ASCache: An Approximate SSD Cache for Error-Tolerant Applications

Fei Li[†], Youyou Lu[†], Zhongjie Wu[‡], Jiwu Shu[†*]
[†]Tsinghua University, [‡]Alibaba Group
{lif17@mails.,luyouyou@,shujw@}tsinghua.edu.cn,alanwu.wzj@alibaba-inc.com

ABSTRACT

With increased density, flash memory becomes more vulnerable to errors. Error correction incurs high overhead, which is sensitive in SSD cache. However, some applications like multimedia processing have the intrinsic tolerance of inaccuracies. In this paper, we propose ASCache, an approximate SSD cache, which allows bit errors in a controllable threshold for error-tolerant applications, so as to reduce the cache miss ratio caused by incorrect cache pages. ASCache further trades the strictness of error correction mechanisms for higher SSD access performance. Evaluations show ASCache reduces the average read latency by at most 30% and the cache miss ratio by 52%.

1 INTRODUCTION

As a cost-effective solution, flash-based solid state drives (SSDs) have been widely deployed as a block level cache in large-scale distributed systems. The SSD caching system combines the high-performance SSDs with the low-cost back-end devices (e.g., Hard disk drives (HDDs)) to pursue a high cost performance ratio. The optimization approaches for the SSD caching system target on the life extension of SSDs as well as the performance improvement.

In a typical caching system, utilizing a large cache and adopting an appropriate replacement algorithm are two intuitive and practical approaches to get better cache performance. However, in a SSD caching system, the endurance and reliability problem of SSDs becomes a key challenge. A flash memory cell has limited endurance. A single-level cell (SLC) can tolerate about 100k program/erase (P/E) cycles while a multi-level cell (MLC) can just tolerate about 10k P/E cycles. What's more, emerging trinary-level cell (TLC) and quad-level cell (QLC) could only survive for about 5k and 1.5k P/E cycles. Raw bit error rate (RBER) is the standard metric to evaluate flash reliability, it is defined as the number of corrupted bits per number of total bits read [14]. The reliability mechanisms like the error-correction code (ECC), the redundant arrays of independent drives (RAID) are used to handle the bit errors during the flash lifetime.

However, to retrieve accurate data could cause significant error correction overhead and degrade the overall performance, especially when the RBER is high. The average response time of a SSD will increase proportionally as the strength of ECC needed [19]. What's more, when bit errors of a SSD cache page couldn't be corrected, it will report a read error. The accurate data will be retrieved from the back-end devices as handling a cache miss, thus leading to high latency accesses and even additional writes to SSDs. Fortunately, the bit errors might not be so catastrophic for some applications in domains like computer vision, multimedia processing, machine learning and so on, it is the intrinsic tolerance to inaccuracies of these applications [7]. For instance, we inject bit errors up to a ratio of 10^{-2} to some BMP format photos, and the differences even couldn't be observed by human eye and the calculated quality loss is also in a tolerable degree. Recent open-channel SSDs [10] enable raw page read/write which could be used to allow returning data with bit errors within a controllable threshold[1]. As such, *on one hand, even though some bit errors couldn't be corrected, they could also be tolerated by the applications, and there is no need to trigger a cache miss for those incorrect cache pages. On the other hand, if the bit errors in a flash page are within the tolerable threshold, we could relax the protection of error correction mechanisms and return the inaccurate data directly.*

In terms of these observations, we try to make the tradeoff between accuracy and performance to design an approximate SSD cache. In the application level, it should tolerate a certain degree of bit errors and have accurate metrics of service quality sacrifice. An exception handling procedure is also recommended in case the bit errors transmitted to the applications may cause unexpected consequences. In the caching system level, the SSD cache is designed to be approximate, and the back-end devices should guarantee the end-to-end data integrity. Once the approximately cached data couldn't meet the quality requirements of applications, a cache miss should be triggered to retrieve accurate data. In the SSD device level, the key challenge of providing approximate flash accessing is to identify which flash page could be returned to the application without error correction processes and only allow bit errors within a predefined tolerable bit error rate (BER) to be transmitted to applications. We propose ASCache, an approximate SSD cache, which features the following mechanisms:

- **Error-Aware Space Management**. In the SSD cache, we perform a pessimistic tracking of RBER for flash memory and orga-

[*]Jiwu Shu is the corresponding author. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61772300, 61832011), Research and Development Plan in Key field of Guangdong Province (Grant No. 2018B010109002), and Alibaba Group through Alibaba Innovative Research (AIR) Program.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317778

[1]This can also be supported by interface extension on standard SSDs.

nize the SSD space into different reliability level groups. We expose approximate read/write interfaces, which ensure that the bit errors transmitted to applications are in a controlled threshold and reduce the cache read latency via bypassing the error correction processes.

- **Approximate Caching**. In the SSD caching system, the uncorrectable errors caused by the incompetence of error correction in conventional SSD caches are not immediately handled as cache misses. If the inaccurate data could be tolerated by the applications, it will be handled as a cache hit and return the inaccurate data. This strategy eliminates unnecessary cache misses and the corresponding cache miss penalty.

2 BACKGROUND

Most NAND flash stores data using floating gate transistors and is prone to bit errors. Although encoding more bits per cell could increase the capacity of a SSD without increasing the chip size. However, it also decreases the reliability of flash memory by making it more difficult to store and read the data bits correctly.

Error Pattern. The error pattern of flash memory has been extensively researched [3–5, 14]. The wearing of the flash cells is significant for the bit errors. Researches show that, as the P/E cycles increase, the bit error rate of MLC flash memory may increase exponentially [3]. Another critical factor is the charge leakage, which causes the retention time of the correct data bits to decrease [4]. Other factors like cell-to-cell interference, read disturb and so on, cause bit flips under certain conditions, and their effects are limited. Since flash memory is erased in block level, pages in a block will suffer the same cell wearing, and their RBER characteristic will be similar.

Error Correction. ECC is the most commonly used method to detect and correct raw bit errors which occur within the flash memory, e.g., Bose-Chaudhuri-Hocquenghem (BCH) and low-density parity-check (LDPC) codes. For data written to flash pages, it is transformed into codewords which each consists of the data and correction code. The strength of error correction offered by a specific ECC algorithm is determined by the codeword length and the coding rate (i.e., the data size divided by the codeword size). For a specific ECC algorithm and a certain codeword length, a higher coding rate provides weaker protection but consumes less additional storage space.

For BCH or LDPC, SSD performs several stages of error correction to retrieve accurate data, which is known as the error correction flow [6]. LDPC is now more widely used in commercial SSDs. In the LDPC error correction flow, the ECC engine performs hard decoding in the first stage which takes about $8\mu s$. In hard decoding, the ECC engine only uses the hard bit value information read from a cell using a single set of reference voltages. If the first stage error correction succeeds, the flow finishes. Otherwise, the flow moves on to the next stage and performs soft decoding. The key idea of soft decoding is to use the soft information of each cell through multiple reads with different sets of reference voltages. Soft information is typically represented as the log-likelihood ratio (LLR). The maximum read level, which determines the strength of protection, is usually set to be six. Generally, data will be decoded level-by-level, and for each additional soft decoding level, it takes about $80\mu s$ to perform an additional flash memory sensing and codeword decoding. Zhao et al. [19] proposes to run operations in different levels concurrently to reduce the overhead when the soft decoding level is predetermined. However, the soft decoding still significantly increases the read latency. In the third stage, the intra-SSD chip-level RAID is triggered. It uses the data in other pages to recover the lost page data and costs about $10ms$. In conventional SSDs, only the successfully decoded or recovered data will be sent to the host. Otherwise, an uncorrectable error will be reported. Each stage including different levels of soft decoding in the LDPC correction flow represents a certain level of protection strength. What's more, the later the flow finishes, the higher the error correction overhead it consumes to retrieve accurate data.

Error Avoiding. To avoid bit errors in flash memory, Cai et al. [4] provides insights into the data retention problem and proposes data refresh mechanisms to handle it by periodically refreshing data with different strategies in SSDs. The threshold voltage in flash reading operations also significantly affects the RBER observed. Some researches [3, 5] propose to tune the threshold voltage dynamically by estimating the RBER of flash. To avoid the bit patterns which is error-prone, a guided scrambling mechanism [15] is proposed to add a pseudo-random sequence to the source data to make such errors less likely.

Approximate Storage. With rising performance demands, the resource budgets including processors and storage space in SSDs are limited for those reliability mechanisms. Researches propose approximate storage which exploits the applications' error tolerance to trade accuracy for performance or energy efficiency. Researches on approximate solid-state storage mainly focus on the flowing aspects: (1) The quantitative analyzing of the applications' intrinsic inaccuracies tolerance [7, 18], application data including photos, mobile phone sensor logs, machine learning models have been intensively studied. (2) Hardware approximation methods, such as writing to embedded flash memory at voltages lower than the recommended microcontroller's specification [12] or lowering the supply voltage to the flash chips during its operations[11], using high-density phase change memory to store photos [8], etc. (3) Application adaptation. Such as analyzing the format of compressed multimedia files to classify data bits into different importance level and store them in different storage regions [8, 9], designing a programming framework for applications to perform approximate computing [13], etc.

ASCache is a fresh attempt in building an approximate SSD cache. Different from the approximate storage mentioned above, ASCache provides the approximation of flash by relaxing the error correction strength rather than adopting the hardware-level approximation, and it is simpler to be applied for practical usage. To adopt approximate SSD caches, ASCache shifts the responsibility of data integrity from SSDs to back-end devices and allows controlled bit errors to be returned to the host. The *Error-Aware Space Management* matches the error tolerance of applications with the RBER characters of flash memory to fully utilize the reliability resources in SSDs. The *Approximate Caching* transmits the uncorrectable errors of reading incorrect flash pages to error-tolerant applications to reduce the cache miss ratio as well as the corresponding cache miss penalty. The overall design trades the accuracy of applications for the performance improvement of SSD cache.

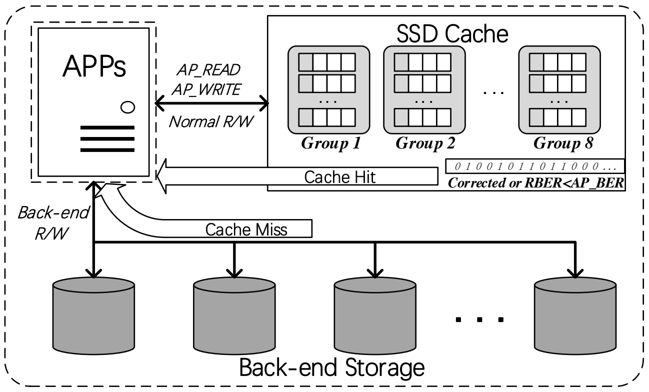

Figure 1: The ASCache Architecture

3 DESIGN

3.1 The ASCache Architecture

As in Figure 1, ASCache is organized based on the conventional SSD caching system architecture with the following key differences: (1) **SSD Cache Accessing**. ASCache defines approximate read/write interfaces as well as maintaining the normal ones. The data retrieved through the approximate read interface will be accurate or contain bit errors within the application's tolerance. (2) **SSD Space Management**. The flash blocks in SSD caches are organized by different reliability levels. ASCache prefers allocating physical pages with a specific reliability level, in which the RBER of flash is within the application's tolerance threshold. In this case, data could be returned to the application even without the error correction process. (3) **Caching Mechanisms**. ASCache narrows the scope of the cache miss by transforming specific uncorrectable errors to cache hits, and it fits the approximate SSD cache to different cache write strategies to provide end-to-end data integrity.

3.2 Error-Aware Space Management

For an error-tolerant application, data retrieved from the storage system should have a lower bit error rate than it could tolerate. With specific metrics, the tolerable threshold for applications could be determined via data analysis. However, the RBER of flash memory is dynamically changing. To build an approximate SSD cache, we should organize the SSD space based on an appropriate flash RBER estimation and provide proper approximate accessing interfaces to guarantee that the RBER of the allocated physical flash pages would always be in the tolerable threshold.

Pessimistic Tracking of RBER. Precisely tracking the RBER of every flash pages in real time is a complex and costly task. On the basis of the prior knowledge about error patterns of flash memory, we try to perform a pessimistic tracking of flash RBER in block level for a SSD.

P/E cycles (P/Es) and data retention time ($T_{retention}$) are two main factors which affect the RBER of flash. The RBER of a flash page could be described as a simplified model:

$$RBER = G(P/Es) + H(T_{retention}) \tag{1}$$

Generally, G(x) is described as exponential, and H(x) is thought to be linear. P/Es, also cell wearing, has the most significant impact on RBER. $T_{retention}$ may function in a time scale of a day or even larger. As pages in one block suffer the similar cell wearing, and

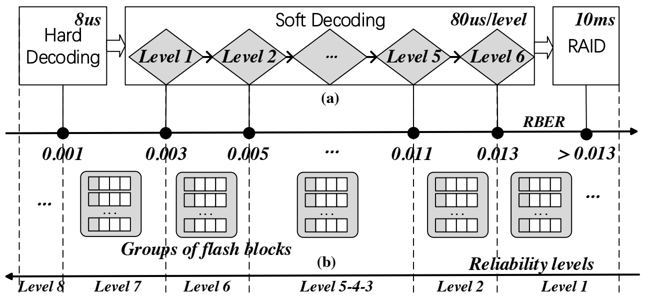

Figure 2: (a) Stages of a typical 8/9 rate LDPC error correction flow, and the latency and correctable RBER of each stage when the UBER is kept within 10^{-16} via Monte Carlo simulations; (b) Example for the organization of flash blocks and reliability levels in ASCache.

retention errors grow as data retention time increases, we utilize the RBER of the first written page in a block as the upper RBER limit ($RBER_U$) of this block, which is maintained as metadata in block management mechanism. Instead of tracking the RBER of all pages separately, we use $RBER_U$ as the RBER of all pages in the same block. It allows RBER variation for most pages in a block and provides a better assurance of being within the tolerable threshold when allocating pages.

The $RBER_U$ of each flash block is updated dynamically. To simplify the update process, the first page or another optional page of each block is reserved as *the reference page* which will be used to track and update the $RBER_U$. We could get the RBER of a page through the error correction process of ECC engine when reading this page. When a block is erased in the garbage collection (GC) process, we write random data to the reference page of this block. Then, record the RBER of this page ($RBER_{ref}$) to update the $RBER_U$. When the system is idle, we can record the RBER of the reference page, and then compare the $RBER_{ref}$ with the present recorded $RBER_U$ ($RBER_{U\_old}$), and use the bigger one to update it. Besides, considering the data retention effect during the $RBER_U$ update intervals, we add a small RBER bias (δ) to $RBER_U$ to get a more pessimistic value. The updating process could be summarized as:

$$RBER_U = \begin{cases} RBER_{ref} + \delta, & When\ GC \\ Max(RBER_{ref}, RBER_{U\_old}) + \delta, & When\ idle \end{cases} \tag{2}$$

By performing a pessimistic tracking for the flash memory with $RBER_U$, we could characterize the bit errors in SSDs at the flash block level efficiently. Furthermore, when retrieving data from a page in a specific flash block, we use $RBER_U$ of this block as the RBER for this page to determine whether it can be tolerated or not.

Reliability Levels and Space Grouping. The standard metric used to describe uncorrectable errors is uncorrectable bit error rate (UBER), which is defined as the codeword failure rate divided by the codeword length. Contemporary SSD manufacturers target a UBER of 10^{-16} [1]. As Figure 2(a) shows, the hard decoding stage could handle RBER within about 1×10^{-3}. When the RBER falls into about 1×10^{-3}~1.3×10^{-2}, the six-levels soft decoding will be triggered. With a higher RBER, RAID would be necessary for data recovery. We classify the reliability status of flash memory into eight levels (from level 1 to level 8) according to the LDPC error correction flow. In level 1, the RBER is higher than the last level soft decoding. In level 2~7, the RBER value range is determined by the two adjacent decoding processes. For example, the level 7 RBER range is between the threshold of hard decoding and the first level soft decoding. The level 6 RBER range is between the threshold of the first level and the second level soft decoding. Level 2~5 are in a similar way. In level 8, the RBER is smaller than the hard decoding threshold. In our definition, the higher level represents the stronger reliability.

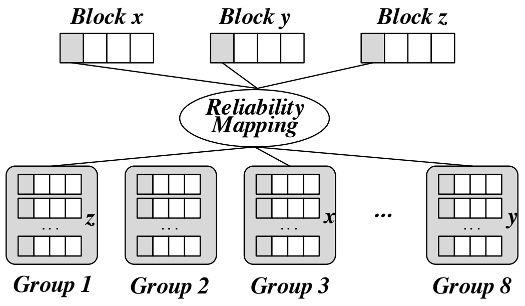

Figure 3: Example for SSDs block management.

As in Figure 3, based on the estimation of flash RBER and the definition of reliability levels, we dynamically gather flash blocks into different groups. Groups are marked with *group id* from 1 to 8. Each group corresponds to a specific reliability level as Figure 2(b) shows. When the $RBER_U$ of a block is updated, the block will be re-mapped to a specific block group. The mapping process is as follows: (1) Comparing the $RBER_U$ of a block with the RBER thresholds of different reliability levels to determine its reliability level. (2) Gathering the block into the corresponding block group, and removing its record from its old block group. We could use simple data structures such as arrays or vectors to organize the block groups, and the grouping process above could be accomplished in linear time. At the early life stage of a SSD, most blocks will be gathered in groups with high-reliability levels. With the wearing of flash cells, some blocks will be reorganized into low-reliability levels. By grouping blocks with different reliability levels, we could find proper blocks for application data efficiently when given a specific application BER tolerant threshold.

Approximate Flash Accessing. As Table 1 shows, we expose two sets of read/write interfaces for accessing flash memory. The normal read/write provides strict data integrity as in conventional SSDs. *In normal write*, blocks with the highest reliability level will be allocated to write data. The correction code and RAID parity will also be written to flash. *In normal read*, data will go through the error correction flow to correct all bit errors. Otherwise, it reports a read error. The approximate read/write provides relaxed data integrity in flash accessing. **AP_BER** is the user-defined parameter to describe the tolerable BER of the application. *In AP_WRITE*, we map the AP_BER to a specific reliability level first, then try to allocate flash blocks gathered in higher levels. What's more, we prefer to allocate pages from blocks in the highest available level. If no available blocks, then allocate space from the lower level groups, and we will mark those pages as *mismatched* to indicate that the AP_BER is lower than the RBER of them. The correction code and RAID parity would be an optional choice if allocated blocks have smaller RBER than AP_BER. We choose to maintain these redundant data to support error correction after AP_WRITE both in normal read

Table 1: Description of Flash Accessing Interfaces

Interface	Parameter	Description
READ	(LPA, ...)	Normal read
WRITE	(LPA, buffer, ...)	Normal write
AP_READ	(LPA, AP_BER ...)	Approximate read
AP_WRITE	(LPA, buffer, AP_BER ...)	Approximate write

and AP_READ. *In AP_READ*, if the flash page isn't marked as mismatched, then retrieve the $RBER_U$ of its corresponding block. If AP_BER is higher than the $RBER_U$, data with bit errors will be returned to the host without error correction. Otherwise, it will go through the error correction flow to correct the bit errors. If the flow fails, it returns a read error. Since AP_WRITE prefers allocating blocks which have lower RBER than AP_BER, AP_READ could avoid triggering the error correction flow in most flash accesses. Besides, the reliability level is divided according to the error correction flow, when it needs to correct bit errors, we could predict the finish point of the flow and avoid doing soft decoding level-by-level. When the flash memory wears seriously, the read errors caused by codeword failure will happen more frequently in conventional SSDs. If AP_BER is high enough, the AP_READ could still provide accessing service and avoid unnecessary read errors.

3.3 Approximate Caching

Although we provide approximate flash accessing interfaces and attempt to keep the bit errors returned to be within the application's tolerance, to design an approximate SSD caching system is still facing challenges. The first challenge is to guarantee the end-to-end data integrity. In conventional systems, SSDs and back-end devices both exploit some reliability mechanisms to avoid the uncorrectable errors. Data retrieved from SSDs is ensured to be accurate, or it will return read error to the host. In ASCache, we relax the data integrity in SSDs, AP_READ may return inaccurate data. Suppose that the correct data is cached and has not been written to the back-end devices. When flushing or data replacement occurs, the approximately cached data will be written back to back-end devices as a correct copy, then the accurate data is lost. The other challenge is to adopt the approximate SSD in the caching system. The approximate SSD allows approximate accessing which will avoid the overhead of error correction mechanisms. Besides, the caching system also focuses on decreasing the cache miss ratio and prolonging the lifetime of SSDs. We should reconsider the cache design to fit in the approximate SSD.

Table 2: Approximate SSD Cache Strategies

Write Strategy	Data Integrity Requirements
Write Back	Relaxed for Applications, Strict for Back-end Storage
Write Through	Relaxed for Applications
Write Around	Relaxed for Applications

Cache Write Strategies. Table 2 shows three cache write strategies. In order to provide end-to-end data integrity, we discuss the writing process in these strategies.

In Write Back caches, data will be written to SSD cache first, and a special flag is used to indicate that the data has not been persisted to back-end devices. When data replacement or cache flushing happens, we will read the relevant data from SSD cache and then write the data back to back-end devices. We could use the normal or approximate write interface to cache data in SSDs while must use the normal read interface to retrieve accurate data in the data write-back process. In this way, the correct copy is persisted. For applications, they could still use AP_READ when adopting normal write or AP_WRITE. Since the reliability level of data could easily be retrieved, and the cache write interfaces allocate flash pages from the available highest reliability group, there is a large chance that the RBER of the flash block will be smaller than AP_BER.

In Write Through and Write Around caches, correct data will be written to back-end devices directly when accesses come to the SSD caching system. The back-end devices maintain the correct copy of data, and this ensures the end-to-end data integrity. In view of this, the SSD cache could be accessed using AP_READ and AP_WRITE as well as normal read/write.

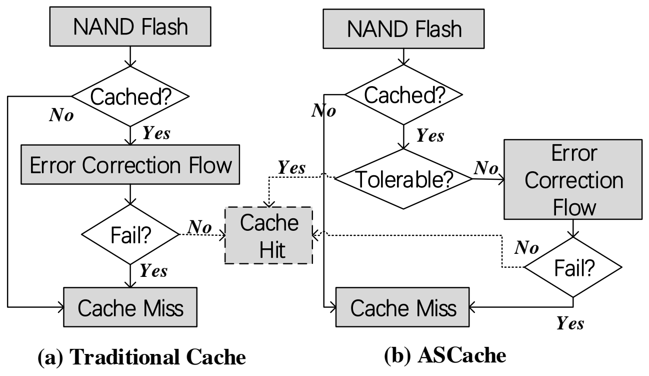

Figure 4: Cache miss in SSD caching systems

Cache Miss Elimination. In conventional SSD caching system, as Figure 4(a) shows, if data is not cached in SSD cache or the correction flow fails, a cache miss occurs. To handle a cache miss, the system will retrieve the correct data from the back-end devices and return it to the application. Then the correct data will also be written into the SSD cache for future access. If the cache miss results from the error correction failure, the old version of data should be invalided, and the new version should be written into a more reliable place in the SSD cache.

By exploiting the approximate flash accessing interfaces, we narrow the scope of cache miss. As in Figure 4(b), unlike the conventional system in Figure 4(a), when reading data bits from flash memory, we could determine its reliability level and whether the application could tolerate the corresponding BER or not. Only when the application couldn't tolerate such bit errors, the error correction flow is triggered as in the conventional SSDs. If AP_BER is higher than the RBER threshold of the error correction flow, the read errors in conventional SSD cache caused by error correction failure could be handled as normal reads by AP_READ in ASCache rather than incurring cache misses. Then the cache miss penalty, such as additional reads to back-end devices and writes to SSDs, could also be avoided.

When a cache miss occurs, data replacement might be triggered to swap a valid data page out and store the data retrieved from back-end devices there. If the cache miss is caused by the error correction failure, it could be eliminated in ASCache, and the valid page could also be retained. In other words, ASCache is more friendly for replacement algorithms by maintaining the data locality to a large extent as in the memory level cache system.

4 EVALUATIONS

In this section, we evaluate ASCache on the following aspects: (1) The benefits of adopting approximate SSD cache. (2) The tradeoff between accuracy and performance. We use a qemu-based NVMe open-channel SSD emulator to build the approximate SSD cache, detailed specification of the emulated flash is in Table 3. To simplify the evaluation process, we scale down the capacity of SSD and HDD to be 812MiB and 20GiB. What's more, the average HDDs read latency is about $3ms$. The caching system is tested under the scenario of video distribution, in which many clients concurrently request different video streams. All videos [17] are in the YUV4MPEG format which is convenient for bit error injection and analysis.

Table 3: The Flash Specification

Geometry	8 luns, 1024 blocks, 65536 pages; OP=20%
Page Size	16KB with 1KB OOB data
Latency	Read 80μs, Write 200μs, Erase 1200μs

Average Read Latency. Figure 5 compares the average read latency in traditional SSD cache and ASCache when AP_BER gradually increases. We model the reliability status of flash blocks in a Gaussian distribution. Most blocks are gathered in the middle level groups as in Figure 5. In this distribution, the SSD blocks are at the life stage when the LDPC soft decoding is needed and the correction flow could handle the bit errors in most accesses. So no cache miss happens due to read errors in the traditional cache, the cache hit ratio in ASCache and traditional cache are close to each other, and it is about 0.83 in our evaluation. We observe that as the AP_BER increases, the average read latency of ASCache will decrease while the traditional cache remains unchanged. It is because the accesses to blocks which have a lower RBER than the AP_BER will bypass unnecessary error correction processes in ASCache while must still go through the correction flow to correct the bit errors in traditional SSD cache. This case demonstrates the effectiveness of approximate SSD accessing. Especially, if the AP_BER is more than 0.011 when bit errors in most blocks could be tolerated, the average read latency drops about 30% in ASCache.

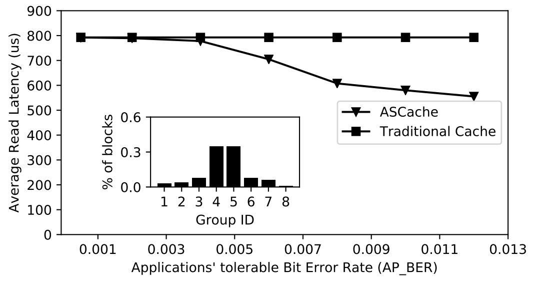

Figure 5: Average read latency under different AP_BER; The cache hit ratio is about 0.83

Cache Misses. When the RBER exceeds the LDPC soft decoding threshold, the codeword failure rate will increase significantly. We simulate different codeword failure rate in traditional SSD cache to test the cache miss ratio. To simplify the evaluation, we simulate

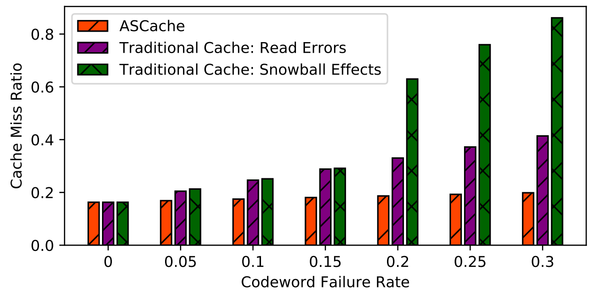

Figure 6: Cache misses under different codeword failure rate

the RAID process as cache misses for its overhead is much larger than the HDDs access. What's more, we suppose the AP_BER could tolerate almost all accesses and only simulate a small portion which increases as the failure rate to be intolerable. In Figure 6, as the codeword failure rate increases, the cache miss ratio due to read errors increases proportionally in the traditional cache. However, the ratio in ASCache almost remains unchanged. This case demonstrates the effectiveness of cache miss elimination. Especially, when the codeword failure rate is 0.3, up to 52% cache misses due to read errors in the traditional cache is eliminated by ASCache. What's more, read errors will incur snowball effects, because retries or some other handlers, which even affect the latency of other accesses, will be triggered. Designs like bcache [2], adopt the cache bypass mechanism, which will bypass the SSD cache when the latency exceeds a defined threshold. As in Figure 6, due to the snowball effects of read errors, the cache miss ratio in traditional cache increases tremendously when the codeword failure ratio exceeds 0.2.

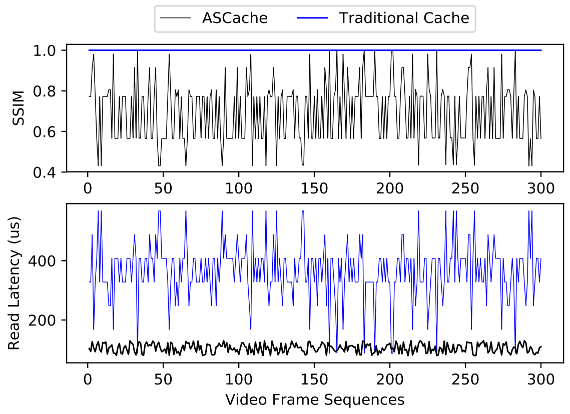

Figure 7: SSIM and read latency of akiyo_cif video frames.

Case Study. To further demonstrate the tradeoff between accuracy and performance in ASCache, we collect the access traces of a specific video and analyze the quality and average read latency of each video frame in ASCache as well as in the traditional cache. The *structural similarity (SSIM) index* [16] is used to measure the video quality. The reliability status of SSD blocks is also modeled as a Gaussian distribution as in Figure 5. AP_BER is set to tolerate almost all the RBER of blocks. In the SSIM curves in Figure 7, the video frames read from ASCache contain bit errors with different RBER, about 87.7% of frames have a SSIM value between 0.4~0.8, and 0.4 is the largest quality loss in this evaluation. The video frames read from the traditional cache are error-free, and the SSIM value of frames is 1. In the latency curves in Figure 7, since traditional cache uses error correction mechanisms to correct bit errors, the average read latency of each frame varies with the RBER of different blocks. About 93.6% accesses have the latency of more than 200μs. However, the average latency of frame pages in ASCache varies between a smaller range between 80~100μs. The evaluation is based on the fact that the application could tolerate 0.4 SSIM quality loss. If higher SSIM is required, the frames which don't meet this requirement should go through the correction flow as in the traditional cache, and the read latency of these frame pages will increase. This case demonstrates the quality loss and corresponding performance gain in ASCache. We can observe that ASCache provides a practical and effective solution to trade the accuracy for performance by exploiting the error tolerance of applications.

5 CONCLUSIONS

In order to eliminate the influence of flash bit errors on the performance of SSD cache, we propose ASCache, an approximate SSD cache for error-tolerant applications. On one hand, ASCache allows inaccurate data for applications and relaxes the strictness of error correction mechanisms. On the other hand, ASCache redesigns the cache strategies to fit in the approximate SSD cache and further avoids unnecessary cache misses while guaranteeing the end-to-end data integrity. Our evaluations demonstrate the effectiveness of ASCache and describe the accuracy-performance tradeoff in detail with a case study. What's more, with the insight into the error correction flow and RBER of flash, ASCache provides a new method for flash approximation. We believe that ASCache can be adopted for much more error-tolerant applications and future researches.

REFERENCES

[1] JEDEC Solid State Technology Association. 2010. Solid-State Drive (SSD) Requirements and Endurance Test Method.
[2] Bcache. 2018. Bcache Wiki. https://bcache.evilpiepirate.org.
[3] Yu Cai et al. 2012. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *DATE*.
[4] Yu Cai et al. 2012. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *ICCD*.
[5] Yu Cai et al. 2013. Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis, and modeling. In *DATE*.
[6] Yu Cai et al. 2017. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery. *arXiv preprint arXiv:1711.11427* (2017).
[7] Sampson et al. 2014. Approximate storage in solid-state memories. *TOCS* (2014).
[8] Qing Guo et al. 2016. High-density image storage using approximate memory cells. In *ASPLOS*.
[9] Djordje Jevdjic et al. 2017. Approximate storage of compressed and encrypted videos. In *ASPLOS*.
[10] Youyou Lu et al. 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *FAST*.
[11] Amir Rahmati et al. 2016. Approximate flash storage: A feasibility study. In *Workshop on Approximate Computing Across the Stack*.
[12] Mastooreh Salajegheh et al. 2011. Exploiting Half-Wits: Smarter Storage for Low-Power Devices. In *FAST*.
[13] Adrian Sampson et al. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*.
[14] Bianca Schroeder et al. 2016. Flash Reliability in Production: The Expected and the Unexpected. In *FAST*.
[15] Beomkyu Shin et al. 2012. Error control coding and signal processing for flash memories. In *ISCAS*.
[16] Zhou Wang et al. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* (2004).
[17] Xiph.org. 2018. Video Test Media. https://media.xiph.org/video/derf/.
[18] Xin Xu and H Howie Huang. 2015. Exploring data-level error tolerance in high-performance solid-state drives. *IEEE Transactions on Reliability* (2015).
[19] Kai Zhao et al. 2013. LDPC-in-SSD: making advanced error correction codes work effectively in solid state drives. In *FAST*.