HAML-SSD: A Hardware Accelerated Hotness-Aware Machine Learning based SSD Management

Bingzhe Li*, Chunhua Deng†, Jinfeng Yang‡, David Lilja‡, Bo Yuan†, and David Du*
*Department of Computer Science and Engineering, University of Minnesota
†Department of Electrical and Computer Engineering, Rutgers University
‡Department of Electrical and Computer Engineering, University of Minnesota
{lixx1743, yang3116, lilja, du}@umn.edu, chunhua.deng@rutgers.edu, bo.yuan@soe.rutgers.edu

***Abstract*—Solid state drive (SSD) as a fast storage device has been playing an important role across many applications from mobile computing to large distributed systems in recent years. However, the performance of the SSD can be degraded tremendously due to the intrinsic properties of NAND-based flash memory including limited erase cycles and asymmetric write and erase operations. Previous works separated hot/cold data into different blocks in order to improve SSD performance. "Hotness" is typically defined as the cumulative update frequencies of pages. However, we believe that an additional new parameter, average update time interval, should also be considered into the "hotness" definition associated with the update frequency. Moreover, to adaptively classify hot/cold data, a machine learning algorithm is applied to better accommodate the dynamically changed I/O access patterns of traces. In this paper, a machine learning (ML) based SSD management called HAML-SSD is proposed. The purpose of applying the ML algorithm is to dynamically cluster the data with similar "hotness" based on a new definition of "hotness". Thus, a two-dimension clustering algorithm is used for storing the pages categorized into the same cluster within the same block. Moreover, to obtain reasonable training time, a specific hardware component called HAML-unit is designed in the SSD. Finally, the experimental results indicate that the HAML-SSD decreases the response time around 26.3% - 57.7% compared to previous works with the evaluation of real traces.**

I. Introduction

With the rapid increase of data volume, the performance of storage systems becomes crucial. Solid state drive (SSD) as a fast storage device has become prevalent across various applications from mobile computing to cloud storage systems. However, due to the intrinsic properties of NAND-based flash memory, a solid state drive has a limited number of program-erase (P/E) cycles. As the technology keeps increasing the capacity density of SSDs (bits per cell), the number of P/E cycles keeps decreasing. For example, TLC-SSD (Triple-Level Cell) starts to wear out after 3K P/E cycles and QLC-SSD (Quad-Level Cell) only has 1K P/E cycles [1]. Moreover, since the write and erase operations have different granularity (pages vs. blocks), SSDs use out-of-place updates with a log-structure [2]. To release more free space, a SSD has to carry out the garbage collection to move valid pages in a block to free spaces and then erase the whole block with invalid pages. Thus, the garbage collection (GC) migrating valid pages to free spaces introduces extra writes and causes large overhead measured by the Write Amplification (WA). The large WA further aggravates the wear-leveling issue.

Some recent works [3][4][5] investigated new SSD architectures to improve the performance of SSDs. For example, Kang *et al.* [5] reduced the long-tail latency by applying reinforcement learning. Kim *et al.* [3] proposed a new architecture called autoSSD to maintain a high-level of QoS performance. However, all these works did not consider the I/O access patterns to SSDs which may cause a large write amplification. To decrease GC overhead, some researchers [6][7][8] partitioned the data according to their access patterns. The data with similar "hotness" are stored in the same blocks in order to reduce GC overhead. Most of them consider "hotness" only based on the update frequency of data pages. However, some other parameters such as time intervals between two adjacent updates should be considered into the "hotness" and have not been well investigated by the previous works. Therefore, there is an opportunity to re-define the "hotness" based on different parameters and further improve the performance of SSDs.

In this paper, a new Hotness-Aware Machine Learning based SSD management, called HAML-SSD, is proposed to reduce the overhead of GCs in SSDs. We have identified that in addition to the update frequency, the average update time interval, which is the average time interval between two adjacent updates to the same data page, is also critical to define the "hotness" of a data page. Therefore, both parameters are used for clustering data with I/O access patterns. Based on those two factors, data are scheduled to different queues. The clustering algorithm uses the K-Means algorithm to distinguish which group/queue data should be allocated to. Meanwhile, a new machine learning based hardware unit is designed to speed up the training process in the SSD. The training execution time with the machine learning hardware unit has the same order of the latency of the SSD I/O operation which is much faster than that of a general processor. In the experimental results, the HAML-SSD is capable of significantly reducing the overall response time of different traces compared to previous works while remaining a low hardware overhead.

There are fourfold contributions. **(i)** In addition to update

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

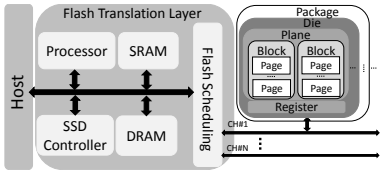


Fig. 1: SSD internal architecture

frequency, a new parameter, average update time interval, is considered into the hotness definition. **(ii)** A machine learning assisted method is proposed to cluster data with similar hotness in order to reduce the GC overhead of SSDs. **(iii)** A hardware unit called HAML-unit is designed to accelerate the machine learning algorithm in order to achieve reasonable execution time. **(iv)** We fully investigate and analyze trade-offs between different values of design parameters and provide insights for how to use the HAML-SSD in the future works.

The structure of the paper is as follows. Section II gives a description of the background of SSDs and the motivation of this work. The machine learning based algorithm and its designs are provided in Section III. Section IV shows the experimental results compared to previous works. Finally, the conclusion is described in Section V.

II. Background and Motivation

A. SSD Background

Conventional NAND-based SSDs consist of a number of main components including host interface controller, embedded processor, read/write buffer, flash memory controllers, NAND flash packages, etc. as seen in Fig. 1. The host interface controller manages the communication between the host and the storage device through a well-defined interface such as PCIe or SATA. The SSD controller is designed to manage internal data placement and implement SSD internal functions such as wear leveling and garbage collection.

SSD manufacturers typically spread NAND flash packages across several independent channels. Multiple NAND flash packages as persistent storage media are located in each channel. Each NAND flash package contains one or more dies. A single die consists of multiple planes and a number of registers for caching I/Os. Each plane contains multiple blocks. A block is the smallest unit to be erased. Each block typically consists of 128 or 256 pages. A page is the smallest unit to perform read and write/program operations. Currently, a single page has 4KB (or larger) data space and 129-byte meta-data area for error correct coding.

NAND SSDs have several unique characteristics. First, write can only change a cell from '1' to '0'. Therefore, a page cannot be directly updated before the page is erased. Since the erase operates at the granularity of a block, a whole block will be erased instead of erasing a page for the page update. To avoid the large overhead of in-place update, people use out-of-place update (log-structure) [2] and the address mapping table to record the mapping relationship between logical page number (LPN) and physical page number (PPN). Second, each block has a limited number of program/erase (P/E) cycles. Once reaching to its threshold, the block starts to wear out

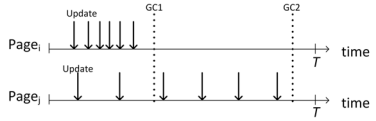
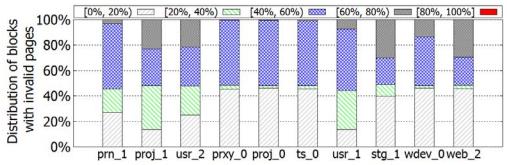
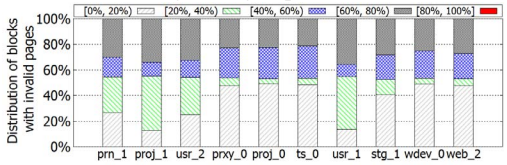


Fig. 2: Views of different GCs for hotness



(a) Not using average update time interval



(b) Using average update time interval

Fig. 3: The distribution of blocks with different percentages of invalid pages. The key [0% - 20%) indicates the blocks contain 0% - 20% invalid pages.

and lose data. To avoid the storage device reaching its lifespan too early, wear-leveling mechanism tries to evenly utilize each block to make sure all blocks reach their limited P/E cycles at roughly the same time.

Garbage collection (GC) is the process of reclaiming storage space from invalid data. As the number of invalid pages increasing, the SSD does not have enough free space to store incoming data. Thus, the SSD controller needs to trigger GC to recycle those invalid pages. During the GC process, the SSD controller will select a NAND flash block (or a group of blocks) based on a certain policy (such as the number of invalid pages or the used P/E cycles). Then, all valid pages in the erase candidate blocks are read out and are written into other free space. After doing that, the address mapping table needs to be updated accordingly and then the candidate blocks can be erased to release free space. Since garbage collection mechanism needs to reallocate all valid pages, it causes extra write operations. To calculate the overhead of extra writes, Hu *et al.* [9] defined it as write amplification (WA), which can be formulated as $WA = \frac{Writes\ \ to\ \ flash}{Writes\ \ by\ \ host}$. A large WA is introduced by the extra operations, which will degrade the SSD performance tremendously. In this paper, we use the greedy algorithm [10] to trigger the GC when the number of available blocks reduces to a threshold.

B. Motivation

SSDs need to carry out a GC to release/increase free space which may cause a large migration overhead. To mitigate the GC overhead, people want to erase the block with the maximum number of invalid pages. If all pages in one block are invalid, the WA caused by the GC equals to 1. However, in a real environment, since the data/pages are accessed randomly, it is hard to invalidate all pages in one block before erasing the block. As discussed in the introduction section, some previous works [6][8][7][11][12] investigated the data temperature (hotness) to reduce the GC overhead.

TABLE I: Pearson correlation coefficient (r) between update frequency and average update time interval

	prn_1	proj_1	usr_2	prxy_0	proj_0	usr_1	stg_1	web_2	LUN0
r	-0.03	0.06	0.05	0.17	0.14	0.03	0.07	0.18	0.04

Most of the previous works only defined the hotness by the update frequency of a data page. However, we have identified that the average update time interval ($UpdateT$) is also important. As seen in Fig. 2, both two pages (i and j) in the time period T have six updates. If GC happens at GC2, two pages will have similar temperature, while if GC happens at GC1, obviously $Page_i$ updates more frequently and thus is hotter than $Page_j$. In addition, we further investigate the effect of $UpdateT$ on the number of invalid pages in blocks. Fig. 3 presents two examples of the distributions of blocks with different invalid pages when partitioning data with/without using $UpdateT$. Comparing Fig. 3a with Fig. 3b, the partition using $UpdateT$ (Fig. 3b) has a larger number of blocks with 60% - 80% invalid pages (black bar) than that of not using $UpdateT$ (Fig. 3a). At the same time, the partition using $UpdateT$ also has a larger number of blocks with 0% - 40% invalid pages (gray and green bars). This is because two methods using the same traces will have the same number of updates. Thus, an increase of the number of blocks with large invalid pages will cause an increase of the number of blocks with small invalid pages. This is to say, using $UpdateT$ in addition to update frequency is more efficient to separate hot and cold data since some blocks stale more quickly and the other blocks stale more slowly. As a result, when GCs are triggered, using $UpdateT$ and update frequency can achieve a lower migration overhead than that of only using update frequency but not $UpdateT$. We have also calculated Pearson correlation coefficient [13] between update frequency and $UpdateT$ for multiple traces. As seen in Table I, most of traces obtain correlation coefficient r around 0, which means that these two parameters have a weak correlation relationship. Therefore, both update frequency and average update time interval should be considered into the hotness classification.

Moreover, previous works [7][11] used a constant threshold to partition data pages into hot and cold. However, they failed to consider that the access patterns of traces are dynamically changing. For example, if setting threshold to 100, page A with 5 updates and page B with 110 updates will be separated into two groups. However, under the constant threshold 100, a new page C with 99 updates will be grouped with page A instead of page B. However, it is apparent that page C should be grouped with page B with similar hotness. In addition, Fig. 4 presents one example of the distributions of $UpdateT$ for different traces in one time period. We can find that in one trace different logical addresses have different $UpdateT$ values whose range covers from 0 to T. Further, comparing among difference traces, the distributions of $UpdateT$ are also varied. Therefore, the partition based on constant thresholds cannot efficiently classify hot/cold data. So, the classification should be adaptively changed based on the I/O access patterns of traces and more categories may be needed.

Machine learning (ML) as a prevalent technology for classification is capable of dynamically classifying data access patterns. Two basic types of ML algorithms can be used for classification, supervised learning (such as k nearest neighbor and random forest) and unsupervised learning (such as K-Means and Mean-Shift). For I/O access patterns of traces in SSDs, there is no ground truth to indicate which class they should belong to. Thus, a clustering algorithm as one type of unsupervised learning methods is a good candidate for this purpose. There are several types of clustering algorithms and we investigate them by running on a server. Based on [3] and our experimental results in Table II, K-Means is simpler, faster and more flexible than other clustering algorithms and thus is used in this work. Please note that the other clustering algorithms are also compatible to the proposed algorithm but will not be implemented in this paper.

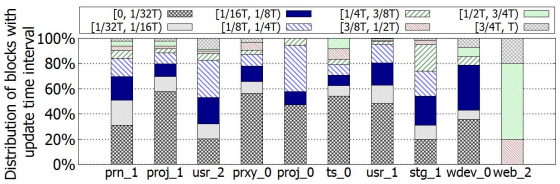


Fig. 4: Distribution of blocks with different average update time intervals. The key [0, 1/32T] indicates the blocks with average update time intervals of [0, 1/32T].

TABLE II: Execution times of clustering 1000 2-D data with different clustering algorithms running with python sklearn

	K-Means	Mean-Shift [14]	HAC [15]	DBSCAN [16]
Complexity	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$ (worst)
Execution time	8.61ms	43.4ms	20.1ms	9.52ms
HAC: Hierarchical agglomerative clustering DBSCAN: Density-based spatial clustering of applications with noise				

The K-Means clustering is used to dynamically cluster pages based on the access patterns of traces. However, one of the drawbacks is its long training time. As seen in Fig. 9, the experimental result shows that the training time of K-Means running in a server with 2.4GHz CPU and 8GB memory varies from around 0.02s to 200s. Compared to the latency (around ten to hundred microseconds) of I/O operations in SSDs, such long time latency is not acceptable since it will cause a long-tail latency and delay the whole application process. Consequently, this motivates us to design a specific ML-based hardware to speed up training time. In the following sections, we introduce how we use ML algorithm to determine the data hotness. Meanwhile, a low-cost ML hardware unit is proposed to be integrated in SSD devices. Finally, the investigation about different design parameters demonstrates the effectiveness of different parameters.

III. HAML-SSD Design

In this section, a new Hotness Aware Machine Learning based SSD management (HAML) is introduced. HAML contains two major parts, the machine learning based scheduling algorithm and the hardware implemented design.

A. Overall architecture

Fig. 5 indicates the overall architecture of the SSD flash translation layer (FTL). The **UpdateF_TBL** and **Up-**

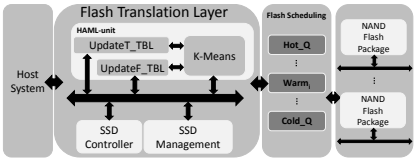


Fig. 5: SSD internal architecture of HAML-SSD design.

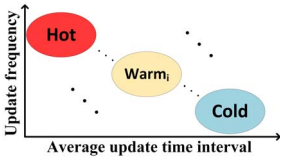


Fig. 6: Regions for K clusters of data hotness

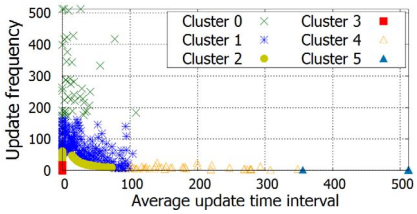


Fig. 7: An example of 6 clusters for the trace LUN0

dateT_TBL tables are used to record two parameters (update frequency and average update time interval) which are used to cluster data. K-Means unit is the hardware implementation of the K-Means algorithm. According to the clustering result, data pages are classified into different clusters (hot, ... $warm_i$, ... and cold) and the data with similar hotness are assigned to the same flash queue. The number of clusters is a design parameter and is set at the beginning of SSD design. A detailed analysis of the HAML-SSD design is provided in the following sections. Please also note that two or more data pages in the same class will have both similar update frequencies and close average update time intervals.

B. ML Allocation Policy

We first introduce the ML-based data allocation policy. Two parameters are recorded for data to indicate the data hotness in this design. The first one is the **update frequency** ($UpdateF$), which is expressed by the number of write accesses during one time period [11][12]. This is a popular parameter used in the hybrid and tiered storage systems. The other parameter is **average update time interval** ($UpdateT$) which indicates the timing relationship of updates. It is defined by the average time interval between two consecutive updates in one time period as seen in Line 20- 24 of Algorithm 1. If there is no update, $UpdateT$ equals to the time period T. Therefore, a larger update time interval value means that the page takes a longer time to be re-written/updated and thus the page has a higher probability to be valid during the garbage collection.

Basically, the accumulated information of all pages should be tracked for clustering. However, assume that the default page size is 8KB and total SSD capacity is 256GB, more than 32 million data need to be clustered. For the clustering algorithm, the execution time is proportional to the number of inputs. Therefore, with ten-million inputs the execution time will be extremely long. To reduce the execution time, Xie *et al.* [12] used a sampling method to reduce the number of inputs. But, the sampling cannot accumulate all trace information and thus cannot well represent the trace characteristics. For example, if one sample page has 1 update but other pages in the same blocks may have 100 updates, the sample page will misrepresent the other pages in the same block. In this paper, we split the SSD logical address into smaller contiguous regions called **slices**. The SSD records two parameters for each slice in each time period and uses these collected information as the clustering dataset.

With those two parameters, we use a two-dimension K-Means clustering algorithm [17] to cluster data into K clusters. Those K regions indicate the different hotness of the data pages or slices. For example, as shown in Fig. 6:

Hot region: since it has the highest update frequency and the smallest average update interval time, those data are rewritten in a very short time and also are updated frequently. Therefore, among K regions, this region is the hottest region.
Cold region: has the longest average update time interval and the smallest update frequency. Therefore, blocks in this region are rarely erased because they do not have many invalid pages.
$\mathbf{Warm_i}$ regions: The accumulated average update time intervals and update frequency in $Warm_i$ regions are in the middle of the cold and hot regions. The Warm regions may either have short update interval time but few number of updates or have a large number of updates but have a long update interval time.

As explained above, different regions have different levels of accumulated information. The K-Means algorithm automatically separates them into K clusters. One example of using 6 clusters based on the real experiment is shown in Fig. 7. Each point stands for one slice (one contiguous logical space). Cluster 0 and Cluster 5 belong to the Hot region and Cold region, respectively.

The HAML-SSD scheduling algorithm is shown in Algorithm 1. There are three major functions, HAML-SSD scheduling, monitoring and K-Means clustering. The monitoring procedure is used to record the update frequency and the average update time interval of each slice during the time period T. The collected UpdateF_TBL and UpdateT_TBL results are used for training K-Means clustering algorithm. After that, update frequency and average update time interval in the two tables are scaled to the same range [0, 512] (hardware design uses 9-bit values). Finally, the HAML-SSD scheduling assigns requests to different queues according to their classifications.

To remedy the wear-leveling, two methods are used to handle that. The first one is that when updates coming the Hot_Q always selects the free blocks with the lowest erase counts and the Cold_Q always selects the free blocks with the highest erase counts. The $Warm_i$ regions randomly select free blocks between those two extreme values. The other method is to periodically migrate between hot data in high erase-count blocks and cold data in low erase-count blocks in order that

Algorithm 1 HAML-SSD Scheduling Algorithm (K clusters)

Input: Req_i

procedure HAML-SSD SCHEDULING
  Compute slice number S_i of Req_i
  if classification[S_i] ==0 **then**
    Hot_Q $\leftarrow$ Req_i
  else if classification[S_i] == i **then**
    $Warm_i \leftarrow Req_i$
  else
    Cold_Q $\leftarrow$ Req_i
end
procedure MONITORING
  Record $T_{starttime}$
  while $t < T$ **do**
    Compute slice number S_i of Req_i
    UpdateF_TBL[S_i] +=1
    UpdateT_TBL[S_i] += $timestamp_{req_i}$ - $T_{starttime}$
    $T_{starttime}$ = $timestamp_{req_i}$
  if t == T **then**
    for $i \leq$ UpdateT_TBL.size() **do**
      if $UpdateF\_TBL[i] >= 1$ **then**
        $UpdateT\_TBL[i] = \frac{UpdateT\_TBL[i]}{UpdateF\_TBL[i]}$
      else
        UpdateT_TBL[i] = T
    K-Means(UpdateF_TBL, UpdateT_TBL)
    for $i \leq$ UpdateF_TBL.size() **do**
      UpdateF_TBL[i] = 0
      UpdateT_TBL[i] = 0
    t = 0
end
procedure K-MEANS(UpdateF_TBL, UpdateT_TBL)
  Find Max_slice in UpdateF_TBL
  Find Max_access in UpdateT_TBL
  Scale UpdateF_TBL and UpdateT_TBL to the range [0, 512]
  Training the K-Means Clustering algorithm
  For slice number S_i: classification[S_i] $\leftarrow$ K-Means results
end

all flash pages have similar erase counts.

C. ML Unit Design

K-Means algorithm [18] as one of the unsupervised learning models has been proven to be an efficient tool for clustering. However, K-Means clustering algorithm needs multiple iterations to converge, which is time-consuming. In the SSD, the controller has a limited computation capacity which results in unacceptable amount of time to finish the training process. Thus, this limitation motivates us to design a hardware unit to reduce the training time of K-Means algorithm.

Fig. 8 demonstrates the architecture of the designed hardware. The centroid values are initialized with K values in Fig. 6 (①). There is a static random access memory (SRAM) to store data in two tables to be clustered. The SRAM data is read sequentially in each iteration to calculate the Euclidean distance (②) to each of the K centroid values. After getting the distances, we compare them, find the smallest distance and ascribe the data into that cluster (③). In each iteration, we need to update the centroid values (④). This is done by accumulating the x-axis and y-axis values of each cluster and counting the number of data that belongs to the cluster. After

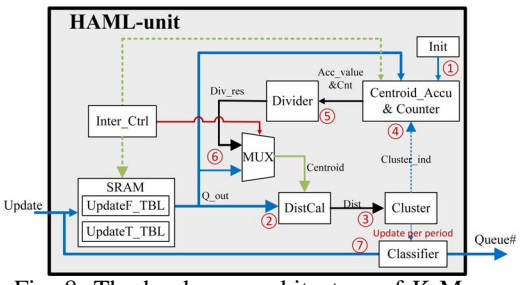


Fig. 8: The hardware architecture of K-Means.

TABLE III: SSD configuration

Parameter	Value	Parameter	Value
physical capacity	300GB	# of pages/block	256
Logical capacity	256GB	Read latency	20us
Page Size	8KB	Program latency	200us
Slice size	100MB (default)	Erase latency	2ms

that, we divide the accumulated values by the number of data to get the updated centroid values (⑤). The divider is a multi-cycle divider, which has a small area size. Since there are K clusters, and each cluster has the centroid of x-axis and y-axis that needs to be updated. Thus, it requires 2*K division operations per iteration with the time-multiplexed manner (⑥) to save area. Most of the delay in K-Means algorithm hardware unit results from distance calculating, clustering and accumulating. So, the time-multiplexed division manner does not dramatically delay the whole performance (less than 5%). Finally, after this training process, the classifier is updated. The step (⑦) is the process that the classifier determines which flash queue the upcoming update should be assigned to. Basically, the classifier can be regarded as a mapping table from LPN to queue number (similar to the mapping table from LPN to PPN) and can quickly find the corresponding queue number of each update. In summary, the main concern of K-Means algorithm is the training process. The overall hardware cost and the comparisons are provided in Section IV-B.

IV. Experimental Results

A. Environment Setup

In this design, the SSD configuration is listed in Table III. Assume the physical capacity of the SSD is 300GB and 256GB capacity is addressable by the host system, the over-provisioning factor is 17%. Given that the slice size is 100MB, the total number of slices is 2560. The experiments with varying design parameters (such as slice size and the number of clusters) are also investigated in Section IV-D to Section IV-F. A greedy garbage collection (GC) scheme is used. The GC activation threshold is set to 128 free blocks. For the performance evaluation, the SSDsim simulator [19] is used to evaluate the performance of different algorithms. Two types of real traces are used, Cambridge MSR traces [20] and Systor'17 traces [21]. The default time period is set to 6 hours which means that the K-Means cluster will be re-trained every 6 hours.

B. HAML-SSD Hardware Implementation

The hardware synthesis result is provided in this section. We use the Synopsys Design Compiler to synthesize the design

TABLE IV: Trace configurations

	Number of IOs (Millions)		Total request size (GB)	
	Write	Read	Write	Read
	MSR Cambridge Traces [20]			
prn_1	2.77	8.46	30.78	181.35
proj_1	2.50	21.14	25.58	750.36
usr_2	1.99	8.58	26.47	415.28
prxy_0	12.14	0.38	53.80	3.05
proj_0	3.70	0.53	144.27	8.97
ts_0	1.49	0.32	11.34	4.13
usr_1	3.86	41.43	56.13	2079.23
stg_1	0.80	1.40	5.99	79.52
wdev_0	0.91	0.23	7.14	2.75
web_2	0.04	5.14	0.78	262.82
	Systor'17 [21]			
LUN0	20.65	47.03	350.31	1257.44
LUN1	17.91	49.16	338.89	1456.00
LUN3	17.56	50.82	315.06	1323.51

with the TSMC 28nm technology [22]. The clock frequency is set to 1.3GHz. We compare the latency of the hardware implementation with the software implementation. For the software implementation, the K-Means clustering algorithm is implemented with C++ on a server with 2.4GHz, 24 cores, 16MB cache size and 8GB memory. The reference values of the power and area are also provided.

First, we investigate K-Means algorithm with 6 clusters, 2560 inputs and 10 iterations. As illustrated in Table V, the total execution time of the ML-unit design is only 20us which is similar to a read operation time. In other words, the execution time of ML-unit is equivalent to inserting one extra read in one time period (default 6 hours). This shows that the K-Means clustering execution time of ML-unit has little effect on the overall performance. To further shorten the training time, we can either increase transistor gate sizes (trade-offs between area and delay) or reduce the iteration number of K-Means algorithm. As illustrated in Fig. 9, the server attains 588ms execution time, which is more than 20,000X slower than that of the ML-unit. Compared to the server, the controller in the SSD is even slower and the training time may even reach to seconds. Thus, if without offloading the K-Means training to a specific hardware, the ML training process will delay the original operations and degrade the SSD performance significantly. Compared to the chip size [23], the area of HAML-unit is only 0.037% of the chip size. Compared to the most recent Samsung SSD product [24], the HAML-unit only costs 0.385% of the whole power.

Moreover, we investigate execution time with different numbers of clusters (3 to 8) and inputs (640 to 25,600) as seen in Fig. 9. The maximum execution time of HAML-SSD with 25,600 inputs is only 197us which is similar to inserting one write operation in one time period (default 6 hours), while the server needs 201.7s. Obviously, the delay of 200s for any operation in SSD is not acceptable. In summary, the hardware overhead of HAML-unit occupies less than 0.5% of the whole chip hardware cost and the proposed HAML-SSD obtains reasonable execution time for the K-Means algorithm.

C. Overall Performance Comparison

We compare HAML-SSD with three previous works. The HAML-SSD uses 6 clusters which is indicated by HAML-6.

TABLE V: Hardware implementation comparisons (K-Means with 6 clusters, 2560 inputs and 10 iterations)

	Total execution time (us)	Area (mm^2)	Power (mW)
HAML-SSD	19.8	0.036	22
Server system	588,484	-	-
V-NAND Flash [23]	-	97.6	-
Samsung SSD PRO970 [24]	-	-	5200
HAML / others	0.0033%	0.037%	0.423%

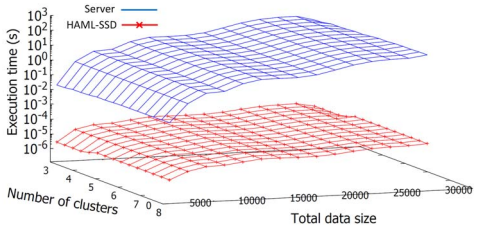


Fig. 9: Execution time comparison with different number of clusters and inputs

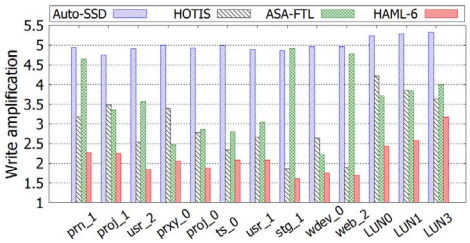


Fig. 10: Write amplification for different workloads

TABLE VI: Total number of block erases (GCs) for different workloads

	Auto-SSD [3]	ASA-FTL [12]	HOTIS [11]	HAML-6
prn_1	5.7E+05	3.7E+05	5.4E+05	2.6E+05
proj_1	3.5E+05	2.4E+05	2.4E+05	1.6E+05
usr_2	6.5E+05	3.4E+05	4.7E+05	2.4E+05
prxy_0	6.5E+05	4.4E+05	3.2E+05	2.7E+05
proj_0	5.4E+06	2.9E+06	2.9E+06	1.9E+06
ts_0	2.3E+05	1.1E+05	1.3E+05	9.6E+04
usr_1	1.5E+06	8.2E+05	9.3E+05	6.4E+05
stg_1	1.2E+05	4.8E+04	1.3E+05	4.1E+04
wdev_0	1.3E+05	6.5E+04	5.5E+04	4.3E+04
web_2	1.0E+04	4.1E+03	9.7E+03	3.6E+03
LUN0	6.8E+06	5.4E+06	4.8E+06	3.1E+06
LUN1	5.9E+06	4.3E+06	4.3E+06	2.9E+06
LUN3	6.2E+06	4.2E+06	4.7E+06	3.7E+06

Auto-SSD [3] is one of the state-of-art works which splits applications into different queues. HOTIS [11] considers the long interval hot data and distinguishes between hot and cold data based on constant threshold values with two clusters. ASA-FTL [12] uses the K-Means clustering to distinguish between hot and cold data only based on the update frequency with K=3. To trigger garbage collections, we precondition the SSD with sequentially writing the whole drive one time and then replaying the first half of one trace each time. We repeat this process for different traces. To evaluate the SSD performance, the results are obtained based on the second half of traces. The precondition setup is used for the evaluation of the following sections as well.

As seen in Fig. 10, the proposed HAML-6 has the least WA values among all traces. On average, the proposed HAML-SSD has the WA value about 2.1292 while the Auto-SSD,

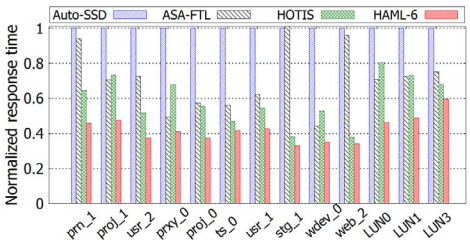


Fig. 11: Normalized response time for different workloads

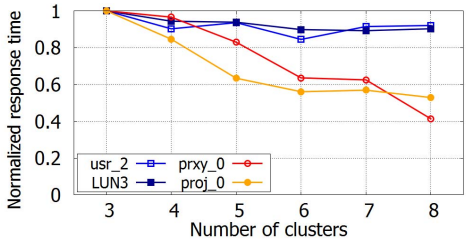


Fig. 12: Normalized response time with varying number of clusters

HOTIS and ASA-FTL have the WAs of 5.002, 2.959 and 3.553, respectively. Among all traces, HAML-SSD decreases WAs about 13% - 57.4% compared to previous works, because the HAML-SSD method uses two dimension parameters to dynamically cluster data. The classification of hot/cold data in HAML-SSD performs more precisely than others. As a result, the erased blocks have fewer valid pages than other three algorithms and thus the HAML-SSD migrates fewer valid pages, which leads to the lower WA values. In addition, we investigate the number of erases for all traces. As seen in Table VI, on average, the proposed HAML-SSD can reduce the number of erases by 30.4% - 53% compared to the Auto-SSD, HOTIS and ASA-FTL, respectively. The lower erase count of HAML-SSD is because it has the lower WA value and then results in less data rewriting.

Finally, we simulate the normalized response time based on SSDsim simulator [25] and the SSD configuration in Table III. Fig. 11 shows that the proposed HAML-SSD reduces the response time about 57.7%, 37.1% and 26.3% on average compared to the Auto-SSD, ASA-FTL and HOTIS, respectively.

D. Varying Number of Clusters

In this section, we investigate the effect of the number of clusters on the performance with the proposed HAML-SSD scheme. The number of clusters is varied from 3 to 8. The slice size is set to 100MB and $T = 6h$. Based on the results in Fig. 12 (only show four typical traces), all traces can be roughly categorized into two groups. For the first group (usr_2 and LUN3), the performance of traces is not sensitive to the number of clusters and the response time is slightly decreased with the increase of the number of clusters. For the second group (prxy_0 and proj_0), as increasing the number of clusters the performance has significant improvement. The performance improvement from 3 clusters to 8 clusters in the second group can reach to 2.4X. The reason is that for traces with different accumulated distributions, a larger number of clusters has a higher ability to classify a lower number of cluster distribution. In other words, if one trace has a 3-cluster distribution, algorithms with more than 3 clusters are also able to classify the 3 clusters and thus obtain similar results with that of a 3-cluster algorithm. In contrast, if one trace follows an 8-cluster distribution, algorithms with less than 8 clusters will not be able to classify the 8 clusters efficiently. According to the synthesis results in Section IV-B, the execution time is not changed much as increasing the number of clusters. Therefore, without any relevant information of traces the number of clusters should be set to a relatively large value.

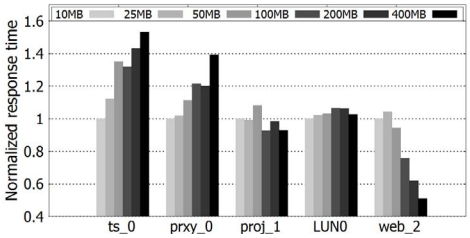


Fig. 13: Normalized response time with varying slice size

E. Performance with Varying Slice Size

The effect of the slice size on the performance is investigated in this subsection. The slice size varies from 10MB to 400MB. Thus, the total number of inputs for K-Means is from 640 to 25600. The number of clusters is set to 6 and $T = 6h$. We select five typical traces as shown in Fig. 13. Basically, the traces belong to three groups according to their performance as varying the slice size. For the first group such as ts_0 and prxy_0, the performance increases with the decrease of the slice size. The reason is that these traces have obvious difference in small slice sizes and thus the small slice sizes provide more accurate accumulation results. For the second group of traces such as proj_1 and LUN0, those traces are not sensitive to the slice size and thus the performance is similar among different slice sizes. Their small variance might result from the little different accumulation. For the last group of trace which contains only one trace (web_2), the performance increases as increasing the slice size. The reason is that this trace does not have enough update information for small slice sizes in one time period. As shown in Table IV, web_2 only has 40000 write I/Os. Therefore, for the large slice sizes, each slice can accumulate enough information to accurately represent the characteristics of access patterns of traces. In summary, considering the overhead of K-Means training and hardware cost as discussed in Section IV-B, the execution time of K-Means clustering is proportional to the number of inputs, which is determined by the slice size. As accumulating enough trace information, smaller slice sizes might provide better I/O performance but cause longer execution time. So, people need to carefully select a proper slice size to balance the I/O performance and K-Means execution time.

F. Performance with Varying T

We investigate the effect of the time period T on performance with varying from 1 to 12 hours. The number of clusters

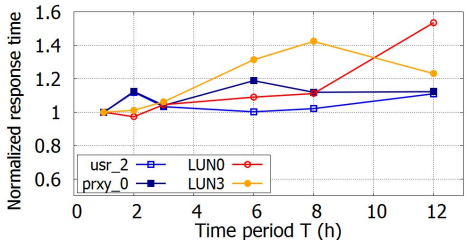


Fig. 14: Normalized response time with varying T

is set to 6 and the slice size is set to 100MB. Four typical traces are used as shown in Fig. 14. Based on the results, the traces can be categorized into two groups. For the traces in first group such as usr_2 and prxy_0, the normalized response time is not sensitive to T. The variance between different T is smaller than 20%. The reason is that those traces have similar update patterns for different T. As a result, no matter what T is the HAML-SSD algorithm obtains similar performance. For the traces in the other group such as LUN0 and LUN3, they obtain obvious difference for different T. The maximum difference can reach to more than 40% as seen in Fig. 14. The reason is that the accumulated access patterns are significantly changed for different T. The different distributions cause different clustering results and thus lead to the performance difference. The overall trend is that the normalized response time is increased as increasing T. This is because the shorter time period is capable of tuning the clustering algorithm in time if the access patterns of traces have a dramatic change. In summary, a shorter time period will cause a larger number of clustering training and thus there is also a trade-off between the time period and clustering execution time.

V. Conclusion

In this paper, we proposed a HAML-SSD management to improve the garbage collection efficiency. There are two parameters considered to cluster data, update frequency and update time interval. By using two-dimension K-Means clustering algorithm with k=6, HAML-SSD can efficiently classify data into different clusters. Moreover, to speed up the machine learning algorithm, a specific HAML-unit is designed in the SSD and the hardware overhead only occupies less than 0.5% hardware cost. The experimental results indicate that the HAML-SSD reduces the response time around 26.3% - 57.7% compared to previous works with running the real traces. Finally, our investigation about different design parameters can help people deeply understand design trade-offs.

Acknowledgment

This work was partially supported by NSF I/UCRC Center Research in Intelligent Storage, member companies and the following National Science Foundation grant no. 1439662, 1525617, 1536447, 1708886, 1763008, 1812537, IIP-1439622, CCF-1854742, 1815699.

References

[1] Micron. https://www.micron.com/products/advanced-solutions/qlc-nand.

[2] Peter Desnoyers. Analytic modeling of ssd write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 12. ACM, 2012.

[3] Bryan S Kim, et al. Autossd: an autonomic ssd architecture. In *2018 USENIX Annual Technical Conference*, pages 677–690. USENIX Association, 2018.

[4] Chun-yi Liu, et al. Pen: design and evaluation of partial-erase for 3d nand-based high density ssds. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 67–82. USENIX Association, 2018.

[5] Wonkyung Kang et al. Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in ssd. In *Proceedings of the 55th Annual Design Automation Conference*, page 8. ACM, 2018.

[6] Dongchul Park et al. Hot data identification for flash-based storage systems using multiple bloom filters. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–11. IEEE, 2011.

[7] Ilhoon Shin. Hot/cold clustering for page mapping in nand flash memory. *IEEE Transactions on Consumer Electronics*, 57(4), 2011.

[8] Chundong Wang et al. Adapt: Efficient workload-sensitive flash management based on adaptation, prediction and aggregation. In *MSST*, pages 1–12, 2012.

[9] Xiao-Yu Hu, et al. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 10. ACM, 2009.

[10] Werner Bux et al. Performance of greedy garbage collection in flash-based solid-state drives. *Performance Evaluation*, 67(11):1172–1186, 2010.

[11] Junqing Gu, et al. Hotis: A hot data identification scheme to optimize garbage collection of ssds. In *Ubiquitous Computing and Communications, 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on*, pages 331–3317. IEEE, 2017.

[12] Wei Xie, et al. Asa-ftl: An adaptive separation aware flash translation layer for solid state drives. *Parallel Computing*, 61:3–17, 2017.

[13] Karl Pearson. *Mathematical contributions to the theory of evolution*, volume 13. Dulau and co., 1904.

[14] Yizong Cheng. Mean shift, mode seeking, and clustering. *IEEE transactions on pattern analysis and machine intelligence*, 17(8):790–799, 1995.

[15] Daniel Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[16] Martin Ester, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[17] John A Hartigan et al. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[18] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[19] Yang Hu, et al. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, 2013.

[20] Dushyanth Narayanan, et al. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[21] Chunghan Lee, et al. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, page 13. ACM, 2017.

[22] Tsmc 28nm technology. https://www.tsmc.com/english/dedicatedFoundry/technology/28nm.htm.

[23] Dongku Kang, et al. 7.1 256gb 3b/cell v-nand flash memory with 48 stacked wl layers. In *Solid-State Circuits Conference (ISSCC), 2016 IEEE International*, pages 130–131. IEEE, 2016.

[24] Samsung v-nand ssd 970 pro data sheet. https://www.samsung.com/semiconductor/global.semi.static/Samsung_NVMe_SSD_970_PRO_Data_Sheet_Rev.1.0.pdf.

[25] Yang Hu, et al. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107. ACM, 2011.