Understanding the Behavior of Solid State Disk

Qingchao Cai[1], Rajesh Vellore Arumugam[2], Quanqing Xu[2], and Bingsheng He[1]

[1] School of Computer Engineering, Nanyang Technological University, Singapore
{qccai,bshe}@ntu.edu.sg
[2] Data Storage Institute, A*STAR, Singapore
{Rajesh_VA,Xu_Quanqing}@dsi.a-star.edu.sg

Abstract. In this paper, we develop a family of methods to characterize the behavior of new-generation Solid State Disks (SSDs). We first study how writes are handled inside the SSD by varying request size of writes and detecting the placement of requested pages. We further examine how this SSD performs garbage collection and flushes write buffer. The result shows that the clustered pages must be written and erased simultaneously, otherwise significant storage waste will arise if such clustered pages are partially written.

We then conduct two case studies to analyze the storage efficiency when an SSD is used for server storage and the cache layer of a hybrid storage system. In the first case, we find that a moderate storage waste exists, whereas in the second case, the number of written pages caused by a write request can be as much as 4.2 times that of pages requested, implying an extremely low storage efficiency. We further demonstrate that most of such unnecessary writes can be avoided by simply delaying the issuance of internal write requests, which are generated when a read request cannot be serviced by the cache layer. We believe that this study is helpful to understand the SSD performance behavior for data-intensive applications in the big-data era.

Keywords: Storage, Solid State Disk, Hybrid storage system, Algorithm.

1 Introduction

NAND-flash based Solid state disks (SSDs)[1] have been incorporated into the computer storage architecture over the past several years, and now have become an important supplementary to traditional rotational hard disk drives (HDDs). Compared with their rotational counterparts, SSDs have a much higher read/write throughput, and due to the absence of moving mechanical components, SSDs are able to sustain an order of magnitude less random access latency.

The layout of data in SSDs is much more complicated than in HDDs. The storage space of an SSD can be partitioned into multiple domains, each containing a number of flash memory pages that share some specific resources [6]. Due

[1] We restrict our discussion to flash based SSDs, as most SSDs in the market are of this kind.

© Springer International Publishing Switzerland 2015

H. Handa et al. (eds.), *Proc. of the 18th Asia Pacific Symp. on Intell. & Evol. Systems – Vol. 1*,
Proceedings in Adaptation, Learning and Optimization 1, DOI: 10.1007/978-3-319-13359-1_27

to resource contention, an access requesting two pages within a same domain might have a longer latency than that requesting two pages placed in different domains. The difference in the internal structure and access latency between SSD and HDD can also lead to the different way in which access requests are serviced. We explore the service of access requests inside SSDs, as it can be used to reveal how SSD realizes its specific internal structure and assist the incorporation of SSD into storage systems.

In this paper, we develop a family of methods to characterize the behavior of a representative SSD. First, we carry out an investigation on how write requests are serviced inside this SSD. To this end, we issue multiple writes with varying request size to the SSD, and then detect the placement of requested pages via comparing the latencies among a set of carefully designed read requests. The result implies there exist clustered pages which must be written simultaneously, and pages for servicing write requests are chosen such that there are least number of partially written clustered pages. Second, we study how garbage collection is performed by overwriting the certain page of clustered pages that have been completely written and then measuring the resulted page placement, and find that the constituting pages of a clustered page must also be erased at same time. In addition, we extract the length of flush periods, defined as the interval between two flushes of SSD write buffer, using a method similar to that of investigating the service of write requests. The difference is that the varying parameter is no longer the request size of writes, but the interval between two consecutive write requests instead.

The characteristics of clustered pages that the four pages must be written and erased simultaneously implies there will be a waste of storage if a clustered page is partially written. In order to quantify storage efficiency, we conduct two case studies in which the SSD is used for different purposes. In the first case, we analyze the block access traces of ten server applications, and find that if the same sequence of write requests are issued to the SSD, hundreds of thousands of wasted pages, i.e., the unwritten pages of partially written clustered pages, will be produced. In the second case, we use Flashcache [24] to deploy a hybrid storage system with SSD serving as cache layer, and collect the traces of accesses to the SSD cache for multiple IO access patterns. The result shows that due to the long inter-arrival interval of internal write requests which are generated when a read request cannot be serviced by cache layer, the wasted pages can be up to 1.2 times more than those requested when reads account for the majority of IOs, which in turn leads to a write amplification up to 4.2. We show that most of such wasted pages can be eliminated by simply first delaying the issuance of internal write requests and then flushing them simultaneously. We believe that our findings can guide the design and implementation of data-intensive applications on SSDs in the big data era.

The remainder of paper is organized as follows. Section 2 describes the background and related works. The methods of capturing SSD behavior and the corresponding results are presented in Section 3 in detail. Two case studies are presented in Section 4 to quantify SSD storage efficiency. We finally conclude this work in Section 5.

2 Background and Related Works

2.1 Solid State Disk

Data is accessed in page granularity in SSDs, and in this sense a flash memory page can be viewed as a block of hard disks. The difference between them is that flash pages do not support in-place update, and can be overwritten only after being erased. The erase operation of SSDs, however, is not page-based, but in a granularity of erase blocks. An erase block is comprised of a number (usually 64 or 128) of consecutive flash pages, and as a result, each time when a block is to be erased, the valid pages in it should first be copied to the free pages of other blocks, which leads to the notorious write amplification problem of SSDs.

To hide these behavioral differences, a flash translation layer (FTL) [7] [10] [14] [17] is employed in SSDs. To support out-of-place update, FTL provides the map of logical page number to physical page number, giving an illusion of in-place update to the host. Another important function of FTL is to perform garbage collection (GC). GC erases one or more blocks when there are no sufficient free pages to service write requests or when device is idle, and generally the blocks with least valid pages are selected for GC so that write amplification is minimized. In addition, since each SSD block can only withstand a limited number of erase cycles, FTL also implements wear-leveling to evenly spread the writes to each block and hence extend SSD lifetime.

FTL is usually implemented as a firmware run by an *SSD controller*. SSD controller translates incoming read/write requests into flash memory operations and issues commands to flash memory through a flash controller. Besides the SSD controller, there are three other major components inside an SSD. The *host interface logic* connects device to the host via an interface connector such as SATA. A *RAM buffer* is also commonly deployed in SSDs to improve access performance by temporarily storing data accessed and buffering write requests. Data is persistently stored in an array of *flash memory packages* which are connected to the flash controller via multiple channels. Each flash memory package is composed of multiple dies, and each die further contains multiple planes, each with a number of flash blocks inside. The design issues of SSD architecture are discussed in detail in [4] [8].

The four-level hierarchy of flash memory corresponds to four levels of parallelism: channel-level, package-level, die-level and plane-level. Flash pages across different channels, packages or dies can be operated independently, and can thus support parallel operations over them natively. However, the plane-level parallelism is not activated in general, unless there are multiple operations of same type simultaneously accessing flash pages across different planes of the same die, in which case the plane-level parallelism can be exploited through *n-plane command* which enables n (typically 2 or 4) planes of a same die to work simultaneously. There have been many studies [11] [19] [23] toward effectively exploiting the rich parallelism inside SSDs for better IO performance. Since the parallelism of SSDs can be exploited effectively by sequential writes, some studies [15] [16] [18] tailor up-level applications to make SSD writes as sequential as possible.

2.2 The Extraction of SSD Parameters

Since SSD parameters can substantially affect the performance of device, it is thus of practical meaning to extract them as it can guide the design of systems and applications to exploit SSD performance more effectively.

While part of these parameters such as page size and block size are well documented, there also exist some implicit parameters, e.g., parallel degree and size of clustered page, hiding inside SSD internals. Chen et al. [6] probe the size of chunks, which consist of pages that are continuously allocated within a single domain, parallel degree and page mapping policy of several SSDs, and give a detailed discussion on the influence of parallelism on SSD performance.

Another work of SSD parameter extraction is [12], which develops a set of micro benchmarks to extract the size of clustered page/block and read/write buffer, and modifies Linux block layer such that the incoming reads/writes are aligned with the boundary of clustered pages and then split into pieces with the same size of read/write buffer. Although the term "clustered page" is also used in this work to represent an internal storage unit of SSD, it has a different meaning from the counterpart used in our work. By its definition in [12], a clustered page is actually composed of pages across different SSD domains, and hence closely related to the degree of parallelism inside SSD. On the contrary, the clustered page defined in our work consists of flash pages that are placed in the same domain and must be written and erased simultaneously.

3 The Measurement of SSD Parameters

3.1 Experimental Enviorment

The experiment is conducted on an HP xw6600 workstation, which is equipped with an Intel Quad Core Xeon(R) E5420 2.5GHz processor and 4GB main memory. For the OS, we use Ubuntu 12.04 with Kernel 3.2.0 and install it in in a 250GB Seagate 7200RPM hard disk. The device for measurement is a 128GB SSD produced by a mainstream SSD manufacturer. It is built upon multi-level cells (MLC) flash memories, and the 4KB random read and write latencies of this device are 33 μs and 12.5 μs, respectively. To avoid the interference from the OS (e.g., page cache and file system), we perform the measurement directly on the raw block device. Following the previous study [6], we choose *noop* as the IO scheduler for this SSD, leaving the optimization for access requests handled by the device itself. For the sake of expression, this SSD will be referred to as "SSD-A" in the following text.

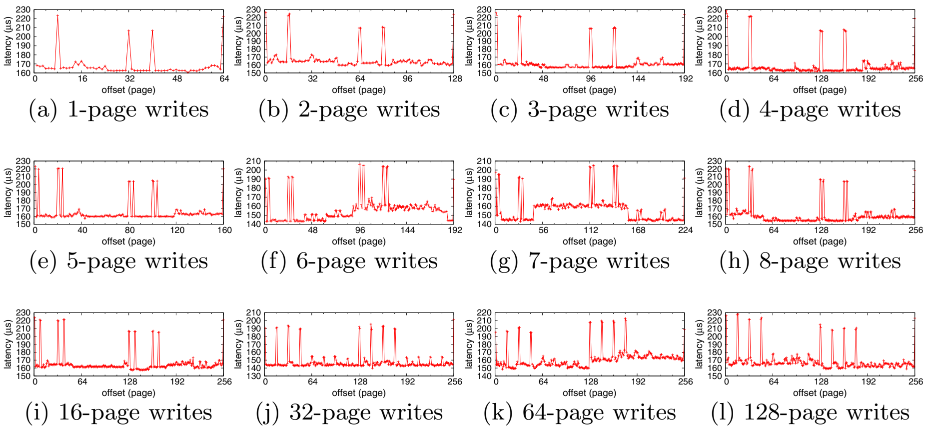
3.2 Characterizing SSD Behaviors

We adopt the generalized model presented in [6] to profile SSD internals. As described in this model, an SSD consists of multiple *domains*, each of which is a set of flash memories that share some specific resources; the pages continuously allocated within one domain comprise a *chunk*.

Table 1. Description of the parameters and functions

name	description
`cycle`	length of variation cycles of `stride_read` latency
`dev_size`	SSD device size
`max_offset`	max. offset in page unit
`max_rq_size`	max. write size in page unit
`pg_size`	page size of SSD
`range`	size of address space initialized
`read_buf_size`	SSD read buffer size
`rand_pos(pos, size)`	randomly choose an address within $[0, pos)$ aligned to `size`
`SSD_read(pos, size)`	read `size` bytes against `pos`
`SSD_write(spos, epos, size)`	sequentially fill range $[spos, epos)$ with `size`-byte write requests, during which OS page cache and SSD write buffer are both disabled
`single_write(pos)`	write one page at `pos` with SSD write buffer enabled
`stride_read(pos, offset)`	read two pages with two concurrent threads, each for one page; the 1st thread reads against `pos`, and the 2nd one skips `offset` over the 1st one

Servicing Write Requests. We intend to investigate how flash pages get written under different write patterns, thereby revealing how write requests are serviced. To this end, we first initialize several disjoint address ranges of SSD-A with writes of varying request sizes. We disable the page cache of operating system and the write buffer of SSD-A so that each write requests will be directly handled by flash memories. Since due to resource contention, the pages inside a domain will experience a longer read latency compared with those across multiple domains, we issue a set of read requests that are able to realize this difference in



(a) 1-page writes (b) 2-page writes (c) 3-page writes (d) 4-page writes

(e) 5-page writes (f) 6-page writes (g) 7-page writes (h) 8-page writes

(i) 16-page writes (j) 32-page writes (k) 64-page writes (l) 128-page writes

Fig. 1. The variation of page placement with the request size of writes

Procedure 1. Measuring SSD page placement (I)

```
for i ← 1 to max_rq_size do
    spos ← (i − 1) × range;   epos ← i × range
    rq_size ← i × pg_size
    SSD_write(spos, epos, rq_size)
    for j ← 1 to max_offset do
        latency ← 0;   offset ← j × pg_size
        for k ← 0 to 1000 do
            latency ← stride_read(spos, offset) + latency
            //polute SSD read buffer
            SSD_read(spos + max_offset × pg_size, read_buf_size);
        end
        print rq_size, offset, latency/1000
    end
end
```

read latency, and compare their service time to derive how pages are placed. The whole process is shown in Procedure 1 in detail. The parameters and functions used in this paper are summarized in Table 1.

In Procedure 1, we create two concurrent threads for each read. Each thread reads only one page. The first thread reads against the start address of current address range, and the second thread skips `offset` pages over the first one. Figure 1 shows how latency of 2-thread read varies with `offset` in several address ranges initialized with writes of different request size. Since the latency varies periodically with `offset`, we only plot the first two cycles in Fig. 1.

From Fig. 1, we can make the following derivation regarding the service of write requests inside SSD-A. *For a write with a request size of n ($n \leq 32$) pages, numbered from 1 to n, $\lceil n/4 \rceil$ domains will be used to handle this request such that domain i ($0 < i < \lceil n/4 \rceil$) holds pages $\{2i-1, 2i, 2(i+\lceil n/4 \rceil)-1, 2(i+\lceil n/4 \rceil)\}$, and domain $\lceil n/4 \rceil$ holds remainder pages.* For instance, 2 domains will be involved in the service of a write with request size of 7 pages; the first domain holds page 1, 2, 5 and 6, and the other domain stores page 3, 4 and 7. A write with larger request size can be viewed as a composition of several sub-writes, each of which requests 32 pages (the last sub-write can have a less request size), and will be handled in the same way as these sub-writes.

The above result demonstrates that SSD-A tries to write as close as possible to four pages for each domain when servicing write requests. It also implies for each incoming write request, SSD-A places the requested pages in the domains next to the last domain involved in the service of last write request, regardless of how many pages were written in this domain. This implication is also validated by the length of variation cycle of read latency. As shown in Fig. 1, the variation cycle has a length of $\frac{128 \times n/4}{\lceil n/4 \rceil}$ for the address range initialized with n-page writes.

A reasonable speculation following the way write requests are handled inside SSD-A is that the certain four pages within same domain must be written simultaneously, and if only part of these four pages have been written, the other pages can be programmed (written) only after the written ones have been erased.

Procedure 2. Measuring SSD page placement (II)

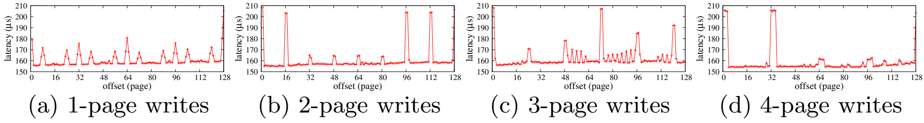
```
len ← cycle × pg_size        //cycle is measured in Procedure 1
for i ← 0 to 4 do
    SSD_write(0, device_size, 256 × pg_size)
    rq_size ← (i + 1) × pg_size
    SSD_write(0, range, rq_size)
    for j ← 1 to max_offset do
        latency ← 0;    offset ← j × pg_size
        for k ← 0 to 50000 do
            pos ← rand_pos(range, len) + rand_pos(len/4, rq_size)
            latency ← stride_read(pos, offset) + latency
        end
        print rq_size, offset, latency/50000
    end
end
```

To verify the above speculation, we carry out another experiment in a similar way to Procedure 1. We first sequentially fill the whole address space of SSD-A with writes of a large request size so that the placement of pages is the same as that shown in Fig. 1j-1l. Then, starting from address 0, we sequentially write SSD-A with a request size of one page, during which the OS page cache and on-device write buffer are disabled.

We use the same method as Procedure 1 to detect the placement of pages within the address range filled in the second write phase. This time we allow the first thread to read addresses other than the start position of address range, i.e., address 0. Specifically, each time the first thread reads against an address randomly selected from the address set $\{i \times \texttt{cycle} \times \texttt{pg_size} + j \times \texttt{rq_size} | 0 \leq i < \frac{\texttt{range}}{\texttt{cycle} \times \texttt{pg_size}}, 0 \leq j < 8\}$, where `cycle` is the cycle length measured in Procedure 1 (32 in this case, as shown in Fig. 1a), and the meanings of other variables can be found in Table 1. We do this because, as can be inferred from Fig. 1, the latency of 2-thread reads keeps almost unchanged when the first thread reads against different addresses of this set. The detailed implementation is shown in Procedure 2.

We repeat this experiment for four times. Each time we choose a different write size in the second write phase, and adjust the candidate address set for the first read thread accordingly. The experiment result is shown in Fig. 2.

Comparing Fig. 2 with Fig. 1a - 1d, it is intuitive to observe that when write request size of the second write phase is less than four pages, there will be a substantial difference in page placement between the two scenarios with/without the first write phase, and such difference disappears when the request size of second-phase writes increases to four pages. Therefore, we can infer that pages written in the second phases of the first three runs (corresponding to Fig. 2a, 2b and 2c, respectively) have been relocated and compacted to provide more available flash pages, otherwise the corresponding page placement should keep unchanged as writes of seconde phase are gradually serviced. This result also confirms our speculation made above: the certain four pages within same domain



(a) 1-page writes (b) 2-page writes (c) 3-page writes (d) 4-page writes

Fig. 2. The variation of page placement with the size of write request and the availability of free space

must be programmed simultaneously; if only part of these four pages have been written, the other pages are left unable to service write requests until the written ones have been erased, leading to a waste in storage. For the sake of expression, we call each such four pages *a clustered page*.

From the definition of clustered pages, it is not difficult to see that storage waste will raise from the service of writes with a request size that is not a multiple of four pages, since at least one involved clustered page will be partially written. Such feature of clustered pages helps to understand our result regarding the service of write requests. As we have mentioned, a write requesting n pages will be handled such that page $2i-1$, $2i$, $2(i+\lceil n/4 \rceil)-1$ and $2(i+\lceil n/4 \rceil)$ requested in this write will be placed in the same domain and occupy a full clustered page. In this way, for each write request, there is at most one clustered page, i.e., the last one involved, that might will be partially written, and the storage waste is thus minimized. In the mean time, the parallelism among domains can be effectively exploited.

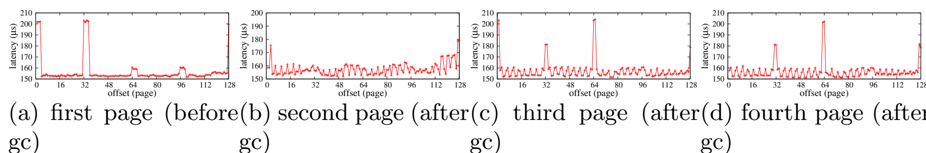
Garbage Collection. The SSD erase granularity has been studied in [12] based on the assumption that after the whole SSD has been sequentially written, the speed of following random writes with a request size equal to the size of erase unit must be same as that of sequential writes, as there is no page relocation in both cases. In this work, we are more interested in whether the blocks with pages in same clustered pages must be erased simultaneously or not, for which a more intuitive result can be obtained by carefully overwriting the clustered pages that have been completely filled.

As shown in Procedure 3, we first sequentially fill the whole space of SSD-A with writes of a request size of four pages so that each involved clustered page is fully written and page placement inside SSD-A is same as in Fig. 2d. After that, from address 0, we gradually overwrite the first page of clustered pages. As write process progresses, garbage collection will be revoked to reclaim the overwritten pages, and we are then able to answer the question concerned by examining whether the non-overwritten pages have been relocated.

Figure 3b - 3d present the placement of non-overwritten pages after garbage collection, and their counterpart before garbage collection is shown in Fig. 3a for comparison purpose. It can be easily derived from Fig. 3 that the three pages of clustered pages that did not get overwritten during the write process have been relocated after garbage collection. We are thus able to conclude that the four

Procedure 3. Measuring page placement after garbage collection

```
chunk_size ← 4 × pg_size
SSD_write(0, device_size, chunk_size)
for i ← 0 to range/chunk_size do
    single_write(i × chunk_size)
end
len ← cycle × pg_size
for i ← 1 to 4 do
    for j ← 0 to max_offset do
        latency ← 0;    offset ← j × pg_size
        for k ← 0 to 50000 do
            /* each time the first thread reads the (i + 1)-th page of a clustered
            page                                                                  */
            pos ← rand_pos(range, len) + rand_pos(len/4, chunk_size) +
            i × pg_size
            latency ← stride_read(pos, offset) + latency
        end
        print i, j, latency/50000
    end
end
```



(a) first page (before gc) (b) second page (after gc) (c) third page (after gc) (d) fourth page (after gc)

Fig. 3. Page placement inside clustered pges before/after garbage collection

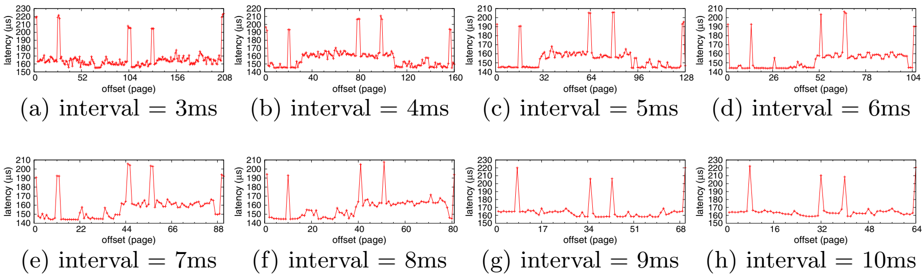
pages of each clustered page must be erased simultaneously, and a write with a request size of n pages thus actually leads to $4 \times \lceil n/4 \rceil$ pages being written in the sense that there is no difference between a written page and an unwritten one within the same clustered page, both of which cannot be programmed until being erased. As a result, the write amplification resulted from a write requesting n pages is $4 \times \lceil n/4 \rceil /n$, and this number may be further increased by 1 in the case where there are many partially written clustered pages that can be relocated and compacted to generate a large number of free pages.

Flushing Write Buffer. The problem of partially written clustered pages can be alleviated by the existence of write buffer in most SSDs. When a write request arrives, the SSD first buffers it and later flushes all the buffered write requests to the flash memory for persistent storage. As such, the number of pages written each time is increased, reducing the ratio of partially written clustered pages.

The flush of write buffer will be triggered when the buffer is full, or after a certain time period, which we call *flush period*. The corresponding two

parameters associated with SSD write buffer are thus the size of write buffer and the length of flush period. As the measurement of the former parameter has been conducted in [12], we are more interested in the latter parameter.

We have revealed how request size of writes affects page placement for the case with write buffer disabled. Conversely, we can also infer from observed page placement the request size of writes and the number of buffered pages for flush, both of which are in principle the same. In this regard, we follow the same way as we did in Procedure 1 with the exception that the parameter varying across address ranges is no longer the request size of writes, which is kept constant at 1 page, but the interval between two consecutive writes instead. In addition, the write buffer is no longer disabled during the initialization phase. Due to page limit and its similarity to Procedure 1, the detailed implementation is not presented in this paper.



(a) interval = 3ms (b) interval = 4ms (c) interval = 5ms (d) interval = 6ms

(e) interval = 7ms (f) interval = 8ms (g) interval = 9ms (h) interval = 10ms

Fig. 4. The variation of page placement with the interval between two consecutive writes

Figure 4 gives the measured latency of two-thread reads for different inter-arrival interval of write requests. For demonstration purpose, we only provide in this figure the result for cases in which the buffered pages for each flush occupy only one clustered page. Comparing Fig. 4 with Fig. 1, we can find that Fig. 4h and 4c are, respectively, the same as Fig. 1a and 1b, which means the numbers of buffered pages for flush in corresponding two cases are 1 and 2, respectively, and the length of flush period is thus 10 milliseconds. Moreover, as can be observed from Fig. 4, the read latency in each case exhibits a periodical pattern with a cycle length reversely proportional to the inter-arrival interval. It can be inferred from this observation that the flush of write buffer is performed once every 10 milliseconds, rather than in 10 milliseconds after the arrival of the earliest buffered write, in which case there would be two buffered pages for each flush if the inter-arrival interval of writes is within the range $(5ms, 10ms)$, and Fig. 4d-4g thus must have the same cycle length as Fig. 4c.

4 SSD Storage Efficiency

4.1 Server Storage

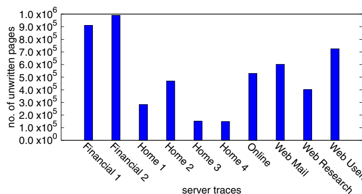
We first study the case in which SSD-A is used as storage device for server applications. To this end, we download the HDD block access traces of two OLTP applications (Financial 1 and Financial 2) [3] and eight other server applications [1] [25]. Table 2 gives the general information of these block traces.

Table 2. Description of server traces

name	no. WR records	name	no. WR records
Financial 1	4,099,354	Home 4	2,354,032
Financial 2	653,082	Online	4,211,728
Home 1	8,882,821	Web Mail	6,381,984
Home 2	4,901,076	Web Research	2,413,936
Home 3	908,835	Web Users	5,127,100

For each trace, we intend to investigate the storage efficiency when the same sequence of writes are issued to SSD-A. To simplify the investigation, we assume that two consecutive writes with an interval less than a certain threshold will be flushed simultaneously; the threshold is chosen to be longer (20ms in our investigation) than the time to access data in most modern HDDs so that two consecutive writes with an interval longer than the threshold are likely to be independent, and thus will be issued with the same interval in SSD case. Under this assumption, the number of buffered pages for flush will be an over-estimation to the real value as pages for flush are those that have been buffered over a time period longer than the flush period.

Figure 5 presents the number of wasted pages, i.e., unwritten flash pages in clustered pages, that will be incurred if the same sequence of writes of each trace are issued to SSD-A. It can be seen from this figure that each trace will generate hundreds of thousands of wasted pages, which means a moderate degree of storage waste, as compared with the number of write records in Table 2.



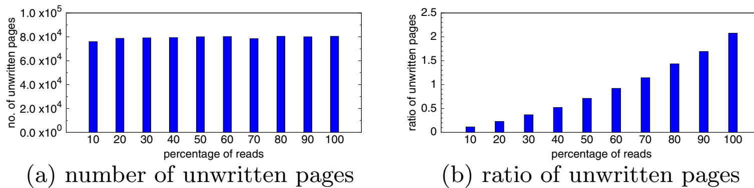
(a) number of unwritten pages

Fig. 5. Storage waste in different server traces

4.2 Hybrid SSD/HDD Storage System

Due to their superior access performance but relatively high cost, SSDs have been extensively used as an additional cache layer on the top of HDDs to form hybrid SSD/HDD storage systems with improved storage performance [9] [13] [21] [22]. In such systems, if a read request cannot be serviced by cache layer, it will generate a write request to the cache, and the inter-arrival interval of such write requests, which we will call *internal write requests* in the sense that they are issued inside the storage system, is thus no less than the service time of read requests of hard disks, which can be as much as tens of milliseconds due to high position delay [2]. As a result, if SSD-A is employed as the cache layer, the internal write requests can lead to a significant waste of cache storage because of their slow arrival rate and the existence of clustered pages inside cache.

We carry out several experiments to investigate the storage efficiency when SSD-A is used as the cache layer of hybrid storage systems. The experimental platform is Flashcache [24], a popular open source solution to hybrid SSD/HDD systems, and the tool for I/O test is fio [5]. For experiment, we issue a set of random reads and writes, each requesting one page, to storage system, and explore the resulted storage waste. We run the experiment five times, each lasting five minutes and with a varied fraction of reads. The page cache of operating system and SSD write buffer are both enabled during the experiment.



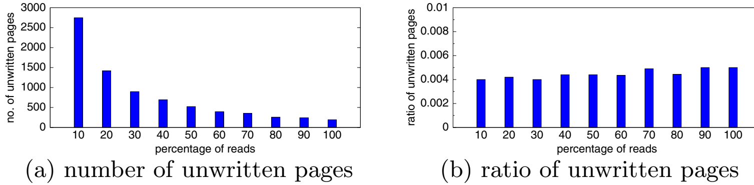
(a) number of unwritten pages (b) ratio of unwritten pages

Fig. 6. Storage waste in hybrid storage system

The result of storage efficiency is demonstrated in Fig. 6. Since the OS page cache is enabled, writes are first handled in the memory and thus completed much faster than reads. Consequently, there are roughly the same number of reads and internal writes across all scenarios. In addition, writes buffered in memory are issued to storage system in a batch mode, and thus incur little partially written clustered pages. Therefore, wasted pages are mostly caused by internal writes, and thus of roughly the same number across all scenarios, which is verified in Fig. 6a.

Fig. 6b describes the ratio of unwritten pages to pages requested (including those involved in internal write requests) in different scenarios. This figure also shows there is a serious storage waste when reads account for a large fraction of total IOs. For instance, in the scenario with all IOs being reads, each write request leads to 2.2 wasted pages on average, implying a write amplification up to 4.2, which will be achieved after page relocation.

When OS page cache is enabled, the data read from storage device will be stored in memory, i.e., page cache, so that future reads requesting the same data can be satisfied without disk access. Therefore, for hybrid storage systems, the issuance of internal write requests can be safely delayed without performance loss, as long as there is a copy of the corresponding data in memory. Consequently, we can suspend the issuance of internal write requests until there are a certain number of such write requests accumulated, and then simultaneously issue them to the cache layer to improve storage efficiency.



(a) number of unwritten pages (b) ratio of unwritten pages

Fig. 7. Storage waste in hybrid storage system with the isssue of internal write requests delayed

We implement the above method in Flashcache [24]. In our implementation, the issuance of delayed internal write requests takes place when a certain time period has passed by after last issuance, or the number of delayed requests exceeds a specific threshold. We repeat the above experiments and show the corresponding results in Fig. 7. We can draw from this figure that there will be little storage waste when the issuance of internal write requests is delayed: in all cases, the number of wasted pages is no more than 0.5% that of pages requested. In addition, it is worth noting that this method of reducing cache writes is orthogonal to those presented in [21] which reduce the writes to cache by neglecting the data that have been requested only a limited times.

Besides improving storage efficiency, there are some other advantages that can arise from delaying the issuance of internal write requests. First, issuing multiple writes requests at same time can effectively exploit the rich parallelism inside SSD. In addition, read/write interference inside SSD, which has been reported to be able to significantly hamper access performance of SSDs [6][20], can also be alleviated as a result of reduced number of flushes of SSD write buffer.

5 Conclusion

In this paper, we carry out an extensive investigation on the behavior of new-generation SSDs, and obtain two major findings. First, the investigation exposes the existence of clustered pages, each of which consists of certain four flash pages that must be programmed and reclaimed simultaneously. Second, pages are placed inside the SSD such that the parallelism of the SSD can be effectively exploited, on the premise that the number of partially written clustered pages, which are the source of storage waste, is minimized.

In order to quantify the impact of clustered pages on storage efficiency, we then conduct two case studies, i.e., server storage and cache layer of a hybrid storage system. For the latter case, we find that when most IOs are reads, the storage efficiency is extremely low due to the long inter-arrival interval of internal write requests which are generated when a read cannot be serviced by cache layer. By delaying the issuance of such internal write requests, we can substantially reduce the number of wasted pages, thereby enhancing the storage efficiency and extending the lifetime of SSD.

Acknowledgement. This work is partially supported by the ASTAR Thematic Strategic Research Programme (TSRP) Grant No. 1121720013 and the Center for Computational Intelligence at Nanyang Technological University.

References

1. FIU Traces, http://iotta.snia.org/traces/390 (retrieved September 11, 2014)
2. Hard disk drive, http://en.wikipedia.org/wiki/Hard_disk_drive (retrieved September 11, 2014)
3. UMassTraceRepository, http://traces.cs.umass.edu/index.php/Storage/Storage
4. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M.S., Panigrahy, R.: Design tradeoffs for ssd performance. In: ATC, Boston, Massachussetts, USA, pp. 57–70 (2008)
5. Axboe, J.: fio, https://github.com/axboe/fio (retrieved September 11, 2014)
6. Chen, F., Lee, R., Zhang, X.: Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In: HPCA, San Antonio, Texas, USA, pp. 266–277 (2011)
7. Chen, F., Luo, T., Zhang, X.: Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In: FAST, San Jose, California, USA (2011)
8. Dirik, C., Jacob, B.: The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In: ISCA, Austin, TX, USA, pp. 279–289 (2009)
9. Guerra, J., Pucha, H., Glider, J., Belluomini, W., Rangaswami, R.: Cost effective storage using extent based dynamic tiering. In: FAST, San Jose, CA, USA (2011)
10. Gupta, A., Kim, Y., Urgaonkar, B.: Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In: ASPLOS XIV, Washington, DC, USA, pp. 229–240 (2009)
11. He, B., Yu, J.X., Zhou, A.C.: Improving update-intensive workloads on flash disks through exploiting multi-chip parallelism. IEEE Transactions on Parallel and Distributed Systems (2014)
12. Kim, J., Seo, S., Jung, D., Kim, J.S., Huh, J.: Parameter-aware i/o management for solid state disks (ssds). IEEE Transactions on Computers 61(5), 636–649 (2012)
13. Koltsidas, I., Viglas, S.D.: Flashing up the storage layer. Proceedings of the VLDB Endowment 1(1), 514–525 (2008)
14. Lee, S.W., Park, D.J., Chung, T.S., Lee, D.H., Park, S., Song, H.J.: A log buffer-based flash translation layer using fully-associative sector translation. ACM Transactions on Embedded Computing Systems 6(3), article No. 18 (2007)

15. Li, Y., He, B., Luo, Q., Yi, K.: Tree indexing on flash disks. In: ICDE, Shanghai, China, pp. 1303–1306 (2009)
16. Li, Y., He, B., Yang, R.J., Luo, Q., Yi, K.: Tree indexing on solid state drives. Proceedings of the VLDB Endowment 3(1-2), 1195–1206 (2010)
17. Ma, D., Feng, J., Li, G.: Lazyftl: A page-level flash translation layer optimized for nand flash memory. In: SIGMOD, Athens, Greece (2011)
18. Min, C., Kim, K., Cho, H., Lee, S.W., Eom, Y.I.: Sfs: Random write considered harmful in solid state drives. In: FAST, San Jose, CA, USA (2012)
19. Park, C., Seo, E., Shin, J.Y., Maeng, S., Lee, J.: Exploiting internal parallelism of flash-based ssds. Computer Architecture Letters 9(1), 9–12 (2010)
20. Park, S., Shen, K.: Fios: a fair, efficient flash i/o scheduler. In: FAST, San Jose, CA, USA (2012)
21. Pritchett, T., Thottethodi, M.: Sievestore: A highly-selective, ensemble-level disk cache for cost-performance. In: ISCA, Saint-Malo, France, pp. 163–174 (2010)
22. Saxena, M., Swift, M.M., Zhang, Y.: Flashtier: A lightweight, consistent and durable storage cache. In: EuroSys, Bern, Switzerland, pp. 267–280 (2012)
23. Seol, J., Shim, H., Kim, J., Maeng, S.: A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks. In: CASE, Grenoble, France, pp. 137–146 (2009)
24. Srinivasan, M.: Flashcache, `https://github.com/facebook/flashcache` (retrieved September 11, 2014)
25. Verma, A., Koller, R., Useche, L., Rangaswami, R.: Srcmap: energy proportional storage using dynamic consolidation. In: FAST, San Jose, California, USA (2010)