Evaluating and Repairing Write Performance on Flash Devices

Radu Stoica
EPFL, VD, Switzerland
radu.stoica@epfl.ch

Manos Athanassoulis
EPFL, VD, Switzerland
manos.athanassoulis@epfl.ch

Ryan Johnson
EPFL, VD, Switzerland
CMU, PA, USA
ryanjohn@ece.cmu.edu

Anastasia Ailamaki
EPFL, VD, Switzerland
anastasia.ailamaki@epfl.ch

ABSTRACT

In the last few years NAND flash storage has become more and more popular as price per GB and capacity both improve at exponential rates. Flash memory offers significant benefits compared to magnetic hard disk drives (HDDs) and DBMSs are highly likely to use flash as a general storage backend, either alone or in heterogeneous storage solutions with HDDs. Flash devices, however, respond quite differently than HDDs for common access patterns, and recent research shows a strong asymmetry between read and write performance. Moreover, flash storage devices behave unpredictably, showing a high dependence on previous IO history and usage patterns.

In this paper we investigate how a DBMS can overcome these issues to take full advantage of flash memory as persistent storage. We propose new a flash aware data layout — append and pack — which stabilizes device performance by eliminating random writes. We assess the impact of append and pack on OLTP workload performance using both an analytical model and micro-benchmarks, and our results suggest that significant improvements can be achieved for real workloads.

Keywords

Flash memory, Data layout, Storage virtualization, Database Management Systems

1. INTRODUCTION

Since the introduction of flash memory in the early 90's, flash storage devices have gradually spread to more and more applications and are now a standard storage solution in embedded devices, laptops and even desktop PCs. This trend is driven by the exponential growth of flash chip capacity (flash chips follow Moore's law as they are manufactured using the same techniques and equipment as integrated circuits), which drives down exponentially their price per GB.

When compared to HDDs, flash has several advantages such as lower access time and lower power consumption, better mechanical reliability, and smaller size. All these properties make flash memory a promising replacement for HDDs for a wide variety of applications. Flash storage, however, does not behave as magnetic disks and can not be abstracted as such. Although the actual hardware is hidden through interfaces such the SCSI protocol or the block device API in the OS, applications are typically optimized for accesses to rotating media storage, characterized by fixed sequential bandwidth and by mechanical delays that severely limit random IO performance. Database systems are arguably one of the best examples of applications tailored specifically to the properties of HDDs; every component from query optimizers to SQL operators to low-level disk management assumes rotational media with long random access times.

Flash devices challenge the practice of modeling persistent storage as magnetic disks as their underlying physics dictate several important differences between the two technologies. First, flash devices have no moving parts and random accesses are not penalized by long seek times or rotational delays, so sequential read accesses are no longer necessary. Second, asymmetric performance between reads and writes arises because NAND memory cells cannot be written directly (updated in place) but instead require slow and costly erase-then-write operations. In addition, NAND cell erasure must be performed for a large number of flash pages (typical values today are 64-128 pages of 2KiB each), while pages can be read individually. Whenever a page is written its physical neighbors must be moved elsewhere to allow the erase operations, causing additional data copies and exacerbating the performance asymmetry between reads and writes. Third, because of the elevated voltage required by an erase operation, NAND cells can only tolerate a limited number of erasures. To prolong the life of flash chips, the firmware must spread erase operations evenly to avoid wearing out individual areas and use error correction codes (ECCs) to ensure data integrity.

Flash device manufacturers try to hide the aforementioned issues behind a dedicated controller embedded in the flash drive. This abstraction layer, called the Flash Translation Layer (FTL), deals with the technological differences and presents a generic block device to the host OS. The main advantage of an FTL is that it helps to maintain backward compatibility, under the assumption that the generic

Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN 2009) June 28, 2009, Providence, Rhode-Island

block device is the prevailing abstraction at the application level. In practice, however, most performance-critical applications developed over the last 30 years are heavily optimized around the model of a rotating disk. As a result, simply replacing a magnetic disk with a flash device does not yield optimal device or DBMS performance, and query optimizer decisions based on HDD behavior become irrelevant for flash devices. Moreover, the complex and history-dependent nature of the interposed layer affects predictability.

An additional problem with database applications is that FTL implementations are not primarily targeted to the IO patterns of database applications, but must be general enough to accommodate various types of applications and file systems. File system IO patterns are characterized in most cases by read-intensive workloads, with infrequent sequential writing of files, and exhibit both spatial and temporal data locality. In contrast, database applications — especially in OLTP workloads — generate small (typically 8kiB) accesses which are randomly distributed, with any locality filtered out by a large buffer pool.

In this paper we argue that data layout on flash devices should embrace the important differences compared to HDDs, rather than approaching flash storage through the conventional rotating drive model. For example, in Figure 1(a) we see that for a high end flash storage card [1] throughput drops significantly after extensive writing in random locations (24 hours of random 8kiB writes). A decrease in performance is expected from flash device manufacturers since the devices are not designed for random-write-heavy workloads. The observed decrease, however, is more than an order of magnitude. On the other hand, when we perform sequential writes after random writes on the same device, we observe that performance not only stabilizes but increases back to its initial high throughput as seen in Figure 1(b). Crucially, significant performance increases can be achieved if we can transform random write accesses into sequential ones and, at the same time, increase the IO size when writing. To demonstrate the need for change in flash device abstraction, we present as a proof-of-concept an analytical model, as well as emulation results of a scheme that performs the aforementioned transformations, using state-of-the-art flash devices. We were able to surpass the performance predicted by our conservative model, achieving a speedup as high as 9.6x comparing to the performance of the flash device treated as a traditional block device.

The rest of the paper is organized as follows. In section 2 we review previous work, presenting various approaches available to optimize applications for flash technologies, in section 3 we present a proof-of-concept algorithm, along with its analytical modeling. In section 4 we present our experimental setup and finally we conclude in section 5.

2. RELATED WORK

Bouganim et al [4] devise a set of micro-benchmarks to quantify flash storage performance, and their findings provide the starting ground for this work. They experiment with several different devices, ranging from USB thumb drives to high-end Solid State Disks (SSDs) and make the important observation that behavior of flash devices depends strongly on IO history. Random writes, in particular, cause large variations in response times even after random writing is stopped. We corroborate these findings and show additional reasons why to consider the peculiarities of flash memory.

Several log-structured file systems [6], [5], [2] target flash devices specifically. All have their root in a proposal by Rosenblum and Ousterhout [10] and try to improve write performance to the detriment of read performance. Some, such as Journaling Flash File System (JFFS2)[11], also provide wear leveling in the absence of a dedicated flash controller. We argue that similar approaches are needed for database systems, especially for OLTP workloads where random writes dominate and can quickly become the performance bottleneck to persistent storage.

Lee et al. [9] show that, when used as a drop-in replacements for HDDs, SSDs can improve performance of certain database operations by 2-10x with no software changes or other optimizations. However, their workloads do not contain the small-sized random writes which penalize flash performance, and we argue that this specific property of flash memory cannot be ignored if flash is to generally replace HDDs.

There are other attempts to improve write performance on flash memory. Lee and Moon [8] propose an In-Page Logging (IPL) method for storing the redo log records for a database together with the data and eliminate the need to perform additional write IO for writing the transactional log. Additional transactional log IO operations can be removed by combining IPL with other data layout schemes such as LGeDBMS [7], proposed by Kim et al. Although removing log writes improves performance by reducing IO counts, it converts sequential log writes into random writes to data pages, leaving the larger problem of random writes unaddressed. Previous work [4] and our experiments 1(a) show that random writes, even in multiples of the flash erase block sizes, lead to severe performance degradation.

3. OUR APPROACH

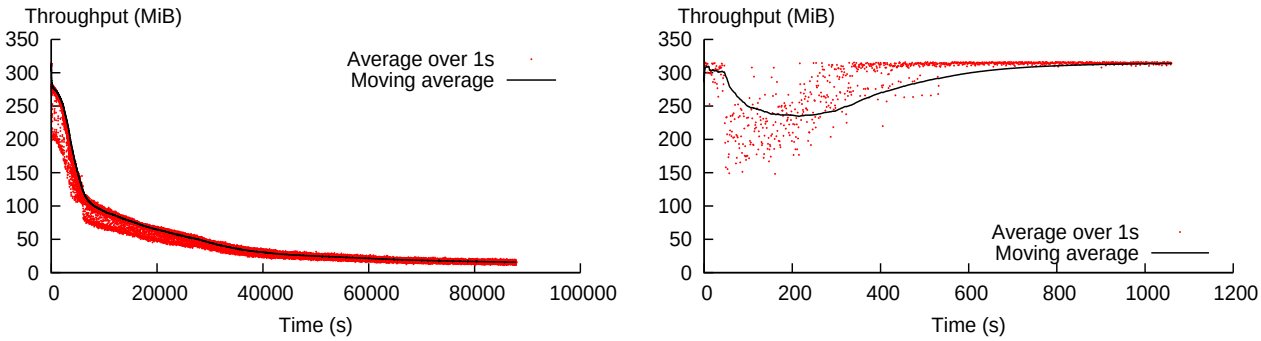
In this section we present a new DBMS data layout, *Append and Pack* , for flash memory. The algorithm avoids the problematic performance of flash memory in case of small random updates by transforming all random writes to sequential writes at the expense of additional overhead and loss of locality when reading. If the additional overhead is low, this trade-off is attractive because random reads perform similarly to sequential ones.

3.1 Random Writes in Flash Devices

Whenever a page of a flash device needs to be overwritten and no pre-erased page is available, the FTL must perform the following operations:

- Identify a block to erase,
- Read all the valid pages from the block,
- Erase the block,
- Write back the valid pages,
- Finally, write the initial page.

The erase operation is slow (milliseconds) and the additional work required to move the valid pages can add a significant overhead as well. Thus, the total cost of an erase operation depends on two factors. The first one is over how many write operations the erase cost is amortized, i.e. the ratio of page updates to erase operation. Usually the flash cells can sustain only a limited number or erase operations (10^4–10^5 in today's devices). As a result, this parameter not only impacts performance, but also dictates the maximum



(a) Random writes (b) Sequential writes

Figure 1: (a) Random write and (b) sequential write throughput. Each dot represents the average throughput over 1s, while the solid line is the moving average.

amount of data that can be written before the end of life of a flash device. The second factor is how many valid pages need to be moved at each erase and is a measure of the additional work that the device firmware must perform to allow erase operations.

An ideal case would be that there are no valid page copies needed and only one erase operation is required for updating a full erase block. We argue that is imperative to change the way applications perform IO in order to achieve optimal performance. First, random writes should be avoided on flash devices, possibly trading off read performance or at the expense of additional IO. Second, all writes should be performed in sizes big enough to overwrite one or more full erase blocks in each IO operation.

3.2 An Append and Pack Data Layout

To approach ideal flash memory performance we propose a new method of storing database pages. We create inside the DBMS storage manager an abstraction layer between the logical data organization and the physical data placement. The layer always writes dirty pages, flushed by the buffer manager of the overlying DBMS, sequentially and in multiples of the erase block size. From a conceptual point of view, the physical database representation is an append-only structure. Previous versions of database pages are not overwritten in place but are instead logically invalidated. In this manner we eliminate the need to move valid data at an erase operation because a whole erase block is written every time. In addition, only one erase operation is performed for a number of pages equal to the erase block size. As a result, our writing mechanism benefits from optimal flash memory performance as long as enough space is available.

When space is low (we cannot continue appending pages), space occupied by invalidated pages must be reclaimed. The proposed layer consolidates the least recently updated logical pages, starting from the head of the append structure, packs them together, then writes them back sequentially to the tail. In the common case a small number of valid pages is scattered across many erase blocks, allowing a single pack operation to reclaim a large amount of space. To minimize the amount of data moved we use two datasets levels. The first dataset stores the *write-hot* database pages while the second stores the *write-cold* database pages. The insight is that by grouping pages with similar update frequencies together we minimize the number of pages that need to be moved during space reclamation. As mentioned previously, page packing is performed by reading valid data from the head of the hot dataset and writing them sequentially; we append them to the write-cold dataset because pages which reach the beginning of the hot dataset have gone the longest without being updated and are therefore likely to be write-cold. In practice, the cold log structure also sees database page updates, although at a much lower frequency; when such an update happens, the page is promoted again to the hot dataset, where it is appended as any other page. We find that, for OLTP workloads, this strategy successfully distinguishes between write-hot and write-cold data and gives good performance even for shifting access patterns.

Because pages in the cold dataset can also be invalidated a packing algorithm is needed as well; we read data from the head of the cold log structure and write them to the end, again sequentially and in multiples of the erase block size.

The cost of the packing process is a function of the probability that a database page is still valid when it reaches the head of the log. This observation is valid both for the hot and cold datasets. In turn, the probability that a page is valid depends on two important factors: on the update probability distribution and on the ratio between the physical capacity available versus the actual space occupied by the logical database pages. We calculate in the following subsection the probability that a page is valid when reaching the beginning of a log structure, providing that all pages in that dataset have uniform update probabilities. We find that the number of valid pages reaching the head of a log decreases exponentially with the physical capacity available to store a dataset.

3.3 Analytical Model

To show the benefits of our approach we consider an OLTP workload characterized by a 50%/50% mix of random read and write accesses with no sequential patterns. This pattern matches disk traces collected from a Microsoft SQL Server running the industry standard TPC-C [3] benchmark. In order to estimate the efficiency of Append and Pack algorithm compared with a traditional page layout we first establish a relation between the duration of a random write for Append and Pack compared with the traditional implementation of in-place page updates. Let us assume that the average time to perform a random write is T_{RW}. In our case the new average response time to write to the hot dataset is

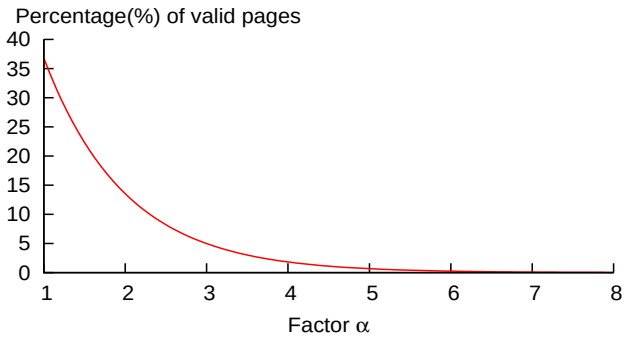


Figure 2: Percentage of pages that need to be copied to cold set when reclaiming space, for different values of α parameter.

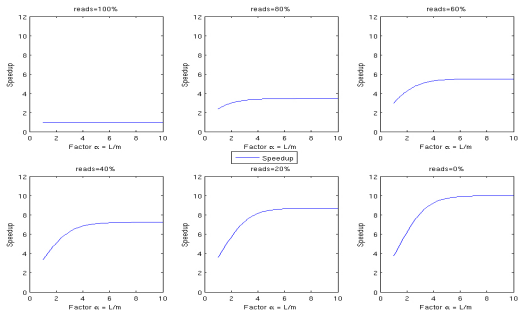


Figure 3: Achieved speedup as a function of the random read/write ratio and the α parameter.

Device	Random Read	Random Write
Intel x25-E	35,000	3,300
Memoright GT	10,000	500
Solidware P100	10,000	1,000

Table 1: Advertised flash SSD performance for random accesses (numbers taken from the specification sheets of various SSD vendors).

equal to the time needed to perform a write sequentially plus the average time spent in reclaiming space (packing invalid pages by copy to the cold dataset):

$$T_{RW}^{'} = T_{SW} + T_{pack}$$

In turn the time required to reclaim a page is equal with the time needed to move the page to the second log structure if valid when reaching the start of the hot log (with probability p_h), plus the time needed to perform page packing in the cold log structure (with probability p_c):

$$T_{pack} = p_h \cdot (T_{copy} + p_c \cdot T_{copy})$$

Assuming that the cold log structure needs to be packed significantly less often than the hot one, ($p_h \gg p_c$):

$$T_{pack} \approx p_h \cdot T_{copy} \quad \text{where} \quad T_{copy} = T_{RR} + T_{SW}$$

This implies

$$\begin{aligned} T_{RW}^{'} &= T_{SW} + p_h \cdot T_{copy} \\ T_{RW}^{'} &= T_{SW} + p_h \cdot (T_{RR} + T_{SW}) \end{aligned}$$

Assuming a uniform probability distribution for page updates at each log structure, we can plot the probability p_h as a function of the ratio between of the physical capacity available and the actual space occupied by the logical pages (see Figure 2). Considering $C_{physical}$ reasonably large (over 1GiB), the function $p_h = f(\alpha)$ converges quickly to $e^{-\alpha}$:

$$p_h(\alpha) \to e^{-\alpha} \quad \text{where} \quad \alpha = \frac{C_{physical}}{C_{logical}}$$

The probability that a page is valid, when page packing is performed, decreases exponentially as α increases. For a value of α greater than 3, more than 95% of reclaimed pages are invalid and need not be copied.

To estimate the efficiency of our algorithm we must define the costs of an IO. In the traditional case, for an OLTP workload with random accesses we have:

$$T_{IO} = r_{RR} \cdot T_{RR} + (1 - r_{RR}) \cdot T_{RW}$$

The above equation describes the time required to perform a random read multiplied by the ratio of reads in the workload (r_{RR}) plus the time taken by a random write multiplied by the ratio of random writes ($r_{RW} = 1 - r_{RR}$).

For our algorithm we have the new average IO time as:

$$T_{IO}^{'} = r_{RR} \cdot T_{RR} + (1 - r_{RR}) \cdot (T_{SW} + p_h \cdot T_{copy})$$

As a result we can define the speedup achieved by our algorithm as:

$$speedup = \frac{T_{IO}}{T_{IO}^{'}}$$

The above equations show the effect of the probability that page reclamation finds a valid page, and in addition they consider the effect of the read/write ratio. Random write intensive workloads benefit most from our algorithm, while a workload showing no random writes at all already achieves good performance on flash and should see no improvement.

To compute the actual speed-up we must first establish a relation between the cost of a random read (T_{RR}) and the cost of a random write (T_{RW}). We fix the cost of a random page write at ten times the cost of a random read: $T_{RW} = 10 \cdot T_{RR}$. To support our assumption we show in Table 1 advertised values for random read and write IO/s for the widespread SSDs available on the market. The values of random write IO/s are usually an order of magnitude lower than for random reading and, in our experience, achieving the advertised random write performance is difficult in practice and can happen only in controlled circumstances. We also assume that sequential write and random read perform similarly ($T_{SW} \approx 2 \cdot T_{RR}$).

Considering the performance difference random write and a random read, we show in Figure 3 a conservative theoretical estimate of the speed-up achieved for an OLTP workload while varying the ratio of read/write operations. The maximum theoretical speed-up ranges from 2x in case of a workload mix with 10% random writes to a speed-up of over 6x in case the read and write IOs are in equal proportions. It is interesting to note that important speedup gains are achieved for small values of α and for $\alpha \geq 3$ performance is maximized even for write heavy workloads.

4. EXPERIMENTAL EVALUATION

In order to assess the real-world behavior of our proposal we have implemented a version of Append and Pack as a standalone dynamic linked library outside of a database. The library exports only two functions, *pread* and *pwrite* that accept the same arguments as the standard system calls and can be a drop-in replacement for them. The main argument for such an implementation is the speed of development and the fact that the library can be applied to existing DBMS systems, without the need for source code.

In addition to the algorithm presented, the library also provides an automatic buffering of writes to match a desired granularity. This approach is particularly useful as the buffer management code does not need to be modified. On the other hand, automatic buffering can invalidate the durability property of a DBMS, in case of a crash before buffered data are written to disk. In a full implementation the shim library would also intercept *fflush* commands; alternatively the database buffer manager could take advantage of our algorithm and explicitly write multiple database pages in each operation. In either case, all the ACID guarantees are preserved with minimal changes to the buffer pool management code or to the storage layer.

4.1 Setup and Methodology

We experiment with a PCIe Fusion ioDrive card [1], running under Linux 2.6.18 64-bit kernel. The OS driver of the card implements also the FTL layer and uses the host's CPUs and main memory for this purpose.

The card has a capacity of 160GB and offers sustained read bandwidth of 700MiB/s and more than 100,000 random read IO/s. The device advertises sequential write throughput of 600MB/s, but in our experiments, the maximum sequential write throughput obtained was roughly 350MB/s. Also, the advertised random writing performance (of over 10^5 IO/s) was not stable and decreased over time to less than 5,000 IO/s. We attribute the initial good performance and slow degradation to the advanced FTL layer that can afford to use the CPU and memory of the host system, being much more powerful than an embedded flash controller.

4.2 Results

We first investigate the performance of a workload composed only of random writing having a parallelism (number of concurrent threads) of 15. We have found that after 15 threads, the throughput of the device does not increase anymore and the device behavior did not change. In Figure 1(a) we present the throughput averaged over a period of 1s dependent on time. The throughput decreased over time from an initial value of 322MiB/s to an average of less than 25MiB/s over a running time period of 24 hours. We found the behavior of the FusionIO drive while sustaining random writing to be difficult to predict. Nevertheless, this kind of behavior is expected when referring to flash devices in general, as both recent research [4] and device specifications show. Depending on the state of the drive, the cost of a random write relative to a sequential write varies from a factor of 2 up to 46. An interesting observation is that adding pauses in the IO pattern temporarily helps throughput when IO is resumed. The device performance seems to be connected to the pause length, which implies that the device state is improved by some background process in the FTL during idle periods. Unfortunately, we did not find a clear correlation between the pause length and the duration of throughput improvement to use an IO pattern that maximizes overall performance. As a result none of our experiments attempt to optimize by adding artificial pauses between IOs. Such an optimization depends heavily on the underlying device and we argue that is not a feasible solution for real applications.

RR/RW	Baseline	Algorithm	SpeedUp	Prediction
50/50	38MiB/s	349MiB/s	9.1	6.2
75/25	48MiB/s	397MiB/s	8.3	4.3
90/10	131MiB/s	541MiB/s	4.1	2.5

Table 2: Speedup achieved using the Append and Pack data layout and predicted speedup, having $\alpha = 2$, $T_{RW} = 10 \cdot T_{SW}$, $T_{SW} = 2 \cdot T_R$.

α	Baseline	Algorithm	SpeedUp	Prediction
2	38MiB/s	349MiB/s	9.1	6.2
3	38MiB/s	336MiB/s	8.8	6.7
4	38MiB/s	365MiB/s	9.6	6.9

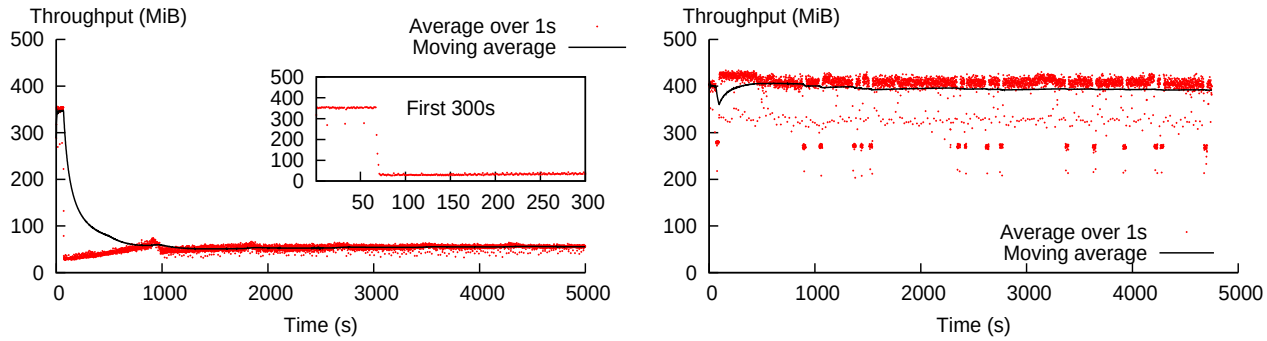
Table 3: Speedup achieved using Append and Pack data layout and predicted speedup, having $RR/RW = 50\%/50\%$, $T_{RW} = 10 \cdot T_{SW}$, $T_{SW} = 2 \cdot T_R$.

In Figure 4(a) we show a representative evolution of device throughput for a workload composed of an even mix of random read and random write operations. Varying the percentage of write operations changes the average throughput but not the decreasing trend. In all cases, the duration of random write operations limits the maximum performance the device can deliver. For example, adding 10% random writes to a read-only workload cuts performance by 90%.

In Figure 1(b), we present the averaged throughput of a flash device performing sequential writes after a preparation step of random writes, which were used to "dirty" the device state with random page updates. As one can observe, the throughput of sequential writing is affected in the beginning by the device state but after a period of time, corresponding to a full sequential write of the device, the performance becomes stable and remains constant at 320MiB/s for the rest of the experiment. The FusionIO card performs far more stably in case of sequential writes than in case of random writes. Repeating the same experiment by applying a random write workload followed by a mix of random reads and sequential writes we observe that throughput improves with a similar trend as in Figure 1(b). The initial performance is affected by the previous random IO workload but slowly recovers and becomes stable with time.

The fact that sequential writing performs more than an order of magnitude better than random writing and stabilizes the performance of the device, verifies our assumptions that by eliminating random writes from a workload increases significantly and stabilizes the performance of the underlying device. To assess the performance impact of α and the read/write ratio we repeat the same experiments but using the algorithm from section 3.2.

In Figure 4(b) we present the total throughput (reads plus writes) over time using the Append and Pack data layout. As compared to Figure 4(a), the total throughput was increased by about an order of magnitude on the long term. The predictability of the response times is worse than pure sequential writing, presented in Figure 1(b), but far bet-



(a) Traditional IO (b) Append and Pack

Figure 4: Throughput of random IO operations (read plus write) in case of (a) traditional IO and of (b) Append and Pack . Each dot represents the average throughput over 1s, while the solid line represents the moving average. Read and write operations are in equal ratios.

ter than of the initial random workload. We believe that the space reclamation algorithm can be improved even further by tuning the number of pages moved at each iteration by the garbage collection algorithm and by performing this activity in the background. However, even without additional optimizations this proof-of-concept implementation shows clearly the potential gains of using a flash-aware data layout.

In Table 2 we present the average throughput of the experiments using the Append and Pack data layout. The α parameter is set equal to 2 for both the hot dataset and the cold dataset, yielding on average a probability p_h of 13% and a p_c close to zero as pages in the cold log structure are invalidated by appends in both datasets). The parameter that we vary is the percentage of random writes over the total IO operations. The experiments show larger speedups for higher percentages of reads in the mix, as the prediction from the model suggested. We observe that the model is indeed conservative, specifically about the difference factor assumed between performance of random writes and sequential writes.

Finally, in Table 3 we present results for experiments with fixed a read/write ration at 50% and varying the value of the α parameter between 2 and 4. We observe that even for α equal to 2 we can achieve significant increase of the device throughput.

5. CONCLUSIONS

In this paper we make a case for the need for change in how flash devices are abstracted. Flash storage should not be treated by following the model of a rotating disk, but its specifics should be taken into consideration. The main differences from HDDs are very fast random reads, asymmetrically slower random writes (by an order of magnitude) and efficient sequential writes (one order of magnitude faster than random writing).

We show the potential benefit of an Append and Pack flash-aware data layout using an analytical model and by emulating database accesses on a storage device. The Append and Pack algorithm transforms temporal locality as provided by the overlying application, to spatial locality when data is placed on persistent storage, which allows to improve write performance. We show that predictions of the conservative theoretical model not only hold in practice, but they are surpassed leading to a gain in performance of up to 9x.

6. ACKNOWLEDGMENTS

This work was partially supported by Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and SNF funds.

7. REFERENCES

[1] The FusionIO drive. Technical specifications available at: http://www.fusionio.com/PDFs/Fusion%20Specsheet.pdf.

[2] The LogFS file system. Available at: http://logfs.org/logfs/ .

[3] Transaction Processing Performance Council (TPC). TPC Benchmark C: Standard Specification. Available online at: http://www.tpc.org/tpcc/spec/tpcc_current.pdf .

[4] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[5] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys*, pages 176–187, 2004.

[6] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *In Proceedings of the Winter 1995 USENIX Technical Conference*, pages 155–164, 1995.

[7] G.-J. Kim, S.-C. Baek, H.-S. Lee, H.-D. Lee, and M. J. Joe. LGeDBMS: a small DBMS for embedded system with flash memory. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1255–1258. VLDB Endowment, 2006.

[8] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD Conference*, pages 55–66, 2007.

[9] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In *SIGMOD Conference*, pages 1075–1086, 2008.

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

[11] D. Woodhouse. JFFS: The Journalling Flash File System. Ottawa Linux Symposium, 2001, available at: http://sources.redhat.com/jffs2/jffs2.pdf .