidth gets saturated, that latency increases quickly. The *supported concurrency* (k) of the device is the number of concurrent I/O issued when we have this sudden latency increase. Fig. 4B shows that asymmetry and concurrency also depend on the access granularity. For instance, when using 4KB blocks, $\alpha \approx 2.8$, while for 8KB blocks, $\alpha \approx 1.9$. In addition, the supported concurrency depends on both the block size and the request type (read/write). For instance, the PCIe SSD gets saturated when issuing 80 concurrent 4KB random read I/Os ($\alpha \approx 2.8,\ k \approx 80$) whereas for 8KB random writes $k \approx 7$. Finally, we observe that for a low number of concurrent I/Os, writes are actually faster than reads. This is attributed to caching benefits when the controller's memory is enough to capture all the concurrent writes and flush them as a batch. As the number of concurrent I/O increases, garbage collection comes into action, consequently exhibiting somewhat slower, steady-state write performance.

Regular SSD. Fig. 4D presents the performance profile of our SATA SSD which closely follows the trend of the PCIe SSD. This device shows lower asymmetry and much lower concurrency compared to the PCIe SSD, but higher than that of the Optane SSD. For example, for 4KB random read requests, $\alpha \approx 1.5$ and $k \approx 25$. The device

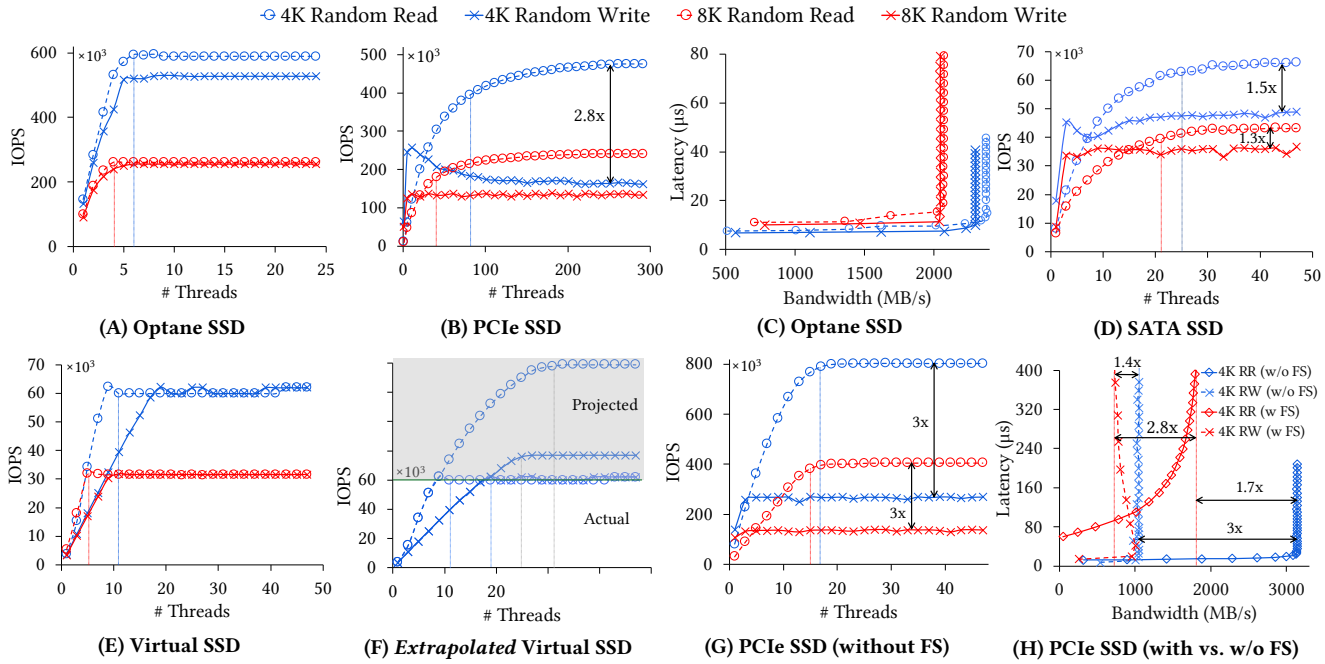


Figure 4: All devices show different degree of k and α depending on the file system, access type, and block size. The benefit of exploiting k can be as high as 40×. (A) The Optane SSD has lower k and $\alpha \approx 1$. (B) The PCIe SSD shows high k with α up to 2.8. (C) Latency does not increase until saturation. (D) The SATA SSD shows lower k than the PCIe SSD with α up to 1.5. (E) The Virtual SSD is software-controlled, exhibiting $\alpha \approx 2$ prior to saturation. (F) The extrapolated performance profile of the Virtual SSD following the trends of other SSDs. (G, H) Bypassing the file system allows for 1.7× (1.4×) higher read (write) throughput for PCIe SSD, while the values of both α and k are more stable across the experiments.

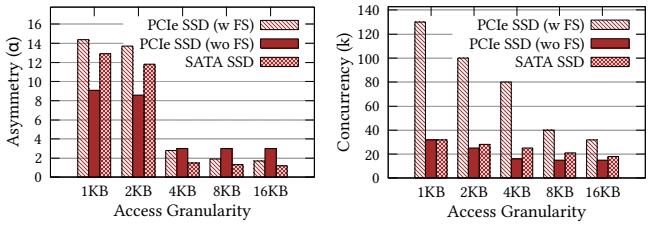
gets saturated quickly in the event of concurrent writing; for 8KB random write, we find $\alpha \approx 1.3$ and $k \approx 5$.

Virtual SSD. Fig. 4E benchmarks the Virtual SSD. For 4K random read requests, there is a 16× throughput improvement when increasing the number of concurrent I/Os from 1 to 11, at which point the device gets saturated. Hence, for a small number of concurrent I/Os, the device remains underutilized and the client does not receive the performance they paid for, resulting in monetary loss. There are two more observations from this graph: (i) since this is a virtual SSD, the device's peak bandwidth is software-capped, yet, by comparing the slope for the 4K read and write performance, we observe that this device has $\alpha \approx 2$, and (ii) the graph is more stable than the graphs of PCIe SSD and SATA SSD i.e., the increase in IOPS is a perfect straight line and there is very little noise in the graph, which can be attributed to the device being virtual, and its performance being managed by the software. Fig. 4E shows that for 4KB access, the read (write) bandwidth reaches its maximum for 11 (19) concurrent I/Os. Note that these values are not the actual concurrency of the device, rather, at this point the maximum allocated IOPS (60K) is achieved and the software layer caps the device bandwidth. We speculate that the actual concurrency of the device is potentially much higher. Fig. 4F shows an extrapolated profile of the underlying physical device of the Virtual SSD, assuming similar performance patterns to the other tested devices and no software limit in the attainable IOPS.

HDD. Our experiments show that the HDD has $\alpha = 1$ and $k = 1$ which is expected. We omit the performance graphs for brevity.

Optane SSD is the Fastest. Our benchmarking shows that the Optane SSD is the fastest, which is 1.8× faster than the PCIe SSD and the access latency can be as low as 6.8μs. We also find that the PCIe SSD is 8× faster than the SATA SSD and the SATA SSD is 70× faster than our HDD. Table 1 lists the access latency of all the devices for 4KB random read/write requests.

Bypassing the File System for PCIe SSD. In our previous experiments, we observe that the PCIe SSD has a maximum read and write bandwidth of 1900 MB/s and 700 MB/s for 4KB block size, while the read latency can be as low as 12μs. For such a fast device, the software latency of the file system and the OS kernel can be significant. This device offers raw access to the storage medium by completely bypassing the file system through Intel's Storage Performance Development Kit (SPDK) [50]. SPDK provides a set of libraries for writing high-performance, scalable, user-mode applications. To run an SPDK application (i) the device must not have an active file system, (ii) hugepages must be allocated to facilitate its driver (2GB by default), and (iii) the device must be bound to a Virtual Function IO (VFIO) kernel driver rather than the native kernel drivers to allow direct device access to userspace. Using SPDK and its *perf* tool, we analyze how α and k change, and how performance is affected when we perform raw access to this device. We issue random I/O requests, varying the request type (read/write), the request granularity (4K/8K) and the number of threads.



(A) Asymmetry **(B) Concurrency**

Figure 5: (A) The asymmetry is high when I/O size is smaller than native page size; as I/O size increases α decreases. (B) The concurrency decreases for larger I/O size.

Fig. 4G presents the performance profile of the PCIe SSD with raw access. The smoothness of the graph indicates that the device has stable performance. The figure shows that the device has $\alpha = 3$ for either access granularity. Similar to previous experiments, k depends on the request type and access granularity. For example, for 4KB random requests $\alpha \approx 3$, and $k \approx 16$ for reads. Although the concurrency value is lower, there is a considerable gain in both the maximum read bandwidth (1.7×) and the maximum write bandwidth (1.4×) as compared to Fig. 4B. Fig. 4H shows the latency/bandwidth profile of the PCIe SSD with and without the file system, clearly supporting the thesis that for a high-performance device, the software layer becomes a bottleneck. Traditionally, applications interact with storage via the OS using an interrupt-based model. Although the interrupt model has an overhead, historically this overhead was negligible compared to disk latency. However, with the emergence of high-performance storage devices that use faster protocols like NVMe [25, 55] and technologies like 3D XPoint [22, 42], the file system overhead is not negligible any more [28, 54], rather, *the storage bottleneck is shifting from hardware to software*, which is corroborated by our experiments.

Impact of Access Granularity. Our previous experiments hinted that k and α depend on the access granularity. In this experiment, we further vary the access granularity between 1KB and 16KB to see this dependency in more detail. Fig. 5A and 5B shows how α and k vary with respect to I/O size for three different devices. Fig. 5A shows that for access size smaller than the native page size (4KB), the asymmetry is very high for all the devices. For example, for 1KB I/O size, the SATA SSD and PCIe SSD (with FS) show asymmetry of 13 and 14.4 while for 4KB these values drop down to 1.5 and 2.8. This is because even for a small write (i.e., 1KB or 2KB), the write must be performed at page level. This results in a lot of updates on the SSD block, which consequently trigger excessive garbage collection that in turn significantly slows down the writes. Hence, writes smaller than the native page size should be avoided. Fig. 5B shows that as I/O size increases, the device bandwidth reaches maximum with fewer I/Os. This is because larger I/O size causes higher data transfer which quickly saturates the device bandwidth.

Characterizing a Storage Device. To create a succinct representation of each device, we now put together the findings from our experiments with three metrics: asymmetry (α), read concurrency (k_r), and write concurrency (k_w). Table 2 shows the values of these metrics for all tested devices for 4KB and 8KB accesses. We note that while for some cases, there is a positive correlation between α and k_r/k_w, the two quantities are not equal. Specifically, we define asymmetry as $\alpha = BW_r^{max}/BW_w^{max}$ where BW_r^{max} and BW_w^{max} correspond to the maximum read and write bandwidth respectively. Further, we can express BW_r^{max} and BW_w^{max} as a function of the number of threads that saturate each operation as follows: $BW_r^{max} = f_r(k_r)$ and $BW_w^{max} = f_w(k_w)$. Following our definition for α, we get $\alpha = f_r(k_r)/f_w(k_w)$. The functions $f_r(\cdot)$ and $f_w(\cdot)$ quantify the attained bandwidth, however, $f_w(\cdot)$ includes the amortized cost of garbage collection. Hence, the two functions are neither identical nor linear, and in general $\alpha \neq k_r/k_w$.

Table 2: Empirical *Asymmetry* and *Concurrency*.

Device	4KB			8KB		
	α	k_r	k_w	α	k_r	k_w
Optane SSD	1.1	6	5	1.0	4	4
PCIe SSD (with FS)	2.8	80	8	1.9	40	7
PCIe SSD (w/o FS)	3.0	16	6	3.0	15	4
SATA SSD	1.5	25	9	1.3	21	5
Virtual SSD	2.0	11	19	1.9	6	10

Summary. We observe from Table 2 that all the NAND-based SSDs (PCIe, SATA, and Virtual SSD) exhibit asymmetry and concurrency. The value of α and k varies across devices depending on the internal of the device, access granularity, access pattern and filesystem. Bypassing the file system can unlock higher performance and stabilize the observed asymmetry and concurrency (Table 2 and Fig. 4G). Optane SSDs generally exhibit low asymmetry and concurrency because of their 3D Xpoint technology [53] and our experiments corroborate this. We notice, however, that even exploiting the *low* concurrency in Optane SSD leads to 4× higher throughput. While the performance specifics vary significantly among different devices [5, 56], the vast majority of modern storage devices exhibit a non-trivial degree of both asymmetry and concurrency. Hence, storage-intensive applications can benefit from a more faithful modeling to fully exploit the underlying device and to predict the expected performance benefits.

3 PARAMETRIC I/O MODEL

In this section, we present the **Parametric I/O Model** (PIO), a new, simple yet expressive model that takes asymmetry and (read and write) concurrency as parameters. PIO enables better algorithm design and helps to accurately reason about the performance of storage-intensive algorithms and data structures.

> ***PIO*(M, k_r, k_w, α)** *assumes a fast main memory with capacity M, and storage of unbounded capacity that has* ***read/write asymmetry*** *α, and* ***read (write) concurrency*** *k_r (k_w).*

We consider that both memory and storage are divided into fixed-size blocks. Since the model is device-specific, the values of k_r, k_w, and α are either given by the device manufacturer, or by a benchmarking process similar to the one used in Section 2. PIO allows us to accurately describe a variety of devices, and reason for their behavior at algorithm-design time. That way, we can make storage-aware optimizations part of the design, as opposed to applying them as an additional *ad hoc* tuning step during deployment. Next, we present how to use PIO to reason about the performance benefits of several fundamental classes of applications.

3.1 Performance Analysis

To analyze the performance of a storage-intensive application, we focus on its interaction with the storage device, that is, on the read and write requests it issues. We classify storage-intensive applications into four classes.

- *Unbatchable Reads, Unbatchable Writes.* An application that cannot batch reads nor writes. We include this class only for completeness and to highlight the impact of asymmetry.
- *Unbatchable Reads, Batchable Writes.* An application that batches writes and utilizes the write concurrency of the device (*example*: concurrent eviction of dirty pages from a bufferpool).
- *Batchable Reads, Unbatchable Writes.* An application that batches reads and utilizes the read concurrency (*example*: concurrent traversal of multiple paths in a graph or in a tree index).
- *Batchable Reads and Writes.* An application that batches both reads and writes and utilizes both read and write concurrency (*example*: LSM-tree compaction).

To maintain the generality of the approach, we quantify the performance gain using the frequency of reads (f_r) and writes (f_w), for which $f_r + f_w = 1$. We assume that read requests have unit cost (1) and write requests have cost α, where $\alpha \geq 1$. A device with read concurrency of k_r and write concurrency of k_w can concurrently perform k_r reads and k_w writes.

Unbatchable Reads, Unbatchable Writes. We first consider the class of applications that cannot batch reads or writes and performs them sequentially. While this class seems to be of no interest as it cannot exploit any concurrency, we include it for completeness and to highlight the importance of asymmetry. Since reads have unit cost and writes have α cost, the normalized cost per operation for this type of application is $f_r + f_w \cdot \alpha$. Fig. 6 shows this cost as we vary the read/write ratio in the workload. For a device with higher asymmetry, the cost increases as more writes are introduced in the workload. This figure highlights that when there is asymmetry, treating reads and writes equally leads to performance degradation. Rather, the slower writes should ideally be masked either algorithmically, or if possible, via batching.

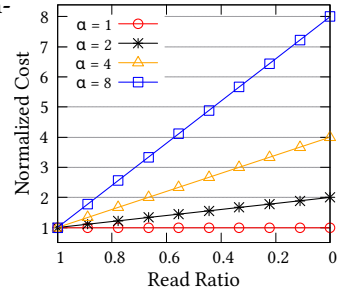


Figure 6: Higher asymmetry leads to higher cost since the more expensive write cost is not amortized (through concurrency or otherwise).

Unbatchable Reads, Batchable Writes. This class of applications exploits the write concurrency of the device to batch write requests. As an example, consider a modified DBMS bufferpool manager that selects several dirty pages and writes them concurrently during an eviction. In this scenario, the application at hand attempts to fully exploit the device's write concurrency via *concurrent flushing* during a page eviction. To achieve this, it submits k_w concurrent writes. Since the device has α asymmetry, the cost of a write following the standard approach of evicting one page at a time, as indicated by the EM model would be $C_W^{EM} = \alpha$. On the other hand, the *amortized* cost per write when we batch k_w writes following PIO is $C_W^{PIO} = \frac{\alpha}{k_w}$. Since reads are not batchable in this application class, each read will have unit cost in both the EM and PIO paradigm, hence, $C_R^{EM} = C_R^{PIO} = 1$. We can now calculate the speedup S_{PIO} of this application based on PIO:

$$S_{PIO} = \frac{f_r \cdot C_R^{EM} + f_w \cdot C_W^{EM}}{f_r \cdot C_R^{PIO} + f_w \cdot C_W^{PIO}} = \frac{f_r + f_w \cdot \alpha}{f_r + f_w \cdot \frac{\alpha}{k_w}} \implies$$

$$S_{PIO} = \frac{k_w \cdot (f_r + f_w \cdot \alpha)}{k_w \cdot f_r + f_w \cdot \alpha} = 1 + \frac{(k_w - 1) \cdot f_w \cdot \alpha}{k_w \cdot f_r + f_w \cdot \alpha}$$

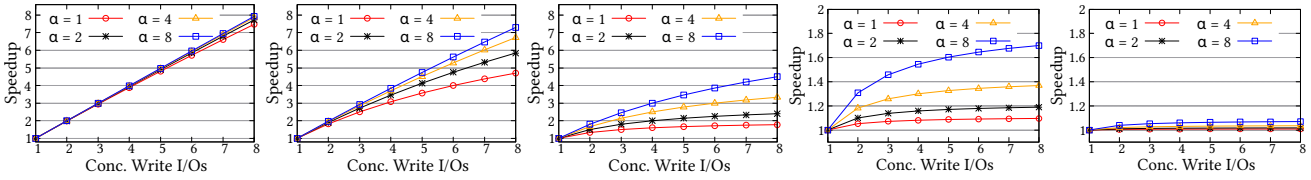
Since $\alpha \geq 1$ and $k_w \geq 1$, then $S_{PIO} \geq 1$ where the maximum value of S_{PIO} can be up to k_w. Fig. 7 shows the speedup when following PIO for different α and k_w values as we change the read/write ratio in the workload. We observe that the speedup increases as more concurrent I/Os are employed, which is expected. Furthermore, we note that the speedup depends on the asymmetry of the device. The *gain is **higher** for a device with **higher** asymmetry*. For example, for $f_r = 0.5$ and when fully exploiting the concurrency of $k_w = 8$ by issuing 8 concurrent I/Os, the speedup for a device with $\alpha = 8$ is 4.5× whereas the speedup for a device with $\alpha = 1$ is 1.78× (Fig. 7C). Since the application batches writes, the gain is maximized for a write-intensive workload (Fig. 7A), when the benefit from efficient writing is more pronounced. For a workload that contains *only* reads or *only* writes, the speedup from the PIO paradigm is the same irrespective of α. The key observation is that for an application with batchable writes, a higher asymmetry between expensive writes and cheap reads leads to a higher speedup.

Batchable Reads, Unbatchable Writes. The second class of applications models scenarios where reads can be issued concurrently to exploit read concurrency, but not writes. As an example, consider a graph store that traverses multiple paths concurrently, thus can accelerate various algorithms including graph search and shortest path. Essentially, the algorithm can visit multiple nodes in parallel instead of one node at a time, and offer faster search time with the same worst-case guarantees. Another example is concurrent traversal of tree indexes [44]. The application performs k_r reads concurrently, thus, $C_R^{EM} = 1$ and $C_R^{PIO} = \frac{1}{k_r}$. The cost of a write request is $C_W^{EM} = C_W^{PIO} = \alpha$ for both paradigms since the writes are not batched. The speedup of this second class of applications is:

$$S'_{PIO} = \frac{f_r \cdot 1 + f_w \cdot \alpha}{f_r \cdot \frac{1}{k_r} + f_w \cdot \alpha} = \frac{k_r \cdot (f_r + f_w \cdot \alpha)}{f_r + k_r \cdot f_w \cdot \alpha} = 1 + \frac{(k_r - 1) \cdot f_r}{f_r + k_r \cdot f_w \cdot \alpha}$$

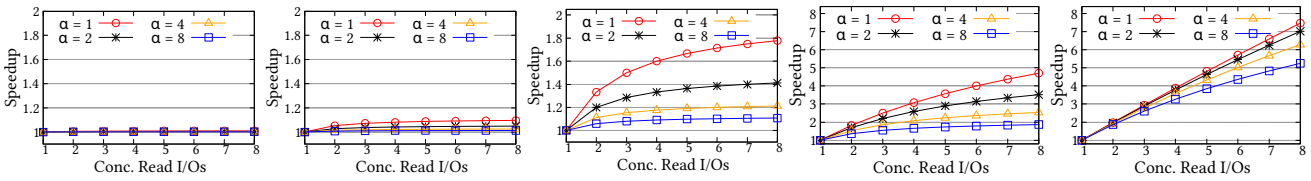
Since $\alpha \geq 1$ and $k_r \geq 1$, then $S'_{PIO} \geq 1$, and also $S'_{PIO} \leq k_r$. Fig. 8 presents the speedup of such an application based on PIO. Like before, the speedup increases as more concurrent I/Os are used. However, this time *the gain is **higher** for a device with **lower** asymmetry*. For instance, with $f_r = 0.5$ and $k_r = 8$, the speedup for a device with $\alpha = 1$ is 1.78× and it drops to 1.11× for $\alpha = 8$ (Fig. 8C). Note that, the overall speedup is lower than the previous application class because, while writes are still more expensive than reads (for $\alpha > 1$), this class of applications can only utilize read concurrency. The speedup is maximized when the workload is read-heavy (Fig. 8A), showing the benefit of batching reads.

Batchable Reads and Writes. The third class of applications can batch both read and write requests. A notable example of such an application that can concurrently issue many concurrent read and write requests without any interdependency are LSM-trees, and specifically their compaction routine. During a compaction, multiple read/write streams are involved since files from the selected



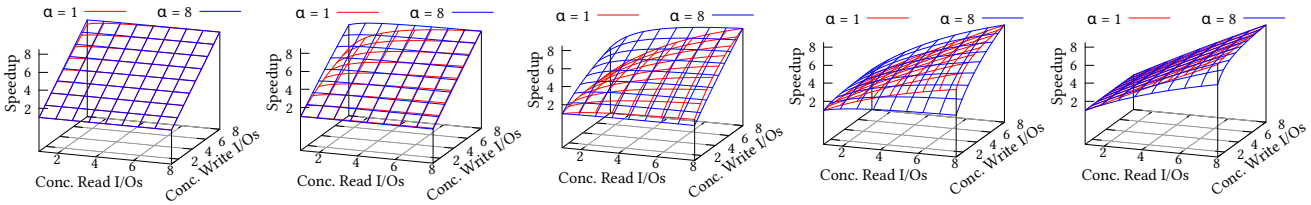
(A) 1% **reads** **(B)** 10% **reads** **(C)** 50% **reads** **(D)** 90% **reads** **(E)** 99% **reads**

Figure 7: Speedup of an *Unbatchable Reads, Batchable Writes* application. The speedup is highest for write-intensive workloads. Furthermore, the speedup depends on the device asymmetry – *the higher the asymmetry, the higher the speedup.*



(A) 1% **reads** **(B)** 10% **reads** **(C)** 50% **reads** **(D)** 90% **reads** **(E)** 99% **reads**

Figure 8: Speedup of a *Batchable Reads, Unbatchable Writes* application. The speedup is highest for read-intensive workloads. Speedup depends on the device asymmetry, however, the trend is reversed – *the lower the asymmetry, the higher the speedup.*



(A) 1% **reads** **(B)** 10% **reads** **(C)** 50% **reads** **(D)** 90% **reads** **(E)** 99% **reads**

Figure 9: Speedup of a *Batchable Reads and Writes* application. (A, B) For fewer reads, speedup increases with more conc. write I/Os irrespective of conc. read I/Os. (C, D, E) Read concurrency comes into action as more reads are introduced in the workload.

levels are read, sort merged and written to the next level. Since the whole process invokes numerous reads and writes, the compaction scheduler can use PIO to decide the degree of concurrency. Specifically, it can start as many concurrent compactions as needed to fully exploit the read concurrency and the write concurrency. The cost of a classical read is $C_R^{EM} = 1$, and of a classical write is $C_W^{EM} = \alpha$. Since this class of applications can batch both reads and writes, the amortized PIO costs are $C_R^{PIO} = \frac{1}{k_r}$ and $C_W^{PIO} = \frac{\alpha}{k_w}$. The speedup of the third class of applications is:

$$S''_{PIO} = \frac{f_r \cdot 1 + f_w \cdot \alpha}{f_r \cdot \frac{1}{k_r} + f_w \cdot \frac{\alpha}{k_w}} = \frac{k_r \cdot k_w \cdot (f_r + f_w \cdot \alpha)}{k_w \cdot f_r + k_r \cdot f_w \cdot \alpha}$$

The maximum value of S''_{PIO} can be up to k_r or k_w depending on the workload. For fewer reads (Fig. 9A, 9B), the speedup increases as more concurrent write I/Os are issued irrespective of the number of concurrent read I/Os. For workloads with more reads (Fig. 9C, 9D, 9E), the impact of concurrent read I/Os becomes more prominent. Similarly to the previous class of applications, the speedup depends on the device asymmetry. Overall, we observe that the impact of utilizing write concurrency is higher than utilizing read concurrency because of the asymmetry.

Impact of Asymmetry (α). The above analysis reveals that the speedup from exploiting concurrency depends on the asymmetry of the device. For an *Unbatchable Reads, Batchable Writes* application, the speedup is higher for a device with higher asymmetry because the higher cost of writes is amortized through batching. On the other hand, for a *Batchable Reads, Unbatchable Writes* application, the speedup is higher for devices with lower asymmetry, because batching actually further reduces the amortized cost of reads. In other words, batching reads exacerbates the impact of read/write asymmetry. To summarize, the degree of the performance gain/loss depends on the asymmetry and the application type, whereas the gain is achieved through exploiting concurrency.

Importance of using the *Actual k*. The speedup of an application depends on carefully exploiting the actual device concurrency. To demonstrate this, we implement a sample application with *unbatchable reads and batchable writes* (a bufferpool that batches writes of dirty pages and flushes them concurrently during eviction). We run this *concurrency-aware* application on our PCIe SSD that exhibits $k_w = 8$, and we vary the number of concurrently issued write-backs during eviction. Fig. 10 shows the speedup when comparing with an application that always evicts a single page, for a workload with $f_r = 0.5$. As we increase the number of concurrent evictions towards the device's $k_w = 8$, the speedup increases as expected, however, after this point, the speedup drops. This is

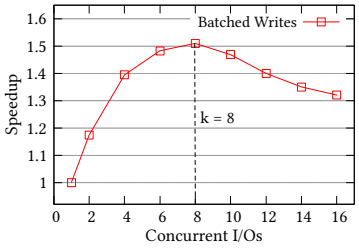


Figure 10: Speedup increases as more concurrent I/Os are used until the device is saturated.

because (i) the device's supported parallelism is 8, hence, there is no benefit from issuing more concurrent writes, and (ii) for higher number of concurrent I/Os, the software overhead of thread management also increases. Hence, the speedup drops after reaching this threshold, indicating that we should refrain from increasing the concurrent requests (especially writes) further than the concurrency supported by the device.

4 ALGORITHM DESIGN GUIDELINES

We now distill five guidelines from our previous discussion that should drive the development of new storage-intensive algorithms.

Guideline 1. *Know Thy Device*

When a new device is used, we frequently focus on the raw performance (throughput and latency), however, modern storage devices are complex software-hardware systems that also exhibit read/write asymmetry, as well as read and write concurrency. In §2 and §3, we see that we need to know and appropriately use both these properties to get the best out of our devices. Hence, before deploying a device it is crucial to benchmark it to quantify its PIO parameters: asymmetry (α), and read (k_r) and write (k_w) concurrency.

Guideline 2. *Exploit Device Concurrency*

When a storage-intensive application has readily available information for multiple read and/or write operations, it should exploit the available parallelism of the device. The modeled concurrency is only an approximation of the actual capabilities of the device (which depend both on the internals of the device and the data access patterns). However, operating at a rate close to the benchmarked concurrency will yield the best device throughput. In addition, algorithms that have been designed with the "one I/O at a time" or "one stream of I/Os at a time" approach should be redesigned. For example, a bufferpool can concurrently write-back multiple dirty pages and concurrently prefetch multiple pages, a graph search can visit multiple nodes in different paths concurrently and cover the graph faster, and an LSM-tree can tune its compaction according to how many concurrent streams can be supported.

Guideline 3. *Use Device Concurrency With Care*

In the effort to fully exploit the bandwidth of a device we might end up pushing the device beyond its limits. In a simple benchmark, read bandwidth simple plateaus for a high number of concurrent I/Os, however, this is not the case for write-intensive or more complex applications. For such scenarios, using the *appropriate* k, i.e., exactly what the device supports, is key to achieve the *optimal* (advertised) performance, while pushing for higher concurrency might have adverse effects (Fig. 10). Further, real-life applications may have other bottlenecks (locking, synchronization, logging, etc.) that can further diminish the benefits of concurrent accesses.

Guideline 4. *It is suboptimal to treat equally a page read and a page write for a device with asymmetry*

While it may seem natural to consider both a page read and a page write equally in terms of performance because of how main memory and hard disk behave, this is not the case for modern storage devices. A page read and a page write follow a very different execution path, and building algorithms and data structures that treat them equally in terms of performance leads to suboptimal designs. To address this, when operating on modern storage devices, page writes should either be delayed (hoping that some will be avoided) or they should be amortized through concurrent writing. Overall, performing one write to facilitate one read or vice versa (e.g., when a bufferpool is saturated) is a suboptimal design in the presence of asymmetry.

Guideline 5. *Read/Write Asymmetry Controls Performance*

For applications that have read and write requests, the device asymmetry controls the magnitude of the performance benefits due to utilizing concurrency (Fig. 7, 8, and 9). Consider the class of applications with batchable writes. Devices with higher asymmetry benefit more from writing in parallel. On the other hand, for applications with batchable reads, devices with higher asymmetry see smaller benefits, because the asymmetrically high cost of writes cannot be amortized. For example, a bufferpool that is saturated and has to evict a dirty page will exchange a write for a read. If the device exhibits asymmetry, this approach is not *fair*, because one write is more expensive than one read. Hence storage-intensive algorithm design should be both *concurrency* and *asymmetry* aware.

5 DISCUSSION

Redesigning Algorithms & Operators. The Parametric I/O model captures the behavior of modern storage devices, and can be used to guide algorithm design for external storage. In this paper we focus on a set of abstract workloads that represent different types of applications, which can benefit from capturing read/write asymmetry and concurrency. Following the same spirit, this premise can be applied to various components of data systems. As PIO ensures *better storage utilization* by considering the specific device properties, we envision better algorithm design for almost any component of a system that interacts with storage.

Tree and Graph Traversal. In light of PIO, tree and graph traversal algorithms can be redesigned to access multiple nodes concurrently, thus, having a wider search space at the same time. Consider a variation of the depth-first search (DFS) algorithm that can offer DFS guarantees, while at the same time covering a wider search-space by loading concurrently sibling nodes to the degree the underlying concurrency can support it. Conversely, one can construct an algorithm with breadth-first-search (BFS) guarantees that follows a few promising paths as deep as they might get.

Query Optimizer Cost Models. Consider a query optimizer that determines which *join* algorithm to employ using cost models [21]. While state-of-the-art optimizers differentiate between sequential random accesses [21], typically, they do not differentiate between reads and writes. For modern storage devices, however, this misses the opportunity to account for the read/write asymmetry. By including asymmetry in the cost-model, we can make more accurate query optimizing decisions. Moreover, the join algorithms themselves can be designed to leverage device concurrency when reading/writing data blocks from/to storage.

Bufferpool. Another component of a database system that can benefit from PIO is the bufferpool. In fact, several bufferpool implementations (e.g., InnoDB, DB2) already exploit write concurrency by batching multiple writes and exploiting background flushing [16, 23]. Using PIO, we can carefully design a concurrency eviction

policy that accounts both for the device write concurrency and the device asymmetry to guarantee a balanced workload execution.

LSM-trees Compactions. Finally, a data-intensive operation that can also benefit from PIO is the Log-structure merge (LSM) tree's *compaction* process. LSM-trees are widely used as the storage layer of many NoSQL key-value and other data stores [15, 32, 37, 46]. LSM-trees write on disk immutable sorted runs and once a level reaches its threshold, a *compaction* is performed in order to reorganize the data between the saturated level and the next level. An LSM-tree typically initiates multiple compactions at the same time, and each compaction merges a number of sequential streams to a single output. Hence, using PIO, the compaction scheduler can decide how many and which compactions to initiate to better exploit the underlying device.

Overall, incorporating read/write asymmetry (α) and concurrency (k) in algorithm design allows for customizability for different devices which leads to more faithful storage modeling and, ultimately, to better device utilization.

Automatic Tuning Using PIO. Further, the benchmarking presented in Section 2 shows that the characteristics of the devices are more accurately captured through careful experimentation, rather than using the advertised performance. Hence, we propose to use our benchmarking methodology to characterize a storage device with respect to their asymmetry (α) and concurrency (k_r and k_w). This is a one-step analysis that allows to carefully tune PIO-aware algorithms or data structures.

6 RELATED WORK

Existing Models. External storage has been traditionally modeled as a simple collection of blocks following the simplicity of the design of a hard disk (EM Model [1]). Blelloch et al. [6, 7] proposed the *Asymmetric RAM* (ARAM) model to analyze algorithms for asymmetric read and write costs, targeting asymmetric non-volatile main memory devices. There are two primary differences between ARAM and PIO. First, ARAM targets main memory that has smaller access granularity. Second, it does not take into account the parallelism of modern storage devices which is central to PIO. The main goal of ARAM is to develop write-efficient main memory algorithms. On the contrary, the goal of PIO is to capture the inherent asymmetry and concurrency of *storage devices*, and study how we can use these in the design process of *storage-intensive algorithms*.

Addressing Read/Write Asymmetry and Concurrency. The read/write asymmetry on storage has been identified as an optimization goal for indexing [3, 11, 12, 30, 31, 52], flash-aware storage engines [36], and other data management operations [20, 39]. In the context of exploiting device parallelism, recent research builds new I/O schedulers for SSDs [33, 41, 49], and designs new data structures [9, 26, 44, 47]. Our work bridges these efforts on addressing asymmetry and concurrency under a unified approach, making both α and k first-class citizens of storage device modeling. PIO lays the groundwork for considering α and k at algorithm-design time, rather than as an optimization during development or deployment.

7 CONCLUSION & FUTURE WORK

The classical I/O model which was developed considering traditional HDDs, cannot accurately model modern storage devices. Contemporary storage devices are characterized by a *read-write asymmetry* and an access *concurrency*, both of which are essential to fully utilize the device. We present a benchmarking process on five storage devices to quantify these properties, and we discuss their implications in performance. We propose a simple yet expressive parametric I/O model, termed PIO, that considers the asymmetry (α) between reads and writes, and concurrency (k) that different devices may support to enable better algorithm design. By capturing α and k, device-specific decisions can be tuned at both algorithm design time and during deployment and testing. We illustrate the impact of PIO on different classes of workloads and outline five guidelines that should drive storage-intensive algorithm design. We envision better algorithm design for *any* component of a system that interacts with the storage. Specifically, we envision an *asymmetry/concurrency-aware* bufferpool manager that prefetches reads concurrently and batches dirty evictions to exploit the device parallelism. Further, we envision that algorithms for tree and graph traversal can be redesigned to access multiple nodes concurrently, thus avoiding the classical paradigm of accessing one node at a time, ensuring better device utilization.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback and the members of DiSC lab for their useful remarks. We are particularly grateful to Subhadeep Sarkar for his feedback. This work was partially funded by National Science Foundation under Grant No. IIS-1850202, a Facebook Faculty Research Award and RedHat.

A EXPECTED PARALLELISM IN MULTI-CHANNEL DEVICES

We approximate the expected parallelism of a multi-channel SSD under uniform I/O distribution. Even if multiple I/O requests are issued, it is not guaranteed that all of the device's channels will be used. It is possible that the I/O requests target blocks are located in the same channel. It is also possible that the scheduler of the I/O requests in the flash controller cannot always disperse the I/Os across different channels. Hence, here, we approximate how many of the channels will be occupied in response to multiple concurrent random I/Os on average.

Consider a device that has n channels and a total of m I/O requests has been issued in these channels. For the sake of generality, we assume a uniform distribution of the I/O requests across the channels. We are interested in the total number of channels that will be occupied with the I/O requests. To calculate that, we first calculate using recursion, the expected number of channels that will have no I/O request to serve. We first define E_m.

$$E_m = \text{expected number of empty channels after } m \text{ I/Os}$$

By definition, after assigning the first $(m-1)$ I/Os to channels, there will be E_{m-1} empty channels. Now, the m-th I/O must be assigned either to an empty channel or a non-empty channel. The probability of assigning it to an empty channel is $\frac{E_{m-1}}{n}$, because there are E_{m-1} empty channels out of the n channels. In other words, $\frac{E_{m-1}}{n}$ of the time, there will be $(E_{m-1}-1)$ empty channels, while the rest of the time $(1-\frac{E_{m-1}}{n})$, there will be E_{m-1} empty channels. We now express E_m recursively in Eq. (1).

$$E_m = \frac{E_{m-1}}{n} \cdot (E_{m-1} - 1) + \left(1 - \frac{E_{m-1}}{n}\right) \cdot E_{m-1} = \frac{n-1}{n} \cdot E_{m-1} \quad (1)$$

Since, when we have received zero requests all channels are empty ($E_0 = n$), the recursion becomes $E_m = n \cdot ((n-1)/n)^m$. So, the fraction of channels that will be empty is $E_m/n = ((n-1)/n)^m = ((1-(1/n))^n)^{\frac{m}{n}}$ which can be approximated by $(1/e)^{\frac{m}{n}}$.

If we allow a device to concurrently serve as many I/O requests as the number of channels ($m \approx n$), then the fraction of idle channels is $(1/e) = 37\%$. As we issue more concurrent I/Os, the fraction of idle channels reaches 0%. We consider an effective concurrency of a device the number of I/Os needed for E_m to approach zero, or $e^{-m/n} \rightarrow 0$. We numerically calculate that when $m \approx 3n$, then $E_m = e^{-m/n} = e^{-3} \approx 0.05$. So, when the device controller allows the number of concurrent I/Os to be 3× the number of channels, we utilize $(1 - E_m) = 95\%$ of the device's channels. We use this analysis as a guide to the level of concurrency that can be efficiently supported by modern storage devices, issuing concurrent I/Os between 1× and 3× the number of the device's internal parallelism.

REFERENCES

[1] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 57–70.

[3] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1881–1892.

[4] Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. 2012. Path Processing using Solid State Storage. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 23–32.

[5] R Bishnoi, M Ebrahimi, F Oboril, and M B Tahoori. 2016. Improving Write Performance for STT-MRAM. *IEEE Transactions on Magnetics* 52, 8 (2016), 1–11.

[6] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. 2015. Sorting with Asymmetric Read and Write Costs. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1–12.

[7] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. 2016. Efficient Algorithms with Asymmetric Read and Write Costs. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*. 14:1–14:18.

[8] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, V Nikitin, X Tang, S Watts, S Wang, S A Wolf, A W Ghosh, J W Lu, S J Poon, M Stan, W H Butler, S Gupta, C K A Mewes, T Mewes, and P B Visscher. 2010. Advances and Future Prospects of Spin-Transfer Torque Random Access Memory. *IEEE Transactions on Magnetics* 46, 6 (2010), 1873–1878.

[9] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage (TOS)* 12, 3 (2016), 13:1–13:39.

[10] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 266–277.

[11] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[12] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[13] Youngdon Choi, Ickhyun Song, Mu Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, Junho Shin, Yoohwan Rho, Changsoo Lee, Min Gu Kang, Jaeyun Lee, Yongjin Kwon, Soehee Kim, Jaehwan Kim, Yong Jun Lee, Qi Wang, Sooho Cha, Sujin Ahn, Hideki Horii, Jaewook Lee, Kisung Kim, Hansung Joo, Kwangjin Lee, Yeong Taek Lee, Jeihwan Yoo, and Gitae Jeong. 2012. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. 46–48.

[14] Michael Cornwell. 2012. Anatomy of a Solid-State Drive. *Communications of the ACM (CACM)* 55, 12 (2012), 59–63.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94.

[16] I B M DB2. 2021. Configuring storage for performance. *https://www.ibm.com/docs/en/db2-for-zos/12?topic=performance-configuring-storage* (2021).

[17] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proceedings of the VLDB Endowment* 14, 3 (2020), 364–377.

[18] Fio. 2021. Flexible I/O tester. *https://fio.readthedocs.io/en/latest/* (2021).

[19] Eran Gal and Sivan Toledo. 2005. Algorithms and data structures for flash memories. *Comput. Surveys* 37, 2 (2005), 138–163.

[20] Yan Gu, Yihan Sun, and Guy E Blelloch. 2018. Algorithmic building blocks for asymmetric memories. In *Leibniz International Proceedings in Informatics, LIPIcs*, Vol. 112. 44:1—-44:15.

[21] Laura M Haas, Michael J Carey, Miron Livny, and Amit Shukla. 1997. Seeking the Truth About ad hoc Join Costs. *The VLDB Journal* 6, 3 (1997), 241–256.

[22] Frank T Hady, Annie P Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.

[23] MySQL InnoDB. 2021. Configuring Buffer Pool Flushing. *https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool-flushing.html* (2021).

[24] Intel. 2016. Intel® Optane™ Technology. *https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html* (2016).

[25] Jon L. Jacobi. 2019. NVMe SSDs: Everything you need to know about this insanely fast storage. *https://www.pcworld.com/article/2899351/everything-you-need-to-know-about-nvme.html* (2019).

[26] Aarati Kakaraparthy, Jignesh M Patel, Kwanghyun Park, and Brian Kroth. 2019. Optimizing Databases by Learning Hidden Parameters of Solid State Drives. *Proceedings of the VLDB Endowment* 13, 4 (2019), 519–532.

[27] Jeong-Uk Kang, Heeseung Jo, Jinsoo Kim, and Joonwon Lee. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*. 161–170.

[28] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a True Direct-Access File System with DevFS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 241–256.

[29] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)* 6, 3 (2007).

[30] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, Ke Yi, and Robin Jun Yang. 2010. Tree Indexing on Solid State Drives. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1195–1206.

[31] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[32] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[33] Bo Mao, Suzhen Wu, and Lide Duan. 2018. Improving the SSD Performance by Exploiting Request Characteristics and Internal Parallelism. *IEEE Trans. on CAD of Integrated Circuits and Systems* 37, 2 (2018), 472–484.

[34] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. 2014. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters* 9, 1 (2014), 526.

[35] Suman Nath and Phillip B. Gibbons. 2008. Online Maintenance of Very Large Random Samples on Flash Storage. *Proceedings of the VLDB Endowment* 1, 1 (2008), 970–983.

[36] Suman Nath and Aman Kansal. 2007. FlashDB: dynamic self-tuning database for NAND flash. *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN)* (2007).

[37] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[38] Tarikul Islam Papon and Manos Athanassoulis. 2021. The Need for a New I/O Model. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[39] Hyoungmin Park and Kyuseok Shim. 2009. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software* 82, 8 (2009), 1298–1312.

[40] Stan Park and Kai Shen. 2009. A performance evaluation of scientific I/O workloads on Flash-based SSDs. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 1–5.

[41] Stan Park and Kai Shen. 2012. FIOS: a fair, efficient flash I/O scheduler. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 13.

[42] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 13:1–13:8.

[43] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Yi-Chou Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4-5 (2008), 465–480.

[44] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *Proceedings of the VLDB Endowment* 5, 4 (2011), 286–297.

[45] Guido De Sandre, Luca Bettini, Alessandro Pirola, Lionel Marmonier, Marco Pasotti, Massimo Borghi, Paolo Mattavelli, Paola Zuliani, Luca Scotti, Gianfranco Mastracchio, Ferdinando Bedeschi, Roberto Gastaldi, and Roberto Bez. 2011. A 4 Mb LV MOS-Selected Embedded Phase Change Memory in 90 nm Standard CMOS Technology. *J. Solid-State Circuits* 46, 1 (2011), 52–63.

[46] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908.

[47] Jinho Seol, Hyotaek Shim, Jaegeuk Kim, and Seungryoul Maeng. 2009. A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 137–146.

[48] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. 2016. United States Data Center Energy Usage Report. *Ernest Orlando Lawrence Berkeley National Laboratory* LBNL-10057 (2016).

[49] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 67–78.

[50] SPDK. 2016. Storage Performance Development Kit (SPDK). *https://spdk.io* (2016).

[51] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80–83.

[52] Stratis D. Viglas. 2012. Adapting the B +-tree for asymmetric I/O. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 7503 LNCS. 399–412.

[53] Kan Wu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2019. Towards an Unwritten Contract of Intel Optane SSD. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*.

[54] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 323–338.

[55] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the ACM International Systems and Storage Conference (SYSTOR)*. 6:1–6:11.

[56] Jianhui Yue and Yifeng Zhu. 2013. Accelerating write by exploiting PCM asymmetries. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*. 282–293.