2018 IEEE 36th International Conference on Computer Design

# Pensieve: a Machine Learning Assisted SSD Layer for Extending the Lifetime

Te I*, Murtuza Lokhandwala†, Yu-Ching Hu* and Hung-Wei Tseng*
*Department of Computer Science, †Department of Electrical and Computer Engineering
North Carolina State University

***Abstract*—As the capacity per unit cost dropping, flash-based SSDs become popular in various computing scenarios. However, the restricted program-erase cycles still severely limit cost-effectiveness of flash-based storage solutions.**

**This paper proposes Pensieve, a machine-learning assisted SSD firmware layer that transparently helps reduce the demand for programs and erases. Pensieve efficiently classifies writing data into different compression categories without hints from software systems. Data with the same category may use a shared dictionary to compress the content, allowing Pensieve to further avoid duplications. As Pensieve does not require any modification in the software stack, Pensieve is compatible with existing applications, file systems and operating systems. With modern SSD architectures, implementing a Pensieve-compliant SSD also requires no additional hardware, providing a drop-in upgrade for existing storage systems.**

**The experimental result on our prototype Pensieve SSD shows that Pensieve can reduce the amount of program operations by 19%, while delivering competitive performance.**

## I. Introduction

With the capacity per unit cost of flash memory technologies improves as well as flash memory's low latency, non-volatile, low-power and shock resistance natures, flash-based solid state drives (SSDs) become popular in all computing scenarios, ranging from mobile phones, personal computers to data center servers. However, the limited number of program-erase cycles and the asymmetrical granularities for read/program operations versus erase operations restricts the cost-effectiveness of flash-based storage solutions. With modern TLC (Triple-level cell) flash memory chip technologies, a cell can start to wear out after 3,000 program-erase cycles and the whole SSD may become unusable if a sufficient amount of cells do not function correctly [25].

To extend the lifetime of an SSD without increasing hardware costs, the system needs to mitigate the demand for programming and erasing data. As compression and deduplication techniques reduce data sizes, the storage system can apply these techniques to reduce the amount of writes to the device or increase the effective capacities. Integrating data compression or deduplication into the storage device allows the system to take advantanges from these schemes without modifying the application or the file system. The system can also optimize the compression or deduplication according to device characteristics [35].

However, applying data compression or deduplication within the device also results in some trade-offs in performance. First, as the storage device only receives block addresses in I/O commands, the storage device cannot use high level context-related information to avoid the computation overhead on compressing files that are already in compressed format or use optimized algorithms for different file types. Second, to accommodate the increased computation overhead, existing solutions need to equip hardware accelerators to avoid significant performance degradation, increasing the hardware costs of the device. Finally, existing solutions decouple the execution of compression and deduplication, resulting in addition writes as doing both is a two-phase process.

This paper presents *Pensieve*, a machine learning assisted SSD layer that achieves transparent, low-overhead data reduction to extend SSD lifetime without increasing device costs. Pensieve leverages idle processor cores that are already presented in modern SSD controllers. Pensieve simply needs to scan very small amount of data in each chunk of written data in the write buffer to make accurate predictions on the context. Pensieve's prediction result will guide whether the SSD needs to compress the data or how to compress the data before writing into the flash medium.

Pensieve brings several benefits to the storage system. First, Pensieve's machine learning model builds upon the similarity of data, Pensieve can compress data blocks belong to the same category to share a compression dictionary, achieving the effects of compression and deduplicating redundant dictionary entries simultaneously. Second, with light-weight machine learning models categorizing the context of incoming data, Pensieve does not rely on hints from the software system. Therefore, Pensieve requires no changes to the host software stack. Third, Pensieve predicts data that are uncompressible or have low potential in compression, allowing the SSD to reduce the computation overhead. Finally, with the simplicity of the in-line compression and deduplication mechanisms that Pensieve enables, Pensieve incurs almost no impact on the cost and performance of SSDs, providing a drop-in upgrade applicable to existing systems.

In describing Pensieve, this paper makes the following contributions:
(1) It provides a machine learning assisted, low-overhead mechanism that naturally achieve both effects of compressing data content and deduplicating dictionary contents.
(2) It presents a machine learning based model that can reconstruct the missing context information and predict uncompressible data without going through every bit of the data or using additional file system or software hints.
(3) It demonstrates that the proposed mechanism in Pensieve requires no change to the software system and adds no overhead to host processors.
(4) It shows that Pensieve provides these benefits without increasing the total cost of ownership through prototype implementation.

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00016

This paper evaluates the performance of Pensieve using a custom-built platform that resembles the design of modern SSDs. Using the Pensieve SSD prototype in a contemporary server machine configuration, the system achieves the same-level performance without additional costs to the hardware. Pensieve successfully reduces the amount of written data by 18%.

The rest of this paper is organized as follows: Section II briefly provides the background of modern SSD architectures. Section III provides an overview of the Pensieve design. Section IV depicts the machine learning model used in Pensieve. Section V introduces the architecture of Pensieve. Section VI presents our results. Section VII provides a summary of related work to put this project in context, and Section VIII concludes the paper.

## II. Background

Pensieve proposes enhancements in the flash translation layer of an SSD with a mechanism that achieves both deduplication and compression. This section will provide some background materials that place the proposed mechanism in context.

### A. Flash translation layer

Flash-based SSDs become the mainstream of high-performance storage alternatives due to their features including shorter access latency, lower power consumption, shock resistance comparing with magnetic hard disk drives (HDDs) as well as they are hitting a reasonable price per unit storage. However, flash memory technologies have several characteristics that SSDs need to especially address.

First, flash memory technologies do not allow reprogramming a cell until the chip performs an erase operation for the cell. Second, although modern flash memory technologies can read and write in the granularity of pages, where each page usually stores 4 KB – 8 KB of data, the erase operation can only perform in the unit of blocks, where each block may contains 64 to 512 pages. In addition, the latencies for read, program and erase operation vary a lot. Reading a page can take less than 100 $\mu$s, but programming a page can take 100 $\mu$s to 2000 $\mu$s. Erasing a block takes even longer, as this operation usually takes several milliseconds to complete [7]. Finally, each block can only function correctly within a limited amount of program-erase cycles. A cell in a block can become unreliable after the erase operations performed on the block reaches a certain threshold.

To address the above characteristics of flash memory technologies but also maintain the compatibility with software system using conventional HDDs, flash-based storage systems usually incorporate a flash translation layer (FTL) in their designs. The FTL virtualizes the storage space to the software systems using logical block addresses (LBAs). The FTL will dynamically map LBAs to physical page numbers (PPNs) where the SSD actually stores data in flash arrays. Since programming a page or erasing a block takes significantly longer time than reading, the FTL performs out-of-place updates for each write/program requests and changes the mapping of logical and physical block addresses. As a result, the FTL also needs to perform garbage collection algorithms to reclaim pages and blocks that stale data occupy.

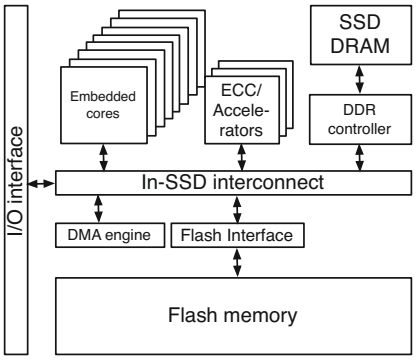


Fig. 1. The architecture of conventional SSDs

As program-erase cycles of flash blocks limit the lifetime of SSDs, the system design should minimize the number of programs and erases to maximize the lifetime. Therefore, most FTL also combines wear-leveling algorithms during garbage collections to identify hot/cold data and evenly distribute program-erase cycles of flash blocks. Furthermore, the system can apply techniques, including compression and deduplication, to reduce the number of programs in the SSD.

### B. Modern flash-based SSDs

Figure 1 depicts the architecture of a modern high-end SSD that data center servers may use. In addition to the flash arrays for data storage, a modern data center SSD can contain a system-on-chip with the I/O interface, several microprocessor cores, hardware accelerators, DMA engines, flash interfaces, and DRAM interface as the controller. The SSD also contains DRAM chips for buffering DMA data. The SSD usually organizes flash arrays into multiple channels to allow parallel accesses and provide abundant internal access bandwidth between data arrays and the SoC-based controller.

The embedded cores inside the SoC will execute firmware programs to parse commands from the I/O interface and perform FTL functions. To support the execution of firmware programs, these cores may leverage the on-die SRAM or utilize the DRAM chips inside the SSD. The firmware programs also use in-SSD DRAM as write buffers or data caches to further improve the access performance. The SoC controller also equips with accelerators to reduce compute latencies for ECC or encryption.

To communicate with the host computer, the SSD wires the I/O interface to PCIe, M.2, SATA slots. The I/O interface interprets signals coming from the host computer using either SATA or NVMe [1] standard. These standards include general commands including data reads, data writes as well as administrative commands that allows the host system to query or adjust the status of the SSD. These commands represent data locations on the SSD using logical block addresses that the host software infrastructure (e.g., the file system and the device driver) abstracts the storage array in the device. As a result, the storage device is completely agnostic to the content of data. In this project, we implemented the prototype SSD using the NVMe standard since NVMe is specifically designed to address the demands of SSDs and generally deliver better performance comparing with other counter-parts.

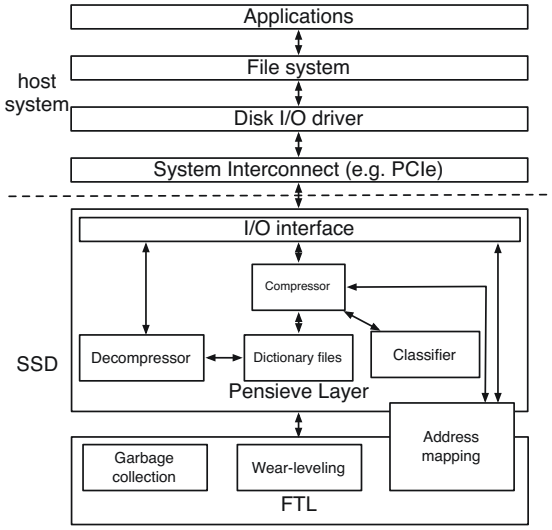


Fig. 2. The system architecture of Pensieve

## III. Overview of Pensieve

Pensieve provides a low-overhead, machine learning assisted layer in the SSD to extend the device lifetime. As Pensieve works within the device, Pensieve is transparent to the host system. Pensieve does not require any changes in system software stack, applications and I/O protocols. Pensieve simply requires changes in the FTL of a modern SSD. Therefore, existing SSDs can adopt the Pensieve FTL design without additional hardware costs. With Pensieve categorizes similar data, the SSD can potentially achieve both effects of compression and deduplication without an additional pass of storage data.

Figure 2 places Pensieve in the system architecture. Pensieve interacts with the host system through a standard disk I/O interface and the FTL of an SSD. Pensieve contains four main components, the classifier, a set of dictionary files, the compressor, and the decompressor. The classifier contains a trained prediction model to categorize incoming data into different compression classes. Each class will have a corresponding dictionary file associated with it. The compressor can compress data using a specified dictionary file. In the case of reading data, the decompressor can extract compressed data using the specified dictionary file.

Upon receiving a write command from the I/O interface, the classifier of Pensieve looks over a small part of the writing data and uses the prediction model to decide the compression class. For classes that are not compressible or have limited benefits with compression, the SSD will bypass the compression to save computation resources. For data belong to the same class, the SSD can assign the same dictionary file if the SSD selects a dictionary-based compression scheme for this class. The compressor will then take the whole writing data and compress the content using the designated dictionary file. Since Pensieve potentially changes the data size, Pensieve also needs to pass the compressed data size and the compression class to the FTL. The FTL also needs to keep this information in the mapping table to locate storage data correctly.

Unlike conventional file-based data compression mechanisms, Pensieve does not consider each file as a single unit of compression. Instead, Pensieve classifies all storage data into several different categories and all data within the same category can use a shared dictionary file. Therefore, Pensieve can achieve data compression while avoiding duplication of dictionary contents.

If the SSD receives a read command, Pensieve works with the FTL to obtain the compression class and the physical locations from the mapping table. The decompressor reconstructs the original data content using the compression algorithm for the specified class as well as the designated dictionary file in the memory buffer. Pensieve will then send the content back to the host computer when the request data is ready.

Though Pensieve adds additional features to compress data in the SSD efficiently, Pensieve does not need additional information from the host system. Therefore, Pensieve is compatible with existing SSD I/O interfaces (e.g., NVMe/SATA).

## IV. Pensieve's Classifier

Pensieve relies on the built-in classifier to categorize incoming data to decide the compressibility of data and avoid duplication in dictionary entries without using hints from the host software. Therefore, the accuracy and the efficiency of the classification model will affect the success of Pensieve.

Pensieve applies two machine learning approaches, agglomerative clustering and random forest, to generate the data classification model. We use agglomerative clustering to group data that are highly similar and figure out the optimal number of classes in our resulting model. We will then tag each data item with the group number that agglomerative clustering produced as the input of random forest algorithm to train the desired classification model.

To build the classification model, we collected real disk contents from a set of daily used Linux machines running Ubuntu 14.04. This collection of data contains 800000 files ranging from text files, program binaries, images, videos, program sources, etc. In the following paragraphs, we will present the details and our considerations for building the classification model in Pensieve.

### A. Data clustering

Pensieve uses the compression class to decide the compressibility of incoming data and the dictionary file to use. Therefore, the prediction model in Pensieve needs first to identify these groups. Because the file information is lost in block device layer I/O commands, using the high-level file information (e.g., suffixes of files) is not feasible. The classification model must be able to identify the type of data by directly looking into the content without any hint from the rest of the system.

In this work, we use agglomerative clustering algorithm to cluster sample files, instead of error-prone, time-consuming human-based labeling approaches on the huge amount of files. For each input file, the compression program uses dictionary-based algorithm that converts every two bytes of data into a 16-bit unsigned integer, and counts the occurrences of each integer to generate an optimal dictionary. The clustering algorithm compares the edit distances of identical symbols

| | average latency | accuracy |
|---|---|---|
| AdaBoost | 3241 $\mu$s | 86.56% |
| Decision Tree | 4.6 $\mu$s | 82.58% |
| Random Forest | 13.5 $\mu$s | 87.16% |
| SVC | 68619 $\mu$s | 60.40% |
| NuSVC | 74917 $\mu$s | 60.21% |
| Linear SVC | 220 $\mu$s | 63.82% |

TABLE I
THE PERFORMANCE OF EVALAUTED CLASSIFICATION ALGORITHMS

in the resulting dictionary files to determine the appropriate clustering.

To decide the number of classes that can generate the best result, we change the target number of groups for each run of the agglomerative clustering algorithm and feed the training model with the complete content from each file that we collected as inputs. For each clustering result, we compute the overall compression rate, including the overhead of dictionary files, for the training dataset.

We tested the compressed data sizes with cluster sizes between 4 and 128. The result shows that 64 clusters will deliver the optimal data size. The clustering algorithm also successfully categorizes uncompressible data types (e.g., jpeg files, mpeg files) into the same classes. Therefore, the resulting model can also use as a predictor whether if the incoming data is compressible or not.

### B. Data classifier

To classify incoming data efficiently, Pensieve's classification model considers two factors – the execution time of making a prediction and the number of bytes that the model needs to process to make an accurate prediction. The resulting Pensieve's classification model uses Random Forest algorithm and simply needs to read 512-byte data from each write request. This section will describe the design decisions we made for Pensieve's classification model.

*1) Classification algorithms:* Pensieve aims at using idle controller cores for desired prediction and compression. The classification model should make a prediction within the latency of writing a flash page so that the firmware program can use pipeline parallelism to hide the latency of prediction in the worst case. In modern flash memory chips, the minimum page program latency is around 200 $\mu$s. Therefore, we target at a classification algorithm that can deliver reasonable performance within 200 $\mu$s.

We evaluated six different classification algorithms: AdaBoost, Decision Tree, Random Forest, C-Support Vector Classification (SVC), Nu-Support Vector Classification (NuSVC) and Linear Support Vector Classification (Linear SVC). We implemented these classification algorithms on the ARM-based controller that Section V-A describes and randomly chose 10000 files as the training data set.

Table IV-B1 lists the performance of these classification algorithms. We validate the accuracy using files from the collected Linux files, excluding the training data set. Among these algorithms, Random Forest delivers the best prediction accuracy and also finishes within the desired latency. Therefore, Pensieve uses Random Forest as the default classifier in the rest part of this paper.

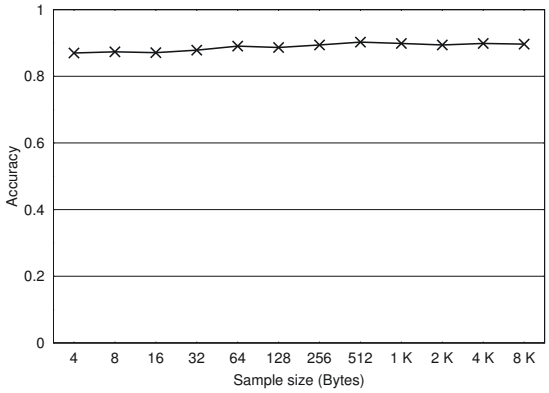


Fig. 3. The accuracy of classifier using different number of bytes from the beginning of data chunks

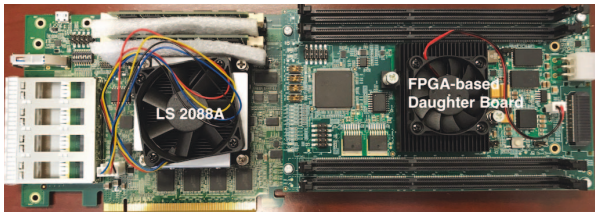


Fig. 4. The hardware of a Pensieve-compliant SSD

*2) Input length for each prediction:* Pensieve also aims at predicting compression groups using the minimum of bytes from each chunk of data to classify incoming data. Therefore, Pensieve can start compression data content in the early phase of writing data and hide the latency of compression task through buffering and multitasking, mitigating the impact of writing to flash chips.

Figure 3 shows the classification accuracy of by randomly select $x$ bytes from each file. We changed the length of bytes ($x$) that Random Forest uses as the input of classification. We used the same training datasets and validation data as in Section IV-B1. The result shows that using 512 bytes reaches the best accuracy (90.25%). And once the length exceeds 512 bytes, the model becomes overfitting the training data and degrades the prediction accuracy. In the rest of the paper, we use 512 as the default number of bytes that Pensieve's classifier needs to make decisions.

## V. THE IMPLEMENTATION OF PENSIEVE

Pensieve leverages existing hardware components in modern SSDs. Therefore, modern SSDs can integrate Pensieve by just modifying firmware programs running on general-purpose processor cores, without increasing the hardware cost. To demonstrate the feasibility of Pensieve on modern SSDs, we developed a prototype SSD with Pensieve. This section will describe the design of our prototype SSD as well as the required firmware modifications.

### A. The hardware architecture of the prototype SSD

Figure 4 shows the hardware of our prototype Pensieve-compliant SSD. This prototype SSD contains an NXP LS2088A board and an FPGA-based daughter board. This

assembly resembles existing SSD architectures that Figure 1 illustrates.

The LS2088A itself is a PCIe expansion card equipped with a system-on-chip (SoC) that consists of eight ARM cores and a set of hardware accelerators. These ARM cores can execute firmware code for NVMe interface and FTL functions, including address mapping, garbage collection and wear-leveling algorithms that modern SSDs need to perform. Similar to modern SSDs that use hardware accelerators to calculate error-correction code for stored data, the Pensieve-compliant SSD also leverages these hardware accelerators for the same purpose. Since the ARM cores on this SSD provide better computation capabilities than existing SSD controllers, we change the frequencies and the number of cores the prototype platform use during experiments to align with the architecture of modern commercially available data-center SSDs.

The FPGA-based daughter board connects to the LS2088A through an on-board interconnect that offers bandwidth up to 7.88GB/sec. This daughter board contains an FPGA and a few memory slots. These memory slots host custom-built NAND flash-based DIMMs using MT29F256G08CMCABH2 MLC chips. The FPGA acts as the flash memory controller that receives commands from the LS2088A and translates those commands into DDR3 signals.

### B. The firmware of Pensieve-compliant SSDs

In addition to the conventional FTL functions, Pensieve's firmware programs also implement the classifier, the compressor and the decompressor. As Pensieve changes the size of data, the firmware programs also need to support Pensieve in the mapping table for LBAs and PBAs. Though Pensieve requires changes to the FTL firmware, Pensieve does not propose any change to the I/O interface protocols (e.g., NVMe or SATA). The rest of this section will describe those parts of firmware programs that are different from conventional SSD firmware programs.

*1) Compressor:* When the I/O interface receives a write command, the I/O interface forwards this command to the compressor. The compressor works with the classifier to determine the compression class. The compressor will then fetch the incoming data and compresses data using the compression class that the classifier determined. The compressor also works with the FTL to allocate space for the compressed data.

To compress incoming data, the compressor locates the dictionary file that the classifier determined and compresses data using that dictionary file. As most dictionaries only need a few MBs to store and the current design of Pensieve uses only 64 clusters, the compressor maintains a cached version of these dictionaries using the SSD DRAM space. After the compressor finishes compressing data, the compressor will pass the LBA, the length of source data, the length of compressed data and a pointer to the compressed data in the SSD DRAM buffer. The compressor discards the raw data upon the completion of compression since the SSD does not need that anymore. Though our classification model rarely mispredicts, in case the length exceeds the original data size, the compressor will pass the uncompressed data and discard the compressed data instead.

Since the effect of compression is more significant when the granularity of data is larger and to simplify the design of the mapping table, the compressor works together with the I/O interface and maintains several data buffers in SSD DRAM. Each buffer stores a fixed size chunk with the starting logical block address aligned with the chunk size (e.g., If the chunk size is 1 MB, the starting address of each buffered data will align with 1 MB). The compressor will notify the FTL to map the LBAs of buffered data blocks to the DRAM locations.

As soon as the FTL mapping for the buffered data completes, the compressor can allow the I/O interface to respond to the host system for the completion of the write commands like how conventional SSDs handle writes. In this way, the host system can consider the request is successful and make forward progress for the application.

In case that the compression class for the buffered data belongs to a category that is compressible and can use a shared dictionary, the compressor can start compressing the buffered data since the shared dictionary approach does not need the whole unit to create a dictionary file. Otherwise, when the buffered data reach the predefined chunk size or the SSD needs to flush the buffer, the compressor will compress the whole chunk of data. The compressor can then update the mapping information with the FTL after data compression finishes.

*2) Classifier:* The classifier in the Pensieve-compliant SSD interacts with the compressor and categorizes the incoming data. The classifier receives DRAM buffer addresses from the compressor returns the categorization result.

Instead of scanning the whole buffered data, the classifier simply needs to use part of the data in the DMA buffer to classify the compression class. As modern file systems (e.g., ext4) tends to allocate logical block addresses consecutively for the same file, sampling part of the incoming data can achieve high accuracy in terms of predicting the data type. The algorithm and the model that the classifier implements follow the result that Section IV presents. Since the classifier only needs the beginning part of the buffered data, the classifier can decide the compression class whenever any core is free, before the buffer is full.

*3) Addressing Mapping:* Unlike the conventional flash storage system that each physical NAND flash page contains the consistent number of logical data blocks, flash pages in Pensieve-compliant SSD can contain variable numbers of logical blocks. As a result, the FTL in a Pensieve-compliant SSD would need to modify the design of the mapping table between LBAs and PBAs accommodate this difference. Except for the address mapping, most FTL features, including garbage collection and wear-leveling, could remain the same as in the FTL of modern SSDs.

The FTL of the prototype Pensieve-compliant SSD uses the "chunk id", which is the starting LBA of the compressed data divided by the chunk size, as the index of the mapping table. Each entry in the mapping table records the physical pages of the compressed data in sequential order and the compression class of the chunk. This design maintains the same table size as the FTL firmware programs we examined in conventional SSDs. For a 1 TB SSD, this table requires at most 1GB of space to maintain the mapping table. Similar to conventional SSDs, the Pensieve-compliant SSD also periodically backs up the mapping table in the SSD.

Figure 5 illustrates an example of mapping table. Since the starting LBA of each chunk aligns with the predefined

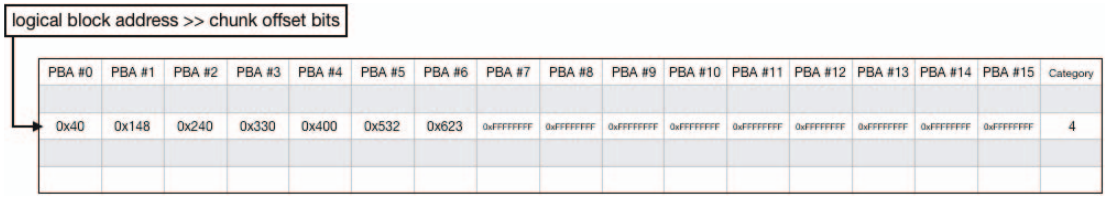


Fig. 5. An exemplary mapping table design of a Pensieve-compliant SSD

chunk size, the FTL simply uses the LBA shifted with chunk offset bits as the index to access the table. The indexed entry will contain the current mapping information of the requesting LBA. For each entry, we reserve the space to map up to $\frac{chunk\_size}{flash\_page\_size}$ cells for physical page locations so that the FTL always has sufficient cells to map data. If the compressed data does not use all reserved cells for physical locations, where is common in most cases of Pensieve, the FTL marked these empty cells in the entry as unused (e.g., fill with all 1s). The mapping table also needs several bits to keep track of the dictionary for decompression as in the category field.

*4) Decompressor:* When the I/O interface receives a read command, the I/O interface works with the decompressor to handle the request. The decompressor queries the FTL to fetch the compressed chunk containing the requesting data from the flash storage array. The decompressor will use the compression class of the chunk and the corresponding dictionary file to decompress the chunk into the SSD DRAM buffer. When the decompression finishes, the I/O interface can then initiate DMA data transfers and complete the requests.

Pensieve compresses data from fixed size chunks in raw, uncompressed data. Therefore, a read request may require the decompressor to fetch every flash page that belongs to the same chunk and hurt the latency of a single read command. To mitigate the latency degradation, the decompressor can response to the read request as soon as the decompression reaches the offset of the read command.

Our Pensieve firmware program will keep decompressed data in the SSD DRAM buffer until the SSD DRAM buffer management policy (e.g., FIFO in our implementation) needs to reclaim the buffer space. Therefore, if the workload exhibits reasonable spatial locality, Pensieve can achieve the effect of prefetching data since the workload tends to request for the rest of the compressed data within a short period of time.

## VI. Results

This section will present the performance of the prototype SSD with Pensieve. This paper evalautes the Pensieve design by building a machine with the prototype SSD and executing a set of applications on the machine. The host machine uses a quad-core Intel Kaby Lake processor clocked at 3.5GHz. The machine contains 8 GB DRAM modules as the physical main memory to host a Linux system with 4.10.0 kernel version. The linux uses ext4 file system for all disk partitions.

### A. Design space exploration of Pensieve

The chunk size and the buffer size are two factors that affect the performance and the design of the mapping table and DRAM buffers in an SSD with Pensieve. This section examines the impact of these two parameters and determines the values of these parameters for our prototype SSD.

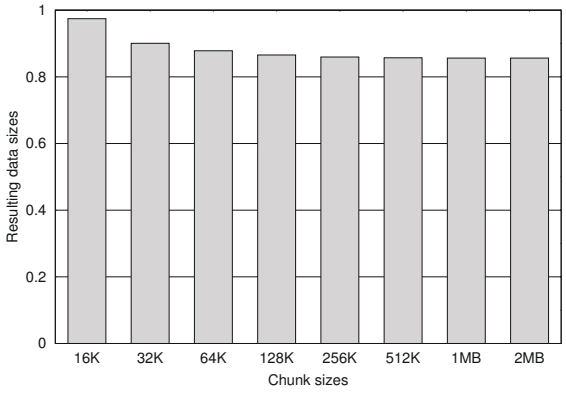


Fig. 6. The data compression rate and the chunk size in SSD buffer.

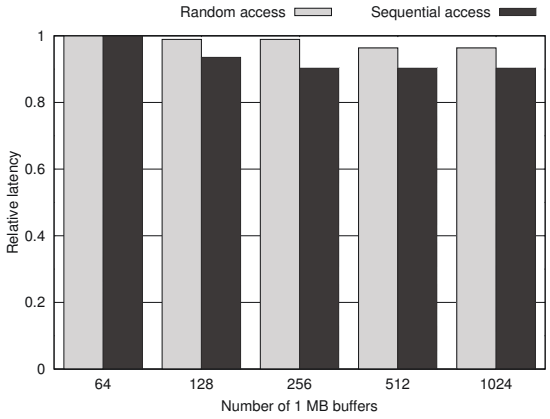


Fig. 7. The relative latency of our SSD under different buffer sizes

Figure 6 presents compressed data size using various chunk sizes. We found that compression rate does not show significant change for chunk sizes larger than 1 MB.

Figure 7 depicts the performance of both sequential and random 4K read requests using different number of buffers with 1 MB in each buffer. We use 64 $\times$ 1 MB buffers as the baseline. For sequential access, we found that the SSD can achieve optimal performance with 512 $\times$ 1 MB buffers. Increasing the buffer size to 1024 $\times$ 1 MB does not help improve the sequential access performance. For random accesses, the buffer size does not significantly improve the the performance. Therefore, we use 512 MB as the default buffer size for the rest of our experiements.

### B. Write reductions

In this sets of experiments, we evaluate the number of program operations for storing different sets of data. These

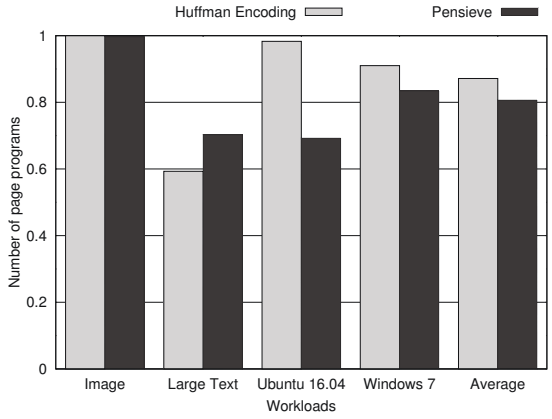


Fig. 8. The relative number of page programs of using Pensieve and conventional compression scheme

workloads include compressed images, large text files, files from a daily used Ubuntu 16.04 machine and files from a daily used Windows 7 machine.

Figure 8 the relative number of program operations of each workload with the baseline configuration that does not employee any data compression. With Pensieve's ML-assisted layer, we reduce program operations by 19% on average.

To demonstrate the effect of sharing dictionary entries that Pensieve's design enables, we also compare Pensieve with regular Huffman encoding using the same chunk size. Pensieve incurs 7% fewer program operations than conventional dictionary-based encoding that requires a dictionary file assoicated with each data chunk. In addition, for workloads that contain large amount of uncompressible files (e.g., image), Pensieve does not have to always compress data to test if data compression helps.

### C. Synthetic Workload

In this sets of experiments, we use filebench [29] to generate synthetic workloads from a different set of files that we collected on another Linux server to estimate the performance of real usage of Pensieve. For this workload, we assume 80% of reads and 20% of writes. We also assume 20% of these requests are random, and the rest are sequential. Each request size ranges from 4 KB to 4 MB. The whole dataset would occupy 90% of our storage space. average lantecy, Pensieve maintains an average latency of 107$\mu$s, only 13.4% slower than the baseline that does not use any data compression without our current implementation relying on general-purpose processor cores.

## VII. Related work

As compression can reduce the amount of writing data, improve the cost per unit capacity and the lifetime of flash-based SSDs, existing research projects applied compression in several ways. Similar to Pensieve, zFTL [26], CaFTL [3], FlaZ [23], Nitro [20], RCFFS [14], Delta-FTL [32], ZBD [24] as well as FCL for SmartMedia [33] all perform compression in the SSD address mapping layer. Compression is transparent to the user program and the file system in these designs. However, without the assistance of the machine-learning based prediction as Pensieve, these systems always need to compress data and compare the data size with original input, significantly increasing the overhead of handling writes. On the other hand, Pensieve does not even try to compress data predicted as inappropriate for compression. In addition, these works constructs dictionaries for different data chunks independently, thus there will have duplication issue in dictionaries that Pensieve avoids.

Deduplication is another approach that can help reduce the amount of storage data, previous research projects also demonstrate the deduplication can help improve the SSD life time [5], [9], [10], [12], [15], [16], [22], [28]. However, these approaches usually require complex computation to achieve better results. To optimize writes performance, these deduplication mechanisms are usually decoupled from regular write operations and data compression, leading to additional writes. Though the effect of deduplication in Pensieve cannot compete with these dedicated deduplication algorithms, Pensieve's approach inherently achieves some deduplication effects without additional computation.

Pensieve implements the whole framework using general-purpose processor cores already presented in modern SSDs. To accelerate the performance, SSD controllers can use hardware accelerators to further improve compression and decompression performance [17], [19], [34]. However, as we demonstrated in this paper, even with general-purpose cores, the proposed algorithm of Pensieve can still achieve competitive performance.

Many file systems, including, NTFS [27], JFFS2 [31], LeCramFS [13] and SquashFS [2], allows users to turn file compression features on and save the required space on the storage device. However, these file system-level compressing mechanisms consume precious CPU resources. Though the file system level compression can leverage content-related information, none of the above applied similar mechanisms that Pensieve uses machine-learning assisted approach to avoid potential duplication in dictionaries.

To more effectively compress data, some applications store data in compression, deduplication friendly data formats [4], [11] or even store data in compressed forms [21]. Kothiyal et. al., also studied the effect of compression algorithms for various file types [18]. Pensieve requires no modifications in applications and relies on no hints from applications. Even if we use Pensieve-compliant SSDs, Pensieve's model can avoid the inflation of storage data due to redundant compressions applied.

As the advancement of microarchitecture and process technologies, modern SSD controllers become more powerful to improve the performance of storage systems without increasing the burden of the host processor. Recent research projects show that using the idle processor cores on the SSD controller can potentially accelerate general-purpose computing [6], [8], [30]. Instead of offloading computation, Pensieve only uses those general-purpose cores to predict and compress data. Therefore, Pensieve provides another approach to reclaim the wasted processing power on the SSD controller without any programmer's effort.

## VIII. Conclusion

This paper presents Pensieve to demonstrate the potential of applying machine-learning techniques in the FTL to improve

the lifetime and capacities of SSDs. As machine learning techniques allow the FTL to predict storage contexts without hints from the software layer accurately, the FTL can still obtain information missed in the storage protocol stack. Pensieve leverages this advantage from machine learning techniques to classify the storage data, enable more efficient data compression as well as naturally reduce duplication of compression dictionaries.

We also implement a prototype Pensieve-compliant SSD using commercially available parts that resemble the architecture of modern SSDs. The experimental results show that Pensieve successfully reduces the number of program operations while maintaining competitive performance.

## ACKNOWLEDGMENTS

This research was supported in part by NSF grant CNS-1657039. This paper also would like to thank Dragan Savic of Dell EMC for his support in developing the prototype for Pensieve and NVIDIA Corporation for the K40 and Quadro P5000 GPU used for this research.

## REFERENCES

[1] AMBER HUFFMAN. NVM Express Revision 1.1. http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf, 2012.

[2] ARTEMIY I. PAVLOV AND MARCO CECCHETTI. SquashFS HOWTO. http://www.tldp.org/HOWTO/html_single/SquashFS-HOWTO/.

[3] CHEN, F., LUO, T., AND ZHANG, X. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association, pp. 6–6.

[4] CONSTANTINESCU, C., GLIDER, J., AND CHAMBLISS, D. Mixing deduplication and compression on active data sets. In *2011 Data Compression Conference* (March 2011), pp. 393–402.

[5] DEBNATH, B., SENGUPTA, S., AND LI, J. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 16–16.

[6] DO, J., KEE, Y.-S., PATEL, J. M., PARK, C., PARK, K., AND DEWITT, D. J. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1221–1230.

[7] GRUPP, L., CAULFIELD, A., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P., AND WOLF, J. Characterizing flash memory: Anomalies, observations, and applications. In *MICRO-42: 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (12 2009), pp. 24 –33.

[8] GU, B., YOON, A. S., BAE, D.-H., JO, I., LEE, J., YOON, J., KANG, J.-U., KWON, M., YOON, C., CHO, S., JEONG, J., AND CHANG, D. Biscuit: A framework for near-data processing of big data workloads. *SIGARCH Comput. Archit. News 44*, 3 (June 2016), 153–165.

[9] GUPTA, A., KIM, Y., AND URGAONKAR, B. Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 229–240.

[10] HA, J. Y., LEE, Y. S., AND KIM, J. S. Deduplication with block-level content-aware chunking for solid state drives (ssds). In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing* (Nov 2013), pp. 1982–1989.

[11] HARNIK, D., KHAITZIN, E., SOTNIKOV, D., AND TAHARLEV, S. A fast implementation of deflate. In *2014 Data Compression Conference* (March 2014), pp. 223–232.

[12] HUA, Y., LIU, X., AND FENG, D. Smart in-network deduplication for storage-aware sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 509–510.

[13] HYUN, S., BAHN, H., AND KOH, K. Lecramfs: an efficient compressed file system for flash-based portable consumer devices. *IEEE Transactions on Consumer Electronics 53*, 2 (May 2007), 481–488.

[14] KANG, Y., AND MILLER, E. L. Adding aggressive error correction to a high-performance compressing flash file system. In *Proceedings of the Seventh ACM International Conference on Embedded Software* (New York, NY, USA, 2009), EMSOFT '09, ACM, pp. 305–314.

[15] KIM, D., AND KANG, S. zf-ftl: A zero-free flash translation layer. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2016), SAC '16, ACM, pp. 1893–1896.

[16] KIM, J., LEE, C., LEE, S., SON, I., CHOI, J., YOON, S., U. LEE, H., KANG, S., WON, Y., AND CHA, J. Deduplication in ssds: Model and quantitative analysis. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)* (April 2012), pp. 1–12.

[17] KJELSO, M., GOOCH, M., AND JONES, S. Design and performance of a main memory hardware data compressor. In *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies* (Sep 1996), pp. 423–430.

[18] KOTHIYAL, R., TARASOV, V., SEHGAL, P., AND ZADOK, E. Energy and performance evaluation of lossless file data compression on server systems. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (New York, NY, USA, 2009), SYSTOR '09, ACM, pp. 4:1–4:12.

[19] LEE, S., PARK, J., FLEMING, K., ARVIND, AND KIM, J. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE Transactions on Consumer Electronics 57*, 4 (November 2011), 1732–1739.

[20] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized ssd cache for primary storage. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 501–512.

[21] LI, J., TSENG, H.-W., LIN, C., SWANSON, S., AND PAPAKONSTANTINOU, Y. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proceedings of the VLDB Endowment 9*, 14 (2016).

[22] MA, J., STONES, R. J., MA, Y., WANG, J., REN, J., WANG, G., AND LIU, X. Lazy exact deduplication. *Trans. Storage 13*, 2 (June 2017), 11:1–11:26.

[23] MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. Using transparent compression to improve ssd-based i/o caches. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 1–14.

[24] MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. Zbd: Using transparent compression at the block level to increase storage space efficiency. In *2010 International Workshop on Storage Network Architecture and Parallel I/Os* (May 2010), pp. 61–70.

[25] MICRON INC. MT29F384G08EBHBBJ4 Datasheet. https://www.micron.com/products/nand-flash/tlc-nand/384Gb#/, 2017.

[26] PARK, Y., AND S. KIM, J. zftl: power-efficient data compression support for nand flash-based consumer electronics devices. *IEEE Transactions on Consumer Electronics 57*, 3 (August 2011), 1148–1156.

[27] RUSSINOVICH, M., AND SOLOMON, D. A. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*, 5th ed. Microsoft Press, 2009.

[28] SEO, B. K., MAENG, S., LEE, J., AND SEO, E. Draco: A deduplicating ftl for tangible extra capacity. *IEEE Computer Architecture Letters 14*, 2 (July 2015), 123–126.

[29] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine 41*, 1 (March 2016), 6–12.

[30] TSENG, H.-W., ZHAO, Q., ZHOU, Y., GAHAGAN, M., AND SWANSON, S. Morpheus: Creating application objects efficiently for heterogeneous computing. In *43rd International Symposium on Computer Architecture* (2016), ISCA 2016.

[31] WOODHOUSE, D. JFFS: the journaling flash file system. In *Proceedings of Ottawa Linux Symposium* (2001), OLS.

[32] WU, G., AND HE, X. Delta-ftl: Improving ssd lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 253–266.

[33] YIM, K. S., BAHN, H., AND KOH, K. A flash compression layer for smartmedia card systems. *IEEE Trans. on Consum. Electron. 50*, 1 (Feb. 2004), 192–197.

[34] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (May 1977), 337–343.

[35] ZUCK, A., TOLEDO, S., SOTNIKOV, D., AND HARNIK, D. Compression and ssds: Where and how? In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)* (Broomfield, CO, 2014), USENIX Association.