Extending SSD Lifetimes with Disk-Based Write Caches

Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, Ted Wobber*
University of Toronto, Microsoft Research Silicon Valley*
gokul@eecg.toronto.edu, {vijayanp, maheshba, wobber}@microsoft.com

Abstract

We present Griffin, a hybrid storage device that uses a hard disk drive (HDD) as a write cache for a Solid State Device (SSD). Griffin is motivated by two observations: First, HDDs can match the sequential write bandwidth of mid-range SSDs. Second, both server and desktop workloads contain a significant fraction of block overwrites. By maintaining a log-structured HDD cache and migrating cached data periodically, Griffin reduces writes to the SSD while retaining its excellent performance. We evaluate Griffin using a variety of I/O traces from Windows systems and show that it extends SSD lifetime by a factor of two and reduces average I/O latency by 56%.

1 Introduction

Over the past decade, the use of flash memory has evolved from specialized applications in hand-held devices to primary system storage in general-purpose computers. Flash-based Solid State Devices (SSDs) provide 1000s of low-latency IOPS and can potentially eliminate I/O bottlenecks in current systems. The cost of commodity flash – often cited as the primary barrier to SSD deployment [22] – has dropped significantly in the recent past, creating the possibility for widespread replacement of disk drives by SSDs.

However, two trends have a potential to derail the adoption of SSDs. First, general-purpose (OS) workloads are harder on the storage subsystem than hand-held applications, particularly in terms of write volume and non-sequentiality. Second, as the cost of NAND flash has declined with increased bit density, the number of erase cycles (and hence write operations) a flash cell can tolerate has suffered. This combination of a more stressful workload and fewer available erase cycles reduces useful lifetime, in some cases to less than one year.

In this paper, we propose Griffin, a hybrid storage design that, somewhat contrary to intuition, uses a hard disk drive to cache writes to an SSD. Writes to Griffin are logged sequentially to the HDD write cache and later migrated to the SSD. Reads are usually served from the SSD and occasionally from the slower HDD. Griffin's goal is to minimize the writes sent to the SSD without significantly impacting its read performance; by doing so, it conserves erase cycles and extends SSD lifetime.

Griffin's hybrid design is based on two characteristics observed in block-level traces collected from systems running Microsoft Windows. First, many of the writes seen by block devices are in fact *overwrites* of a small set of popular blocks. Using an HDD as a write cache to coalesce overwrites can reduce the write traffic to the SSD significantly; for the desktop and server traces we examined, it does by an average of 52%. Second, once data is written to a block device, it is not read again from the device immediately; the file system cache serves any immediate reads without accessing the device. Accordingly, Griffin has a time window within which to coalesce overwrites on the HDD, during which few reads occur.

A log structured HDD makes for an unconventional write cache: writes are fast whereas random reads are slow and can affect the logging bandwidth. By logging writes to the HDD, Griffin takes advantage of the fact that a commodity SATA disk drive delivers over 80 MB/s of sequential write bandwidth, allowing it to keep up with mid-range SSDs. In addition, hard disks offer massive capacity, allowing Griffin to log writes for long periods without running out of space. Since hard disks are very inexpensive, the cost of the write cache is a fraction of the SSD cost.

We evaluate Griffin using a simulator and a user-level implementation with a variety of I/O traces, both from desktop and server environments. Our evaluation shows that, for the desktop workloads we studied, our caching policies can cut down writes to the SSD by approximately 49% on average, with less than 1% of reads serviced by the slower HDD. For server workloads, the observed benefit is more widely varied, but equally signifi-

cant. In addition, Griffin improves the sequentiality of the write accesses to the SSD by an average of 15%, which can indirectly improve the lifetime of the SSD. Reducing the volume of writes by half allows Griffin to extend SSD lifetime by at least a factor of two; by additionally improving the sequentiality of the workload seen by the SSD, Griffin can extend SSD lifetime even more, depending on the SSD firmware design. An evaluation of the performance of Griffin shows that it performs much better than a regular SSD, where the average I/O latency is reduced by 56%.

2 SSD Write-Lifetime

Constraints on the amount of data that can be written to an SSD stem from the properties of NAND flash. Specifically, a block must be erased before being re-written, and only a finite number of erasures are possible before the bit error rate of the device becomes unacceptably high [7, 20]. SLC (single-level cell) flash typically supports 100K erasures per flash block. However, as SSD technology moves towards MLC (multi-level cell) flash that provides higher bit densities at lower cost, the erasure limit per block drops as low as 5,000 to 10,000 cycles. Given that smaller chip feature sizes and more bits-per-cell both increase the likelihood of errors, we can expect erasure limits to drop further as densities increase.

Accordingly, we define a device *write-lifetime*, which is the total number of writes that can be issued to the device over its lifetime. For example, an SSD with 60 GB of NAND flash with 5000 erase-cycles per block might support a maximum write-lifetime of 300 TB (5000 $\times$ 60 GB). However, write-lifetime is unlikely to be optimal in practice, depending on the workload and firmware. For example, according to Micron's data sheet [18], under a specific workload, its 60 GB SSD only has write-lifetime of 42 TB, which is a reduction in write-lifetime by a factor of 7. It is conceivable that under a more stressful workload, SSD write-lifetime decreases by more than an order of magnitude.

Firmware on commodity SSDs can reduce write-lifetime due to inefficiencies in the Flash Translation Layer (FTL), which maintains a map between host logical sector addresses and physical flash addresses [14]. The FTL chooses where to place each incoming logical sector during a write. If the candidate physical block is occupied with other data, it must be moved and the block must be erased. The FTL then writes the new data and adjusts the map to reflect the position of the new data. While sequential write patterns are easy to handle, non-sequential write patterns can be problematical for the FTL by requiring data copying in order to free up space for each incoming write. In the absolute worst case of continuous 512 byte writes to random addresses, it may be necessary to move a full MLC flash block (512 KB) less 512 bytes for each incoming write, reducing write-lifetime by a factor of 1000. The effect is usually known as *write-amplification* [10] to which we must also add the cost of maintaining even wear across all blocks. Although the worst-case workload is not likely, and the FTL can lessen the negative impact of a non-sequential write workload by maintaining a pool of reserve blocks not included in the drive's advertised capacity, non-sequential workloads will always trigger more erasures than sequential ones.

It is not straightforward to map between reduced write workload and increased write-lifetime. Halving the number of writes will at least double the lifetime. However, the effect can be greater to the extent it also reduces write-amplification. Overwrites are non-sequential by nature. So if overwrites can be eliminated, or out-of-order writes made sequential, there will be both fewer writes and less write-amplification. As explored by Agrawal *et al.* [1], FTL firmware can differ wildly in its ability to handle non-sequential writes. A simple FTL that maps logical sector addresses to physical flash at the granularity of a flash block will suffer huge write-amplification from a non-sequential workload, and therefore will benefit greatly from fewer of such writes. The effect will be more subtle for an advanced FTL that does the mapping at a finer granularity. However, improved sequentiality will reduce internal fragmentation within flash blocks, and therefore will both improve wear-leveling performance and reduce write-amplification.

Write-lifetime depends on the performance of wear-leveling and the write-amplification for a given workload, both of which cannot be measured. However, we can obtain a rough estimate of write-amplification by observing the performance difference between a given workload and a purely sequential one; the degree of observed slowdown should give us some idea of the effective write-amplification. The product manual for the Intel X25-M MLC SSD [13] indicates that this SSD suffers at least a factor of 6 reduction in performance when a random-write workload is compared to a sequential one (sequential write bandwidth of 70 MB/s versus 3.3 K IOPS for random 4 KB writes). Thus, after wear-leveling and other factors are considered, it becomes plausible that practical write-lifetimes, even for advanced FTLs, can be an order of magnitude worse than the optimum.

3 Overview of Griffin

Griffin's design is very simple: it uses a hard disk as a persistent write cache for an MLC-based SSD. All writes are appended to a log stored on the HDD and eventually migrated to the SSD, preferably before subsequent reads. Structuring the write cache as a log allows Grif-

fin to operate the HDD at its fast sequential write mode. In addition to coalescing overwrites, the write cache also increases the sequentiality of the workload observed by the SSD; as described in the previous section, this results in increased write-lifetime.

Since cost is the single biggest barrier to SSD deployment [22], we focus on write caching for cheaper MLC-based SSDs, for which low write-lifetime is a significant constraint. MLC devices are excellent candidates for HDD-based write caching since their sequential write bandwidth is typically equal to that of commodity HDDs, at 70-80 MB/s [13].

Griffin increases the write-lifetime of an MLC-based SSD without increasing total cost significantly; as of this writing, the cost of a 350 GB SATA HDD is around 50 USD, whereas an 128 GB MLC-based SSD is around 300 USD. In comparison, a 128 GB SLC-based SSD, which offers higher write-lifetime than the MLC variant currently costs around 4 to 5 times as much.

Griffin also increases write-lifetime without substantially altering the reliability characteristics of the MLC device. While the HDD write cache represents an additional point of failure, any such event leaves the file system intact on the SSD and only results in the loss of recent data. We discuss failure handling in Section 5.3.

3.1 Other Hybrid Designs

Other hybrid designs using various combinations of RAM, non-volatile RAM, and rotating media are clearly possible. Since a thorough comparative analysis of all the options is beyond the scope of this paper, we briefly describe a few other designs and compare them qualitatively with Griffin.

• **NVRAM as read cache for HDD storage**: Given its excellent random read performance, NVRAM (*e.g.*, an SSD) can work well as a read cache in front of a larger HDD [17, 19, 24]. However, a smaller NVRAM is likely to provide only incremental performance benefits as compared to an OS-based file cache in RAM, whereas a larger NVRAM cache is both costly and subject to wear as the cache contents change. Any design that uses rotating media for primary storage will scale-up in capacity with less cost than Griffin. However, this cost difference is likely to decline as flash memory densities increase.

• **NVRAM as write cache for SSD storage**: The Griffin design can accommodate NVRAM as a write cache in lieu of HDD. The effectiveness of using NVRAM depends on two factors: 1) whether SLC or MLC flash is used; and, 2) the ratio of reads that hit the write cache and thus disrupt sequential logging there. The use of NVRAM can also lead to better power savings. However, all these benefits come at a higher cost than Griffin configured with a HDD cache, especially if SLC flash is used for write caching. Later, we evaluate the Griffin's performance with both SLC and MLC write caches (Section 6.4) and explore the minimum write cache size required (Section 7).

• **RAM as write cache for SSD storage**: RAM can make for a fast and effective write cache, however the overriding problem with RAM is that it is not persistent (absent some power-continuity arrangements). Increasing the RAM size or the timer interval for periodic flushes may reduce the number of writes to storage but only at the cost of a larger window of vulnerability during which a power failure or crash could result in lost updates. Moreover, a RAM-based write cache may not be effective for all workloads; for example, we later show that for certain workloads (Section 6.1.2), over 1 hour of caching is required to derive better write savings; volatile caching is not suitable for such long durations.

3.2 Understanding Griffin Performance

The key challenge faced by Griffin is to increase the write-lifetime of the SSD while retaining its performance on reads. Write caching is a well-known technique for buffering repeated writes to a set of blocks. However, Griffin departs significantly from conventional caching designs, which typically use small, fast, and expensive media (such as volatile RAM or non-volatile battery-backed RAM) to cache writes against larger and slower backing stores. Griffin's HDD write cache is both inexpensive and persistent and can in fact be *larger* than the backing SSD; accordingly, the flushing of dirty data from the write cache to the SSD is not driven by either capacity constraints or synchronous writes.

However, Griffin's HDD write cache is also *slower* than the backing SSD for read operations, which translate into high latency random I/Os on the HDD's log. In addition, reads can disrupt the sequential stream of writes received by the HDD, reducing its logging bandwidth by an order of magnitude. As a result, dirty data has to be flushed to the SSD before it is read again, in order to avoid expensive reads from the HDD.

Griffin's performance is thus determined by competing imperatives — data must be held in the HDD to buffer overwrites, and data must be flushed from the HDD to prevent expensive reads. We quantify these with the following two metrics:

• **Write Savings**: This is the percentage of total writes that is prevented from reaching the SSD. For example, if the hybrid device receives 60M writes and the SSD receives 45M of them, the write savings is 25%. Ideally, we want the write savings to be as high as possible.

• **Read Penalty**: This is the percentage of total reads serviced by the HDD write cache. For example, if the hybrid device receives 50M reads and the HDD receives

1M of these reads, the read penalty is 2%. Ideally, we want the read penalty to be as low as possible.

There will be no read penalty if an oracle informs Griffin in advance of data to be read; all such blocks can be flushed to the SSD before an impending read. With no read penalty, the maximum write savings possible is workload-dependent and is essentially a measure of the frequency of consecutive overwrites without intervening reads. In the worst case, there will be no write savings if there are no overwrites, *i.e.*, no block is ever written consecutively without an intervening read. An idealized HDD write cache achieves the maximum write savings with no read penalty for any workload.

To understand the performance of an idealized HDD write cache, consider the following sequence of writes and reads to a particular block: $WWWRWW$. Without a write cache, this sequence results in one read and five writes to the SSD. An idealized HDD write cache would coalesce consecutive writes and flush data to the SSD immediately before each read, resulting in a sequence of operations to the SSD that contains two writes and one read: WRW. Accordingly, the maximum write savings in this simple example is $3/5$ or 60%.

Griffin attempts to achieve the performance of an idealized HDD write cache by controlling policy along two dimensions: *what data to cache*, and *how long to cache it for*. The choice of policy in each case is informed by the characteristics of real workloads, which we will examine in the next section. Using these different policies, Griffin is able to achieve different points in the trade-off curve between read penalty and write savings.

4 Trace Analysis

In this section, we explore the benefits of HDD-based write caching by analyzing traces from desktop and server environments. Our analysis has two aspects. First, we show that an idealized HDD-based write cache can provide significant write savings for these traces; in other words, overwrites commonly occur in real-world workloads. Second, we look for spatial and temporal patterns in these overwrites that can help determine Griffin's caching policies.

4.1 Description of Traces

Our desktop I/O traces are collected from desktops and laptops running Windows Vista, which were instrumented using the Windows Performance Analyzer. Although we analyzed several desktop traces, we limit our presentation to 12 traces from three desktops due to space limitations.

Most of our server traces are from a previous study by Narayanan *et al.* [21]. These traces were collected from 36 different volumes from 13 servers running Windows Server 2003 SP2. Out of 36 different traces, we used only the most write-heavy data volume traces that have at least one write for every two reads, and have more than 100,000 writes in total (read-intensive workloads already work well on SSDs and do not require write caching). In addition, we also used a Microsoft Exchange server trace, which was collected from a RAID controller managing a terabyte of data.

Trace	Time (hr)	Number of 4 KB I/Os	Read (%)	Write (%)	Max Write Savings (%)	Overwrites in top 1% (%)	Reads in top 1%
D-1A	114	14 M	43	57	46	87	4
D-1B	70	29 M	45	55	39	87	2
D-1C	153	36 M	50	50	52	88	2
D-1D	27	07 M	40	60	64	84	1
D-2A	99	39 M	49	51	39	71	3
D-2B	105	30 M	48	52	36	63	2
D-2C	149	17 M	44	56	58	52	2
D-2D	103	22 M	56	44	52	47	1
D-3A	52	13 M	56	44	43	68	2
D-3B	105	33 M	50	50	56	72	4
D-3C	96	37 M	52	48	47	77	6
D-3D	55	16 M	51	49	51	78	4
S-EXCH	0.25	209 K	59	41	42	34	0
S-PRXY1	167	543 M	65	35	57	99	63
S-SRC10	168	408 M	47	53	14	11	2
S-SRC22	176	16 M	37	63	47	8	2
S-STG1	168	23 M	93	7	93	41	0
S-WDEV2	166	369 K	1	99	94	10	0

Table 1: **Windows Traces.**

Table 1 lists the traces we used for the analysis, where the desktop traces are prefixed by a "D" and server traces by an "S". D-1, D-2, and D-3 represent the three desktops that were traced. EXCH, PRXY1, SRC10/22, STG1, and WDEV2 correspond to traces from a Microsoft Exchange server, firewall or web proxy, source control, web staging, and a test web server. For each trace, the columns 2-5 show the total tracing time, number of I/Os, and read-write percentage.

All the traces contain block-level reads and writes below the NTFS file system cache. Each I/O event specifies the time stamp (in ms), disk number, logical sector number, number of sectors transferred, and type of I/O. Even though the desktop traces contain file system level information such as which file or directory a block access belongs to, the server traces do not have them.

4.2 Ideal Write Savings

Our first objective in the trace analysis is to answer the following question: *do desktop and server I/O traffic have enough overwrites to coalesce and if so, what are the maximum write savings provided by an idealized*

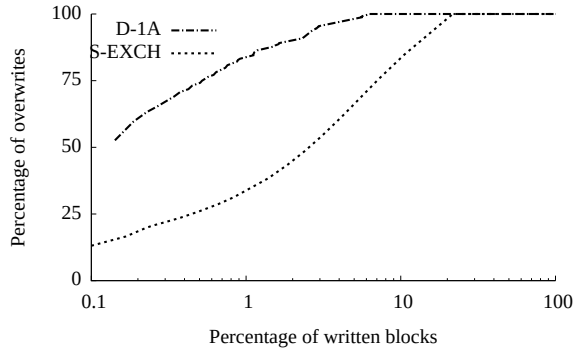


Figure 1: **Distribution of Block Overwrites.**

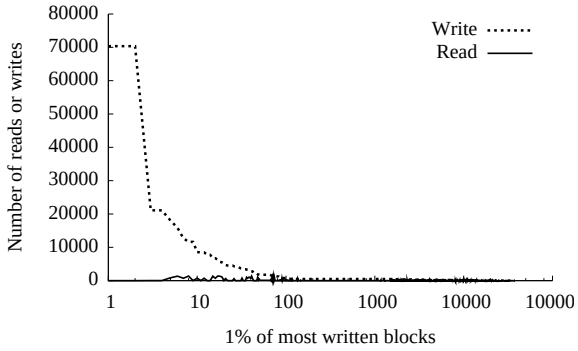


Figure 2: **Reads in Write-Heavy Blocks.**

Rank	Filenames
1	C:\Outlook.ost
2	C:\...\Search\...\Windows.edb
3	C:\$Bitmap
4	C:\Windows\Prefetch\Layout.ini
5	C:\Users\<name>\NTUSER.DAT
6	C:\$Mft

Table 2: **Top Overwritten Files in Desktops.**

HDD write cache? The 6th (highlighted) column in the Table 1 shows the maximum write savings achieved by an idealized write cache that incurs no read penalty.

From the 6th column of Table 1, we observe that an idealized HDD write cache can cut down writes to the SSD significantly. For example, for desktop traces, the maximum write saving is at least 36% (for D-2B) and as much as 64% (for D-1D). The server workloads exhibit similar savings; ideal write savings vary from 14% (S-SRC10) to 94% (S-WDEV2). On an average, desktop and server traces offer write savings of 48.58% and 57.83% respectively. Based on this analysis, the first observation we make is: *desktop and server workloads contain a high degree of overwrites*, and an idealized HDD write cache with no read penalty can achieve significant write savings on them.

Given that an idealized HDD-based write cache has high potential benefits, our next step is to explore the two important policy issues in designing a practical write cache: what do we cache, and how long do we cache it? We investigate these questions in the following sections.

4.3 Spatial Access Patterns

If block overwrites exhibit spatial locality, we can achieve high write savings while caching fewer blocks, reducing the possibility of reads to the HDD. Specifically, we want to find out if some blocks are overwritten more frequently than others. To answer this question, we studied the traces further and make two more observations. First, *there is a high degree of spatial locality in block overwrites*; for example, on an average 1% of the most written blocks contribute to 73% and 34% of the total overwrites in desktop and server traces.

Figure 1 shows the spatial distribution of overwrites for two sample traces: D-1A and S-EXCH. On y-axis, we plot the cumulative distribution of overwrites and in x-axis, we plot the percentage of blocks written. We can notice that a small fraction of the blocks (*e.g.*, 1%) contribute to a large percentage of overwrites (over 70% in D-1A and 33.5% in S-EXCH). For all the traces, we present the percentage of total overwrites that occur in the top 1% of the most overwritten blocks in 7th column of Table 1. We can notice that a small number of blocks absorb most of the overwrite traffic.

The second observation we make is that *the blocks that are most heavily written receive very few reads*. Figure 2 presents the total number of writes and reads in the most heavily written blocks from trace D-1A. We collected the top 1% of the most written blocks and plotted a histogram of the number of writes and reads issued to those blocks. For all the traces, the percentage of total reads that occur in the write-heavy blocks is presented in the last column of Table 1. On average, the top 1% of the blocks in the desktop traces receive 70% of overwrites but only 2.7% of all reads; for the server traces, they receive 0-2% of the reads, excepting S-PRXY1.

To gain some insight into the file-level I/O patterns that cause spatial clustering of overwrites, we compiled a list of the most overwritten files for desktops and present it in Table 2. Not surprisingly, files such as mail boxes, search indexes, registry files, and file system metadata receive most of the overwrites. Some of these files are small enough to fit in the cache (*e.g.*, bitmap or registry entries) and therefore, incur very few reads. We do not report on the most overwritten files in the server traces because they did not contain file-level information. We believe that a similar pattern will be present in other operating systems where majority of overwrites are issued to application-level metadata (*e.g.*, search indexes) and

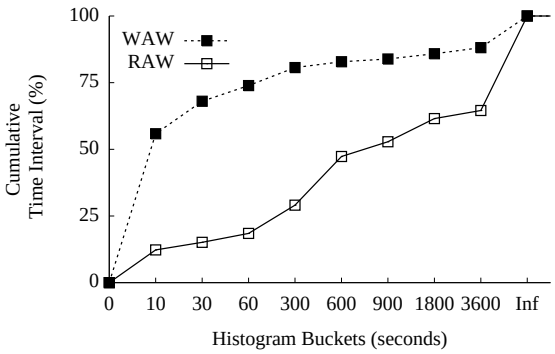


Figure 3: **WAW and RAW Time Intervals.**

system-level metadata (*e.g.*, bitmaps).

At a first glance, such a dense spatial locality of overwritten blocks appears as an opportunity for various optimizations. First, it might suggest that a small cache of few tens of megabytes can be used to handle only the most frequently overwritten blocks. However, separating blocks in this fashion can break the semantic associations of logical blocks (for example, within a file) and make recovery difficult (Section 5.3). Second, a Griffin implementation at the file system-level (Section 7) can easily relocate heavily overwritten files to the HDD. However, when Griffin is implemented as a block device, which is much more tractable in practice, it becomes quite difficult to make use of overwrite-locality lacking file system-level and application-level knowledge.

4.4 Temporal Access Patterns

As mentioned earlier, it is also important to find out how long we can cache a block in the HDD log without incurring expensive reads. To answer this question, we must first understand the temporal access patterns of I/O traces and for that purpose, we define two useful metrics.

Write-After-Write (WAW): WAW is the time interval between two consecutive writes to a block before an intervening read to the same block.

Read-After-Write (RAW): RAW is the time interval between a write and a subsequent read to the same block.

Figure 3 presents the cumulative distribution of the WAW time intervals (indicated by black squares) and the RAW time intervals (indicated by white squares) from 10 seconds to 1 hour for D-1A. Interval larger than 1 hour is indicated by "Inf" on the x-axis. Table 3 presents the WAW and RAW distribution for all the traces.

From Figure 3 and Table 3, we notice that a large percentage of the WAW intervals on desktops are relatively small. In other words, most of the consecutive writes to the same block occur within a short period of time; for example, on average 54% of the total overwrites occur within the first 30 seconds of the previous write. However, this trend is not so clear in servers, where we see widely varying behaviors, most likely depending upon the specific server workloads. But, we still see benefits from long-term caching: on average, 60% of the overwrites in the server traces occur within an hour of a previous write.

	WAW				RAW			
Trace	**30 s (%)**	**60 s (%)**	**900 s (%)**	**3600 s (%)**	**30 s (%)**	**60 s (%)**	**900 s (%)**	**3600 s (%)**
D-1A	68	74	84	88	15	18	53	65
D-1B	71	76	87	90	12	16	49	64
D-1C	69	73	81	86	8	9	19	30
D-1D	76	80	89	93	17	18	27	37
D-2A	51	55	69	75	4	6	22	58
D-2B	38	44	62	70	7	8	13	25
D-2C	28	34	59	68	9	9	16	21
D-2D	25	30	56	66	6	7	16	31
D-3A	40	53	71	78	20	22	31	39
D-3B	57	63	71	75	8	10	27	35
D-3C	60	66	73	77	7	8	40	48
D-3D	62	68	75	79	9	16	50	58
S-EXCH	46	54	100	100	9	16	50	58
S-PRXY1	52	64	98	98	12	37	100	100
S-SRC10	2	2	9	10	0	0	4	6
S-SRC22	15	16	17	85	3	3	14	14
S-STG1	6	7	27	41	1	1	9	9
S-WDEV2	7	20	23	23	0	0	0	0

Table 3: **WAW/RAW Distribution**

In addition, we also notice that the time between a write to a block and a subsequent read to the same block (*i.e.*, RAW) is relatively long. For example, only an average of 30% the written data is read within 900 seconds of a block write. As with the WAW results, the RAW distribution for the server traces also varies depending on the specific workload.

We believe that the time interval from a write to a subsequent read is large due to large OS-level buffer caches and a smaller percentage of most overwritten blocks; as a result, the buffer cache can service most reads that occur soon after a write, exposing only later reads that are issued after the block evict to the block device. These results are similar to the WAW and RAW results presented in earlier work by Hsu *et al.* [9].

We calculated the WAW and RAW time intervals for the most overwritten files from Table 2. Even though the WAW distribution was similar to the overall traces, RAW time intervals were longer. For example, for the frequently overwritten files, only an average of 21% of the written data is read within 900 seconds of a write.

From this temporal analysis, we make two observations that are important in determining the duration of caching in HDD: first, *intervals between writes and subsequent overwrites are typically short for desktops*; sec-

Trace	Time (hr)	Number of 4 KB I/Os	Read (%)	Write (%)	Max Write Savings (%)	Overwrites in top 1% (%)	Reads in top 1% (%)
D-DEV	164	4 M	27	73	62	72	0
S-SVN	165	241 K	32	68	81	50	0
S-WEB	5	7 M	91	9	81	21	0

Table 4: **Linux Traces.**

	WAW				RAW			
Trace	30 s (%)	60 s (%)	900 s (%)	3600 s (%)	30 s (%)	60 s (%)	900 s (%)	3600 s (%)
D-DEV	9	24	35	45	6	24	84	85
S-SVN	23	32	53	67	2	2	6	10
S-WEB	5	22	46	100	5	9	54	95

Table 5: **Linux WAW/RAW Distribution**

ond, *the time interval between a block write and its consecutive read is large (tens of minutes).*

These observations provide us with insight on how long to cache blocks in the HDD before migrating them to the SSD: long enough to capture a substantial number of overwrites (*i.e.*, higher than some fraction of WAW intervals) but not long enough to receive a substantial number of reads to the HDD (*i.e.*, lower than some fraction of RAW intervals). Using different values for the migration interval clearly allows Griffin to trade-off write savings against read penalty.

4.5 Results from Linux

We also examined Linux block-level traces to find out if they exhibit similar behavior. We used traces from previous work by Bhadkamkar *et al.* [3]. Table 4 presents results from 3 traces: D-DEV is a trace from a development environment; S-SVN consists of traces from SVN and Wiki server; and S-WEB contains traces from a web server. We can see certain similarities between the Linux and Windows traces. For example, in the desktop trace, coalescing of overwrites leads to only 38% of the total writes going to the SSD (and thereby resulting in 62% write savings). Also, we can notice spatial locality in overwrites, with no read I/Os in the top 1% of the most written blocks. Table 5 presents the distribution of WAW and RAW time intervals as was presented for the Windows traces. Unlike Windows, only 50% or less of the overwrites happen within 1 hour, which motivates longer caching time periods in the HDD. Although shown here for completeness, we do not use Linux traces for the rest of the analysis.

4.6 Summary

We find that block overwrites occur frequently in real-world desktop and server workloads, validating the central idea behind Griffin. In addition, overwrites exhibit both spatial and temporal locality, providing useful insight into practical caching policies that can maximize write savings without incurring a high read penalty.

5 Prototype Design and Implementation

Thus far, we have discussed HDD-based write caching in abstract terms, with a view to defining policies that indicate what data to cache in the HDD and when to move it to the SSD. The only metrics of concern have been write savings and read penalty.

However, Griffin's choice and implementation of policies are also heavily impacted by other real-world factors. An important consideration is *migration overhead*, both direct (total bytes) and indirect (loss of HDD sequentiality). For example, a migration schedule provided by a hypothetical oracle may be optimal from the standpoint of write savings and read penalty, but might require data to be migrated constantly in small increments, destroying the sequentiality of the HDD's access patterns.

Another major concern is *fault tolerance*; the HDD in Griffin represents an extra point of failure, and certain policies may leave the hybrid system much more unreliable than an unmodified SSD. For example, a migration schedule that pushes data to the SSD while leaving associated file system metadata on the HDD would be very vulnerable to data loss.

Keeping these twin concerns of migration overhead and fault tolerance in mind, Griffin uses two mechanisms to support policies on what data to cache and how long to cache it: *overwrite ratios* and *migration triggers*.

5.1 Overwrite Ratios

Griffin's default policy is *full caching*, where the HDD caches every write that is issued to the logical address space. An alternate policy is *selective caching*, where only the most overwritten blocks are cached in the HDD. In order to implement selective caching, Griffin computes an overwrite ratio for each block, which is the ratio of the number of overwrites to the number of writes that the block receives. If the overwrite ratio of a block exceeds a predefined value (which we call the *overwrite threshold*), it is written to the HDD log. Full caching is enabled simply by setting the overwrite threshold to zero. As the overwrite threshold is increased, only those blocks which have a higher overwrite ratio – as a result of being frequently overwritten – are cached.

Selective caching has the potential to lower read penalty, as Section 4.3 showed, and to reduce the amount of data migrated. However, an obvious downside of selective caching is its high overhead; it requires Griffin to compute and store per-block overwrite ratios. Additionally, as we will shortly discuss, selective caching also complicates recovery from failures.

5.2 Migration Triggers

Griffin's policy on how long to cache data is determined not by per-block time values, which would be prohibitively expensive to maintain and enforce, but by coarse-grained triggers that cause the entire contents of the HDD cache to be flushed to the SSD. Griffin supports three types of triggers:

Timeout Trigger: This trigger fires if a certain time elapses without a migration. The main advantages of this trigger are that it is simple and predictable. It also bounds the recency of data lost due to HDD failure; a timeout value of 5 minutes will ensure that no write older than 5 minutes will be lost. However, since it does not react to the workload, certain workloads can incur high read penalties.

Read-Threshold Trigger: The read-threshold trigger fires when the measured read penalty since the last migration goes beyond a threshold. The advantage of such an approach is that it allows the read penalty, which could be a reason for Griffin's performance hit, to be bounded. If used in isolation, however, the read-penalty trigger can be subject to pathological scenarios; for example, if data is never read from the device, the measured read penalty will stay at zero and the data will never be moved from the HDD to the SSD. This can result in the HDD running out of space, and also leave the system more vulnerable to data loss on the failure of the HDD.

Migration-Size Trigger: The migration-size trigger fires when the total size of migratable data exceeds a certain size. It is useful in bounding the quantity of data lost on HDD failure. On its own, this trigger is inadequate in ensuring low read penalties or constant migration rates.

Used in concert, these triggers can enable complex migration policies that cover all bases: for example, a policy could state that the read penalty should never be more than 5%, and that no more than 100 MB or 5 minutes worth of data should be lost if the HDD fails.

The actual act of migration is very quick and simple; data is simply read sequentially from the HDD log and written to the SSD. Since the log and the actual file system are on different devices, this process does not suffer from the performance drawbacks of cleaning mechanisms in log-structured file systems [26], where shuttling between the log and the file system on the same device can cause random seeks.

5.3 Failure Handling

Since Griffin uses more than one device to store data, failure recovery is more involved than on a single device.

Power Failures. Power failures and OS crashes can leave the storage system state distributed across the HDD log and the SSD. Recovering the state from the HDD log to the primary SSD storage is simple; Griffin leverages well-developed techniques from log-structured and journaling systems [8, 26] for this purpose. On a restart after a crash, Griffin reads the blockmap that stores the log-block to SSD-block mapping and restores the data that were issued before the system crash.

Device Failures. The HDD or SSD can fail irrecoverably. Since SSD is the primary storage, its failure is simply treated as the failure of the entire hybrid storage, even though the recent writes to the log can be recovered from the HDD. HDD failure can result in the loss of writes that are logged to the disk but not yet migrated to the SSD. The magnitude of the loss depends on both the overwrite ratio and the migration triggers used.

In full caching, since every write is cached, the amount of lost data can be high. However, full caching exports a simple failure semantics; that is, every data block that is available from the SSD is older than every missing write from the HDD. This recovery semantics, where the most recent data writes are lost, is simple and well-understood by file systems. In fact, this can happen even in a single device if the data stored on the device's buffer cache is lost due to say, a power failure.

On the other hand, selective caching minimizes the amount of data loss because it writes fewer blocks in the HDD. However, the semantics of the recovered data is more complex and can lead to unexpected errors: that is, some of the data that is present in the SSD might be more recent than the data that is lost from the HDD because of selective caching.

The migration triggers used directly impact the amount of data loss, as explained in the previous subsection. Timeout and migration-size triggers can be used to tightly bound the recency and quantity of lost data.

5.4 Prototype

We implemented a trace-driven simulator and a user-level implementation for evaluating Griffin. The simulator is used to measure the write savings, HDD read penalties, and migration overheads, whereas the user-level implementation is used for obtaining real latency measurements by issuing the I/Os from the trace to an actual HDD/SSD combo using raw device interfaces.

On a write to a block, Griffin redirects the I/O to the tail of the HDD log and records its new location in an internal in-memory map. The recent contents of the in-

memory map are periodically flushed to the HDD for recovery purposes. On a read to the block, Griffin reads the latest copy of the block from the appropriate device.

Whenever the chosen migration trigger fires, the cached data is migrated from the HDD to the SSD. In order to identify the mapping between the log writes and the logical SSD blocks, Griffin reads the blockmap from the HDD (if it is not already present in memory) and reconstructs the mapping. When migrating, Griffin reads the log contents as sequentially as possible, skipping only the older versions of the data blocks, sorts the logged data based on their logical addresses and writes them back to the SSD. As we show later, this migration improves the sequentiality of the data writes to the SSD.

Even though writes are logged sequentially, the HDD may incur rotational latency. Such rotational latencies can be minimized either by using a small buffer (*e.g.*, 128 KB) to cache writes before writing them to the HDD or by using new mechanisms such as range writes [2].

6 Evaluation

6.1 Policy Evaluation

Although we have several caching and migration policies, we must pick those that are not only effective in reducing the SSD writes but also efficient, practical, and high performing. In this section, we analyze all the policies and pick those that will be used for the evaluation of write savings and performance.

6.1.1 Caching Policies

We evaluate the full and selective caching policies by running different traces through the trace-driven simulator, for different overwrite thresholds; a value of zero for the threshold corresponds to full caching. We then measure the write savings and the read penalty. We disable migrations in these experiments, to compare their performance independent of migration policies.

Figure 4a shows the write savings on y-axis for different traces on x-axis. Each stacked bar per trace plots the cumulative write savings for a specific overwrite threshold. From the figure, we notice that using an overwrite threshold can lower write savings, sometimes substantially as in the server traces.

Figure 4b plots the read penalty on y-axis, where each stacked bar per trace plots the percentage of total reads that hit the HDD for the corresponding overwrite threshold. We observe that a high overwrite threshold has the advantage of eliminating a large fraction of HDD reads.

From Figures 4a and 4b, it is apparent that full caching has the advantage of providing the maximum write savings, but suffers from a higher read penalty as well. It is important to note, however, that the read penalty reported in Figure 4b *is an upper bound on the actual read penalty*, since in this experiment data is never migrated from the HDD and all reads to a block that occur after a preceding write must be served from the HDD. In addition, as described in Section 5.1, a non-zero value on the overwrite threshold comes at a high overhead, requiring Griffin to compute and maintain per-block overwrite ratios. It also complicates recovery from failures.

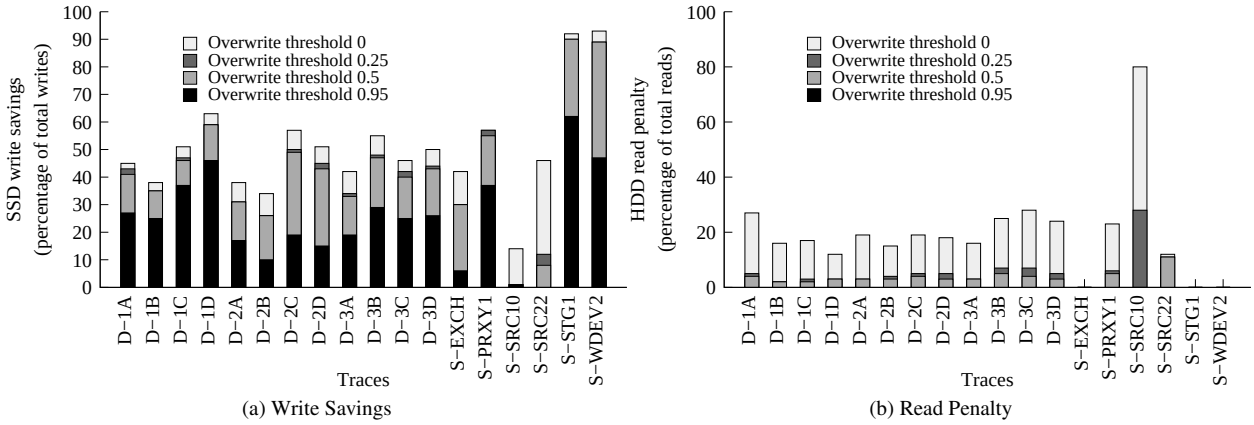
These factors lead us to the conclusion that full caching wins in most cases and therefore, in the remaining experiments, we use full caching exclusively.

6.1.2 Migration Policies

Next, we evaluate different migration policies using the trace-driven simulator. In addition to the write savings, we also measure the inter-migration interval, read penalty, and migration sizes. We start by plotting the write savings for timeout triggers in Figure 5a. We observe that logging for 15 minutes (900 s) gives most of the write savings (over 80% in nearly all cases). For some traces, such as S-STG1, over 1 hour of caching is required to derive better write savings. The durability and large size of the HDD cache allows us to meet such long caching requirements; alternative mechanisms such as volatile in-SSD caches are not large enough to hold writes for more than 10s of seconds.

We also show the read penalty for different timeout values in Figure 5b. We find that the read penalty is low (less than 20%) in most cases except one (S-PRXY1). In particular, read penalty is much lower than the no-migration upper bound reported in Figure 4b, underlining the fact that full caching is not hampered by high read penalties because of frequent migrations. In addition, we also find that timeout-based migration bounds the migration size. The average migration size varied between 91 MB to 344 MB for timeout values of 900 to 3600 seconds.

Figure 6a shows the write savings for read-threshold triggers. Even a tight read-threshold bound of 1% produces write savings similar to those for timeout triggers for most traces. However, the drawback of a smaller read-threshold is frequent migration. Figure 6b plots the average time between two consecutive migrations as a log scale on y-axis for various traces and read penalties. We observe that for most traces, a smaller read-threshold triggers more frequent migrations, separated by as low as 6 seconds as in S-PRXY1. Interestingly, for some traces such as S-WDEV2, which has a very small percentage of reads, even a small read-trigger such as 1% never fires and therefore, the data remains on HDD cache for a long time. As explained earlier (Section 5.3), such behavior increases the magnitude of data loss on HDD failure. The



(a) Write Savings

(b) Read Penalty

Figure 4: **Write Savings and Read Penalty Under Full and Selective Caching.**

migration size varied widely from an average of 129 MB to 1823 MB for 1% to 10% read-thresholds.

Since timeout-based migration was also bounding the migration size, we simplified our composite trigger to consist of a timeout-based trigger combined with a read-threshold trigger. For the rest of the analysis, we use full caching with the composite migration trigger.

6.2 Increased Sequentiality

One of the additional benefits of aggressive write caching is that as writes get accumulated for random blocks, the sequentiality of writes to the SSD increases. Such increased sequentiality in write traffic is an important factor in improving the performance and lifetime of SSDs as it reduces write amplification [10].

Figure 7 plots the percentage of sequential page writes sent to the SSD with and without Griffin, on the desktop and server traces. We use the trace-driven simulator to obtain these results. We count a page write as sequential if the preceding write occurs to an adjacent page. For most traces, Griffin substantially increases the sequentiality of writes observed by the SSD.

6.3 Lifetime Improvement

As mentioned in Section 2, it is not straightforward to compute the exact lifetime improvement from write savings as it depends heavily on the workload and flash firmware. However, given the write I/O accesses, we can find the lower bound and upper bound of the flash block erasures, assuming a perfectly optimal and an extremely simple FTL, respectively.

We ran all the traces on our simulator with full caching and composite migration trigger. The I/O writes are fed into two FTL models to calculate the erasure savings. Ideal FTL assumes a page-level mapping and issues all writes sequentially, incurring fewer erasures. Therefore, erasure savings are smaller on ideal FTL because it is already good at reducing erasures. Simple FTL uses a coarse-grained block-level mapping, where if a write is issued to a physical page that cannot be overwritten, then the block is erased. Based on these models, Figure 8 presents the SSD block-erasure savings, which can directly translate into lifetime improvement.

6.4 Latency Measurements

Finally, we measure Griffin's performance on real HDDs and SSDs using our user-level implementation. We use four different configurations for Griffin's write cache: a slow HDD, a fast HDD, a slow SSD, and a fast SSD. In all the measurements, an MLC-based SSD was used as the primary store. We used the following devices: a Barracuda 7200 RPM HDD, a Western Digital 10K RPM HDD, an Intel X25-M 80 GB SSD with MLC flash, and an Intel X25-E 32 GB SSD with SLC flash with a sequential write throughput of 80 MB/s, 118 MB/s, 70 MB/s, and 170 MB/s respectively. When MLC-based SSD is used for write caching, we used Intel X25-M SSDs as the write cache as well as the primary storage.

Since each trace is several days long, we picked only 2 hours of I/Os that stress the Griffin framework. Specifically, we selected two 2-hour segments, T_1 and T_2, out of all the desktop traces that have a large number of total reads and writes per second that hit the cache. T_2 also happened to contain the most number of I/Os in a 2 hour segment. These two trace segments represent I/O streams that stress Griffin to a large extent. We ran each of these trace segments under full caching with a migration timeout of 900 seconds; Griffin's in-memory blockmap was flushed every 30 seconds. The average migration sizes are 2016 MB and 2728 MB for T_1 and T_2.

Figure 9 compares the latencies (relative to the de-

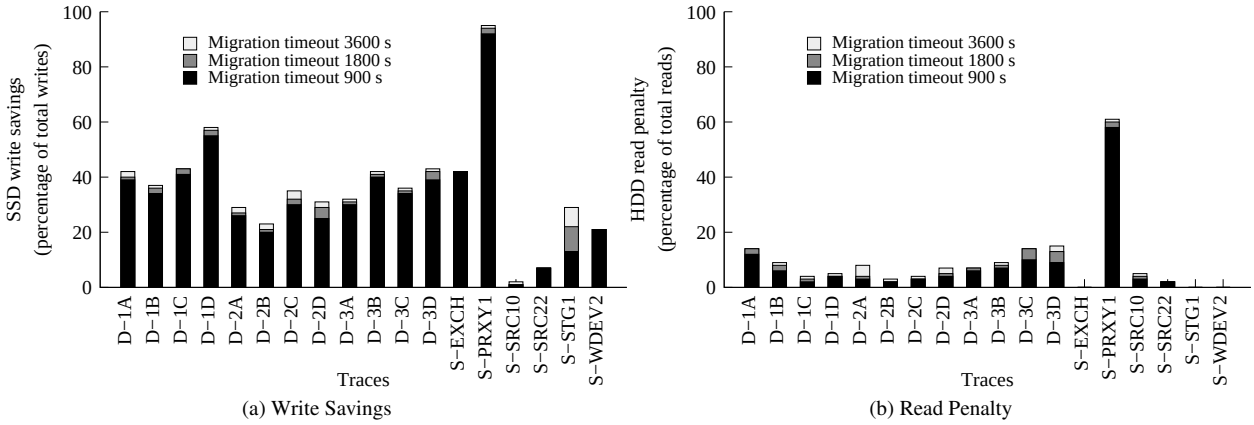


Figure 5: **Write Savings and Read Penalty in Timeout-based Migration.**

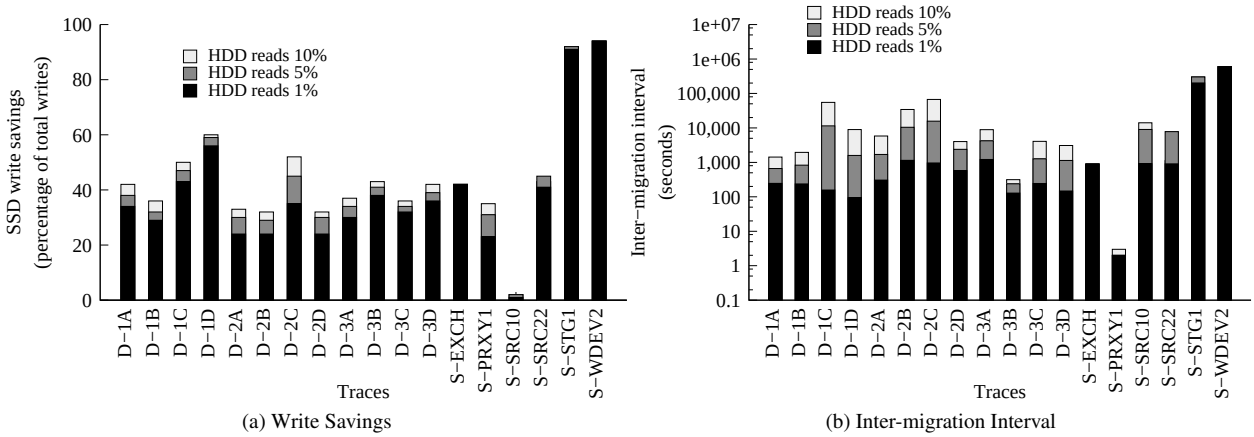


Figure 6: **Write Savings and Inter-migration Interval in Reads-Threshold Migration.**

fault MLC-based SSD) of all I/Os, reads, and writes with different write caches. Unsurprisingly, Griffin performs better than the default SSD in all the configurations (with HDDs or SSDs as its write cache). This is because of two reasons: first, write performance improves because of the excellent sequential throughput of the write caches (HDD or SSD); second, read latency also improves because of the reduced write load on the primary SSD. For example, even when using a slower 7200 RPM HDD as a cache, Griffin's average relative I/O latency is 0.44. That is, Griffin reduces the I/O latencies by 56%. Overall performance of Griffin when using an MLC-based or SLC-based SSD as the write cache is better than the HDD-based write cache because of the better read latencies of SSD. While it is not a fair comparison, this performance analysis brings the high-level point that even when a HDD, which is slower than an SSD for most cases, is introduced in the storage hierarchy the performance of the overall system does not degrade. Figure 9 also shows that using another SSD as a write cache instead of an HDD gives faster performance. But, this comes at a much higher cost because of the price differences between an HDD and SSD. Given the excellent performance of Griffin even with a single HDD, we may explore setups where a single HDD is used as a cache for multiple SSDs (Section 7).

7 Discussion

• **File system-based designs:** Griffin could have been implemented at the file system level instead of the block device level. There are three potential advantages of such an approach. First, a file system can leverage knowledge of the semantic relationships between blocks to better exploit the spatial locality described in Section 4.3. Second, it is possible that Griffin can be easily implemented by modifying existing journaling file systems to store the update journal on the HDD and the actual data on the SSD, though current journaling file systems are

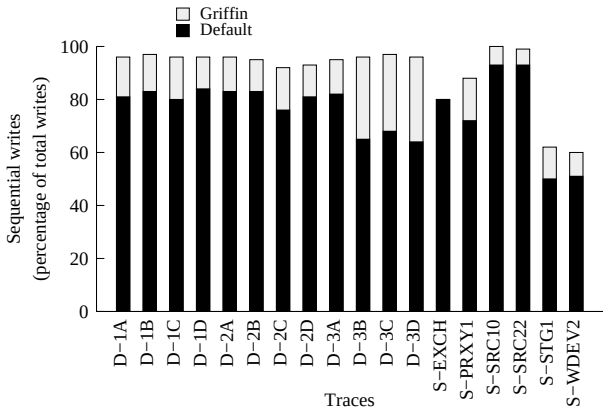


Figure 7: **Improved Sequentiality.**

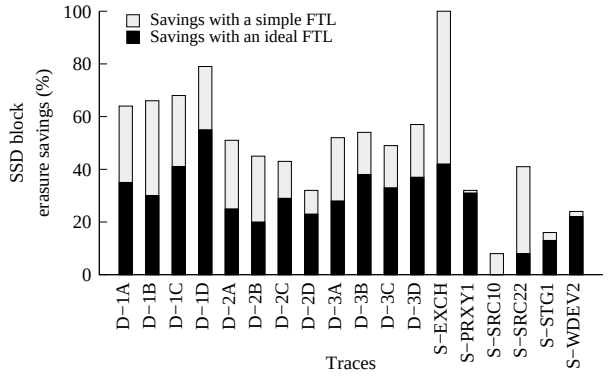


Figure 8: **Improved Lifetime.**

typically designed to store only metadata updates in the journal and many of the overwrites we want to buffer occur within user data.

The third advantage of a file system design is its access to better information, which can enable it to approach the performance of an idealized HDD write cache. Recall that the idealized cache requires an oracle that notifies it of impending reads to blocks just before they occur, so dirty data can be migrated in time to avoid reads from the HDD. At the block level, such an oracle does not exist and we had to resort to heuristic-based migration policies. However, at the file system level, evictions of blocks from the buffer cache can be used to signal impending reads. As long as the file system stores a block in its buffer cache, it will not issue reads for that block to the storage device; once it evicts the block, any subsequent read has to be serviced from the device. Accordingly, a policy of migrating blocks from the HDD to the SSD upon eviction from the buffer cache will result in the maximum write savings with no read penalty.

However, a block device has the significant advantage of requiring no modification to the software stack, working with any OS or architecture. Additionally, our evaluation showed that the simple device-level migration policies we use are very effective in approximating the performance of an idealized cache.

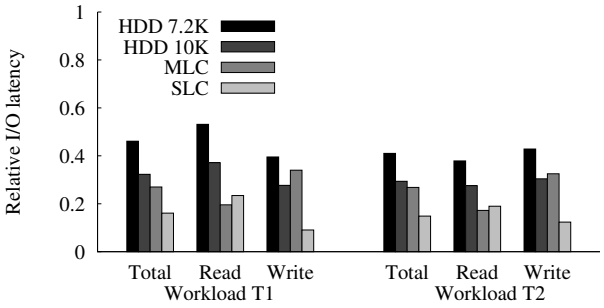


Figure 9: **Relative I/O Latencies for Different Write Caches.**

• **Flash as write cache:** While Griffin uses an HDD as a write cache, it could alternatively have used a small SSD and achieved better performance (Section 6.4). Since SLC flash is expensive, it is crucial that the size of the write cache be small. However, the write cache must also sustain at least as many erasures as the backing MLC-based SSD, requiring a certain minimum size.

Since each SLC block can endure 10 times the erasures of an MLC block, an SLC device subjected to the same number of writes as the MLC device would need to be a tenth as large as the MLC to last as long. If the SLC receives twice as many writes as the MLC, it would need to be a fifth as large.

Consequently, a caching setup that achieves a write savings of 50% – and as a result, sends twice as many writes to the SLC than the MLC – requires an SLC cache that's at least a fifth of the MLC. For example, if the MLC device is 80 GB, then we need an SLC cache of at least 16 GB. In this analysis we assumed an ideal FTL that performs page-level mapping, a perfectly sequential write stream, and identical block sizes for MLC and SLC devices. If the MLC's block size is twice as large as the SLC's block size, as is the case for current devices, the required SLC size stays at a fifth for a perfectly sequential workload, but will drop for more random workloads; we omit the details of the block size analysis for brevity. We believe that a 16 GB SLC write cache (for an 80 GB MLC primary store) will continue to be expensive enough to justify Griffin's choice of caching medium.

• **Power consumption:** One of the main concerns that might arise in the design of Griffin is its power consumption. Since HDDs consume more power than SSDs, Griffin's power budget is higher than that of a regular SSD. One way to mitigate this problem is to use a smaller, more power-efficient HDD such as an 1.8 inch drive that offers marginally lower bandwidth; for example, Toshiba's 1.8 inch HDD [28] consumes about 1.1 watts to seek and about 1.0 watts to read or write, which

is comparable to the power consumption of Micron SSD [18], thereby offering a tradeoff between power, performance, and lifetime. Additionally, desktop workloads are likely to have intervals of idle time during which the HDD cache can be spun down to save power.

Finally, we can potentially use a single HDD as a write cache for multiple SSDs, reducing the power premium per SSD (as well as the hardware cost). Going by the Intel X25-M's specifications, a single SSD supports 3.3K random write IOPS, or around 13 MB/s, whereas a HDD can support 70 to 80 MB/s of sequential writes. Accordingly, a single HDD can keep up with multiple SSDs if they are all operating on completely random workloads, though non-trivial engineering is required for disabling caching whenever the data rate of the combined workloads exceeds HDD speed.

8 Related Work

SSD Lifetimes: SSD lifetimes have been evaluated in several previous studies [6, 7, 20]. The consensus from these studies is that both the reliability and performance of the MLC-based SSDs degrade over time. For example, the bit error rates increase sharply and the erase times increase (by as much as three times) as SSDs reach the end of their lifetime. These trends motivate the primary goal of our work, which is to reduce the number of SSD erasures, thus increasing its lifetime. With less wear, an SSD can provide a higher performance as well.

Disk + SSD: Various hybrid storage devices have been proposed in order to combine the positive properties of rotating and solid state media. Most previous work employs the SSD as a cache on top of the hard disk to improve read performance. For example, Intel's Turbo Memory [17] uses NAND-based non-volatile memory as an HDD cache. Operating system technologies such as Windows ReadyBoost [19] use flash memory, for example in the form of USB drives, to cache data that would normally be paged out to an HDD. Windows ReadyDrive [24] works on hybrid ATA drives with integrated flash memory, which allow reads and writes even when the HDD is spun down.

Recently, researchers have considered placing HDDs and SSDs at the same level of the storage hierarchy. For example, Combo Drive [25] is a heterogeneous storage device in which sectors from the SSD and the HDD are concatenated to form a continuous address range, where data is placed based on heuristics. Since the storage address space is divided among two devices, a failure in the HDD can render the entire file system unusable. In contrast, Griffin uses the HDD only as a cache allowing it to expose an usable file system even in the event of an HDD failure (albeit with some lost updates). Similarly, Koltsidas *et al.* have proposed to split a database store between the two media based on a set of on-line algorithms [15]. Sun's Hybrid Storage Pools consist of large clusters of SSDs and HDDs to improve the performance of data access on multi-core systems [4].

In contrast to the above mentioned works, we use the HDD as a write cache to extend SSD lifetime. Although using the SSD as a read cache may offer some benefit in laptop and desktop scenarios, Narayanan *et al.* have demonstrated that their benefit in the enterprise server environment is questionable [22]. Moreover, any system that forces all writes through a relatively small amount of flash memory will wear through the available erase cycles very quickly, greatly diminishing the utility of such a scheme. Setups with the HDD and SSD arranged as siblings may reduce erase cycles and provide low-latency read access, but can incur seek latency on writes if the hard disk is not structured as a log. Additionally, HDD failure can result in data loss since it is a first-class partition and not a cache.

SLC + MLC: Recently, hybrid SSD devices with both SLC and MLC memory have been introduced. For example, Samsung has developed a hybrid memory chip that contains both SLC and MLC flash memory blocks [27]. Alternatively, an MLC flash memory cell can be programmed either as a single-level or multi-level cell; FlexFS utilizes this by partitioning the storage dynamically into SLC and MLC regions according to the application requirements [16].

Other architectures use SLC chips as a log for caching writes to MLC [5, 12]. These studies emphasize the performance gains that the SLC log provides but do not investigate the effect on system lifetime. As we described in Section 7, a small SLC write cache will wear out faster than the MLC device, and larger caches are expensive.

Disk + Disk: Hu *et al.* proposed an architecture called Disk Caching Disk (DCD), where an HDD is used as a log to convert the small random writes into large log appends. During idle times, the cached data is de-staged from the log to the underlying primary disk [11, 23]. While DCD's motivation is to improve performance, our primary goal is to increase the SSD lifetime.

9 Conclusion

As new technologies are born, older technology might take a new role in the process of system evolution. In this paper, we show that hard disk drives, which have been extensively used as a primary store, can be used as a cache for MLC-based SSDs. Griffin's design is motivated by the workload and hardware characteristics. After a careful evaluation of Griffin's policies and performance, we show that Griffin has the potential to improve SSD lifetime significantly without sacrificing performance.

10 Acknowledgments

We are grateful to our shepherd, Jason Nieh, and the anonymous reviewers for their valuable feedback and suggestions. We thank Vijay Sundaram and David Fields from the Windows Performance Team for providing us the Windows desktop traces. We also thank Dushyanth Narayanan from Microsoft Research Cambridge and Prof. Raju Rangaswami from Florida International University for keeping their traces publicly available. Finally, we extend our thanks to Marcos Aguilera, John Davis, Moises Goldszmidt, Butler Lampson, Roy Levin, Dahlia Malkhi, Mike Schroeder, Kunal Talwar, Yinglian Xie, Fang Yu, Lidong Zhou, and Li Zhuang for their insightful comments.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of USENIX Annual Technical Conference*, pages 57–70, 2008.

[2] A. Anand, S. Sen, A. Krioukov, F. Popovici, A. Akella, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Banerjee. Avoiding File System Micromanagement with Range Writes. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

[3] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proceedings of the File and Storage Technologies Conference*, pages 183–196, San Francisco, CA, Feb. 2009.

[4] R. Bitar. Deploying Hybrid Storage Pools With Sun Flash Technology and the Solaris ZFS File System. Technical Report SUN-820-5881-10, Sun Microsystems, October 2008.

[5] L.-P. Chang. Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs. In *Proceedings of the 13th Asia South Pacific Design Automation Conference*, pages 428–433, Jan. 2008.

[6] P. Desnoyers. Empirical evaluation of nand flash memory performance. In *First Workshop on Hot Topics in Storage and File Systems (HotStorage'09)*, 2009.

[7] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations and applications. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pages 24–33, 2009.

[8] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 155–162, 1987.

[9] W. W. Hsu and A. J. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2):347–372, 2003.

[10] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

[11] Y. Hu and Q. Yang. Dcd - disk caching disk: A new approach for boosting i/o performance. In *Proceedings of the International Symposium on Computer Architecture*, pages 169–178, 1996.

[12] S. Im and D. Shin. Storage architecture and software support for SLC/MLC combined flash memory. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1664–1669, 2009.

[13] Intel Corporation. Intel X18-M/X25-M SATA Solid State Drive. `http://download.intel.com/design/flash/nand/mainstream/mainstream-sata-ssd-datasheet.pdf`.

[14] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, 2008.

[15] I. Koltsidas and S. Viglas. Flashing up the storage layer. *Proceedings of the VLDB Endowment*, 1(1):514–525, 2008.

[16] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2009.

[17] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel®turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *Transactions on Storage*, 4(2):1–24, 2008.

[18] Micron. C200 1.8-Inch SATA NAND Flash SSD. `http://download.micron.com/pdf/datasheets/realssd/realssd_c200_1_8.pdf`.

[19] Microsoft Corporation. Microsoft Windows ReadyBoost. `http://www.microsoft.com/windows/windows-vista/features/readyboost.aspx`.

[20] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill. Bit error rate in NAND Flash memories. In *IEEE International Reliability Physics Symposium (IRPS)*, pages 9–19, April 2008.

[21] D. Narayanan, A. Donnelly, and A. I. T. Rowstron. Write offloading: Practical power management for enterprise storage. In *Proceedings of the File and Storage Technologies Conference*, pages 253–267, San Jose, CA, Feb. 2008.

[22] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158, 2009.

[23] T. Nightingale, Y. Hu, and Q. Yang. The design and implementation of a dcd device driver for unix. In *Proceedings of the USENIX Annual Technical Conference*, pages 295–307, 1999.

[24] Panabaker, Ruston. Hybrid Hard Disk and ReadyDrive Technology: Improving Performance and Power for Windows Vista Mobile PCs . `http://www.microsoft.com/whdc/system/sysperf/accelerator.mspx`.

[25] H. Payer, M. A. Sanvido, Z. Z. Bandic, and C. M. Kirsch. Combo drive: Optimizing cost and performance in a heterogeneous storage device. *First Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, 1(1):1–8, 2009.

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

[27] Samsung. Fusion Memory: Flex-OneNAND. `http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_FlexOneNAND.html`.

[28] Toshiba. MK1214GAH (HDD1902) 1.8-inch HDD PMR 120GB. `http://sdd.toshiba.com/main.aspx?Path=StorageSolutions/1.8-inchHardDiskDrives/MK1214GAH/MK1214GAHSpecifications`.