Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters

Dongchul Park and David H.C. Du
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455, USA
Email: {park, du}@cs.umn.edu

***Abstract*—Hot data identification can be applied to a variety of fields. Particularly in flash memory, it has a critical impact on its performance (due to a garbage collection) as well as its life span (due to a wear leveling). Although the hot data identification is an issue of paramount importance in flash memory, little investigation has been made. Moreover, all existing schemes focus almost exclusively on a *frequency* viewpoint. However, *recency* also must be considered equally with the frequency for effective hot data identification. In this paper, we propose a novel hot data identification scheme adopting multiple bloom filters to efficiently capture finer-grained recency as well as frequency. In addition to this scheme, we propose a Window-based Direct Address Counting (WDAC) algorithm to approximate an ideal hot data identification as our baseline. Unlike the existing baseline algorithm that cannot appropriately capture recency information due to its exponential batch decay, our WDAC algorithm, using a sliding window concept, can capture very fine-grained recency information. Our experimental evaluation with diverse realistic workloads including real SSD traces demonstrates that our multiple bloom filter-based scheme outperforms the state-of-the-art scheme. In particular, ours not only consumes 50% less memory and requires less computational overhead up to 58%, but also improves its performance up to 65%.**

***Index Terms*—Hot Data Idenficiation, Hot and Cold Data, Bloom Filter, Flash Memory, SSD, WDAC**

I. Introduction

A tendency toward flash-based Solid State Drives (SSD) now holds sway even in the enterprise servers as well as in personal computers, especially, laptops. This trend of the storage world results from the recent technological breakthroughs in flash memory and dramatic reduction of its price [1].

Flash memory is organized in units of blocks and pages. Each block consists of a fixed number (32 or 64) of pages. Reads and writes in flash memory are performed on a page basis, while erases operate on a block basis. Its most distinguishing feature is that it does not allow in-place updates. That is, the flash memory system cannot overwrite new data into existing data. Instead, the new data are written to clean spaces somewhere (i.e., out-of-place updates), and then the old data are invalidated for reclamation in the future.

In order to resolve this in-place update issue, Flash Translation Layer (FTL) has been developed and deployed to flash memory to emulate in-place update like block devices [2], [3], [4], [5], [6], [7]. This layer enables users to utilize the flash memory like disks or main memory on top of existing conventional file systems without significant modification by hiding the characteristics of the out-of-place update. As time goes on, this out-of-place update inevitably causes the coexistence of numerous invalid (i.e., outdated) and valid data. In order to reclaim the spaces occupied by the invalid data, a new recycling policy is required, which is a so-called garbage collection. However, the garbage collection can give rise to not only considerable amount of valid data copies to other clean spaces, but also data erases to reclaim the invalidated data spaces. Data erase (1,500μs) is the most expensive operation in flash memory compared to data read (25μs) and write (200μs) [8]. Consequently, a garbage collection results in a significant performance overhead as well as unpredictable operational latency. Flash memory exhibits another limitation: cells can be erased for a limited number of times (e.g., 10K-100K). Thus, frequent erase operations reduce the lifetime of the flash memory, which causes a wear leveling issue. The objective of wear leveling is to improve flash lifetime by evenly distributing cell erases over the entire flash memory [8]. Both the wear leveling and garbage collection are fundamentally based on the hot data identification.

We can simply classify the frequently accessed data as hot data. Otherwise, they are regarded as cold data. This definition is still vague and takes only *frequency* (i.e., the number of appearance) into account. However, there is another important factor–*recency* (i.e., closeness to the present)–to identify hot data. In general, many access patterns in workloads exhibit high temporal localities [9]; therefore, recently accessed data are more likely to be accessed again in near future. This is the rationale for including the recency factor in hot data classification.

The definition of hot data can be different for each application and also can be applied to a variety of fields. First of all, this can be primarily used in data caching [10], [11]. By caching these hot data in the memory space in advance, we can significantly improve system performance. It is also applied to B-tree indexing in sensor networks [12]. FlashDB is a flash-based B-tree index structure optimized for sensor networks. In FlashDB, the B-tree node can be stored either in read-optimized mode or in write-optimized mode, whose decision can be easily made on the basis of a hot data identification algorithm. Flash memory adopts this classification algorithm particularly for a garbage collection and a wear leveling [13], [14], [15]. We can perform a garbage collection more efficiently by collecting and storing hot data to the same block, which can reduce the garbage collection overhead. Moreover, we also improve flash reliability by allocating hot data to the flash blocks with low erase count. Hybrid SSD is another good application of hot data identification [16], [17]. We can store hot data to SLC (Single-Level Cell) flash memory, while cold data can be stored to MLC (Multi-Level Cell) part in the SLC-MLC hybrid SSD. In addition to these, hot data identification has a big potential to be exploited by many other applications. In this paper, we focus on flash-based applications; thus, we consider only write accesses, not read accesses.

Flash memory contains a fixed small amount of memory (SRAM) inside; so we need to exploit the SRAM efficiently. Since FTL needs the majority of this memory for a more efficient address translation, we have to minimize this memory usage for the hot data identification. This is one of the challenges to design an efficient hot data identification scheme. Moreover, computational overhead is another important issue since it has to be triggered whenever every write request is issued.

Although this hot data identification is an issue of paramount importance in flash memory, it has been least investigated. Existing schemes either suffer from large memory space requirements [18] or incur huge computational overhead [19]. To overcome these problems, Hsieh et al. recently proposed a multiple hash function framework [20] to identify hot data. This scheme adopts multiple hash functions and a counting bloom filter to capture a frequency. Although this approach accurately captures frequency information because it maintains counters, it cannot appropriately capture recency information due to its exponential batch decay process (i.e., to decreases all counter values by a half at a time). While investigating other hot data identification schemes, we also posed a question about the existing baseline algorithm (i.e., direct address method in [20]): it cannot properly capture recency either. The direct address method assumes that unlimited memory space is available to keep track of hot data. It maintains a counter for each LBA to store access counts and periodically decays all LBA counting information thereby dividing by two at once. However, this algorithm also still retains the same limitation as the multihash function scheme.

Considering these observations, an efficient hot data identification scheme has to meet the following requirements: **1)** effective capture of recency information as well as frequency information, **2)** small memory consumption, and **3)** low computational overhead. Based on these requirements, in this paper, we propose a novel hot data identification scheme based on multiple bloom filters. The key idea of this scheme is that each bloom filter has a different weight and recency coverage so that it can capture finer-grained recency information.

Whenever a write request is issued, the hash values of the LBA are recorded into one of multiple bloom filters (for short, BFs) in a round robin fashion. All information of each BF will be periodically erased by turns in order to maintain fine-grained recency information, which corresponds to an aging mechanism. Thus, each BF retains a different recency coverage. We also dynamically assign a different recency weight to each BF: the BF that just erased (i.e., reset BF) has higher recency weight, while the lowest recency weight is assigned to the BF that will be erased in right next turn because this BF has stored LBA access information for the longest period of time. For frequency, our proposed scheme does not maintain a specific counter for all LBAs; instead, the number of BF recording the LBA information can exhibit its frequency information. The main contributions of this paper are as follows:

- ***An Efficient Hot Data Identification Scheme:*** A bloom filter can provide computational and space efficiency. Both multihash scheme and our proposed scheme try to take advantage of the bloom filter. Unlike the former using one counting bloom filter, the latter adopts multiple bloom filters. The multiple bloom filters enable our proposed scheme to capture finer-grained recency information so that we can achieve more accurate hot data classification. Multiple and smaller bloom filters empower our scheme to require not only less memory space, but also lower computational overhead.
- ***A More Reasonable Baseline Algorithm:*** Our proposed approximation algorithm named Window-based Direct Address Counting (WDAC) adopts a window that is a conceptual buffer with a predefined size to store each coming request. Each LBA maintains a corresponding access counter. Whenever a write request is issued, the LBA is stored in the head of the window and the oldest one is evicted like a FIFO (First In First Out) queue. WDAC assigns different recency weights to all LBAs in the window according to the closeness to the present. Thus, when a new request arrives, all LBAs are shifted toward the tail of the window and all their recency values are reevaluated. Consequently, WDAC can catch precise (very fine-grained) recency as well as frequency.

The remainder of this paper is organized as follows. Section II gives an overview of flash memory and bloom filters. It also describes existing hot data identification schemes. Section III explains the design and operations of our proposed hot identification scheme and WDAC scheme. Section IV provides a variety of our experimental results and analyses. Finally, Section V concludes the discussion.

II. Background and Related Work

In this section, we explain flash memory characteristics and introduce the existing hot data identification schemes with their pros and cons.

A. The Characteristics of Flash Memory

Figure 1 describes a typical system architecture of flash memory-based storage systems. Both Memory Technology

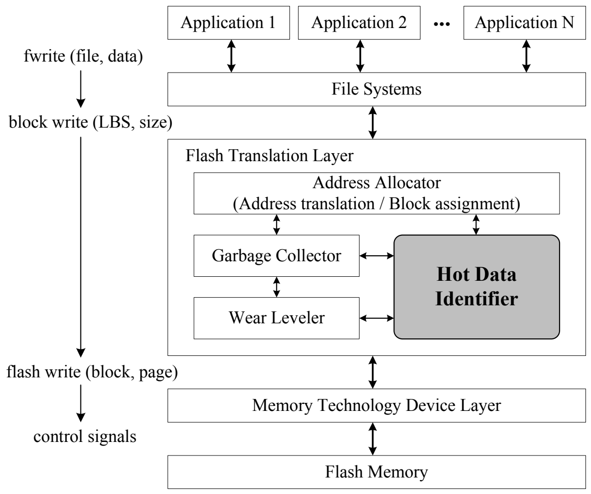


Fig. 1. Typical System Architecture of Flash Memory-based Storage Systems

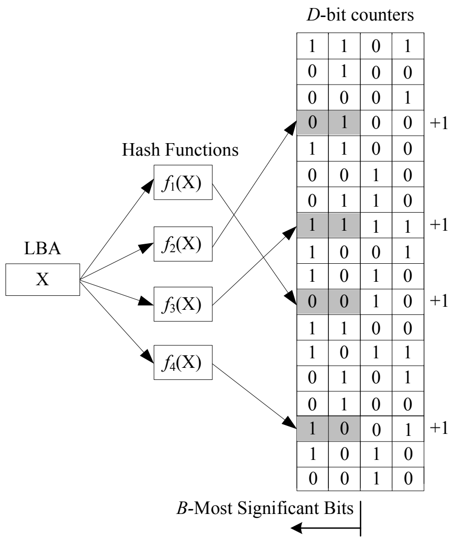


Fig. 2. Multiple Hash Function Framework. Here, D = 4 and B = 2.

Device (MTD) and Flash Translation Layer (FTL) are two major parts of flash memory architecture. The MTD provides primitive flash operations such as read, write, and erase. The FTL plays a role in address translation between Logical Block Address (LBA) and its Physical Block Address (PBA) so that users can utilize the flash memory with existing conventional file systems without significant modification. A typical FTL is largely composed of an address allocator and a cleaner. The address allocator deals with address translation and the cleaner tries to collect blocks filled with invalid data pages to reclaim them for its near future use [21]. This garbage collection is performed on a block basis; so all valid pages in the victim block must be copied to other clean spaces before the victim block is to be erased. Another crucial issue in flash memory is wear leveling. The motivation of wear leveling is to prevent any cold data from staying at any block for a long period of time. Its main goal is to minimize the variance among erase count values for each block so that the life span of flash memory is to be maximized [22], [23], [13]. Currently, the most common allowable number of write per block is typically 10K for MLC and 100K for SLC [24].

B. Bloom Filters

The main goal of the bloom filter (for short, BF) is to probabilistically test set membership with a space efficient data structure [25]. Since the space efficiency is an important factor for the BF, the correctness can be sacrificed in order to maximize it. Thus, although a given key is not in the set, a BF may provide a wrong positive answer called a *false positive*. However, the basic BFs never provide a false negative. We can also adjust design parameters (i.e., BF size (M), the number of hash function (K) and the number of unique element (N)) of a BF to allow a very low probability of the false positive.

The BF is a bit array of M bits and all bits are initially set to 0. In addition to this bit array, there must also be K independent hash functions, each of which maps the given elements to the corresponding bit positions of the array thereby setting them all to 1. To insert an element to the BF, the key value of the element is fed to the K hash functions so that we can get K hash values which correspond to the bit array positions. Then all the K bit positions are set to 1. To execute a membership query (i.e., check if the element is in the set), we first need to get the K bit positions by feeding the key value of the element to all K hash functions. If any of the K bits are 0, this means the element is not in the set because all the bits would have been set to 1 when the element was fed. If all are 1, there exist two possible cases: the corresponding element is in the set, or it is just a false positive due to the insertions of other elements. Assuming the probability of a false positive is very low, the answer to the query is positive.

C. Hot and Cold Data Identification Schemes

In this subsection, we describe the existing hot and cold data classification schemes and explore their advantages and disadvantages.

Chiang et al. [18] proposed an out-of-update scheme called Flash Memory Server (FMS) in order to reduce the number of erase operations in flash memory. They made an attempt to classify data into three types for an efficient garbage collection: read-only, hot and cold data. Although FMS exhibits a good hot and cold data classification performance, it requires a large amount of memory space since it must keep the last access time information of all LBAs.

To resolve this limitation, Chang et al. [19] used a two-level LRU list that consists of a hot LBA list and a candidate list. Both lists are of fixed size and operate under LRU. Whenever a write request comes into the FTL driver, if the corresponding LBA already exists in the hot list, the data are classified as hot data. Otherwise, it is defined as cold data. If the LBA is not in the hot list but in the candidate list, the data are promoted to the hot list. If the data are not in either list, it will be inserted into the candidate list. This two-level LRU scheme consumes less memory than FMS; nevertheless, it incurs other problems. The performance of hot data identification is totally dependent on the sizes of both lists. In other words, a small hot list can save memory space, but its performance decreases because highly likely hot data may be demoted to the candidate list or even evicted from the candidate list. Moreover, this scheme requires high computing overheads to emulate LRU discipline.

Recently, Hsieh et al. [20] proposed a multiple hash function framework to identify hot data (Figure 2). This adopts multi-hash functions and one BF with a D-bit counter for each bit position in the BF to capture the frequency of the data access by incrementing the corresponding counters. The recency of the hot data is implemented by dividing the counter value by two periodically. If any one bit of B-most significant bits is set to 1, this bit position is considered as 1. Similarly if all K-bit positions (from the K-hash functions) are 1, the data are classified as hot. For example, as shown in Figure 2, assuming this scheme adopts 4-bit counters and 2-most significant bits, 4 will be its hot threshold value. Thus, the access counter values of corresponding positions are all greater than or equal to 4, the data in the LBA are identified as hot. After a specified time period, this scheme decreases all the counter values by a half with 1-bit right shifting to implement a decay effect. Compared to the other schemes, this achieves relatively less memory consumption as well as less computing overhead. However, it does not appropriately catch recency information due to its exponential decrement of all LBA counters.

III. Multiple Bloom Filter-based Hot Data Identification Scheme

This section describes our proposed multiple bloom filter-based hot data identification scheme (Section III-A) and our Window-based Direct Address Counting (WDAC) scheme as our baseline algorithm (Section III-B).

A. The Framework

As shown in Figure 3, our scheme adopts a set of V independent bloom filters (for short, BFs) and K independent hash functions to capture both frequency and finer-grained recency. Each BF consists of M bits to record K hash values. The basic operation is simple: whenever a write request is issued to the Flash Translation Layer (FTL), the corresponding LBA is hashed by the K hash functions. The output values of each hash function ranges from 1 to M, and each hash value corresponds to a bit position of the M-bit BF respectively. Thus, K hash values set the corresponding K bits in the first BF to 1. When the next write request comes in, our scheme chooses the next BF in a round robin fashion to record its hash values. In addition, it periodically selects one BF in a round robin manner and erases all information in that BF to reflect a decay effect.

• ***Frequency:*** Unlike the multihash function framework adopting 4-bit counters, we do not maintain the specific BF counters for each LBA to count the number of appearance. Instead, our scheme investigates multiple BFs to check if each BF has

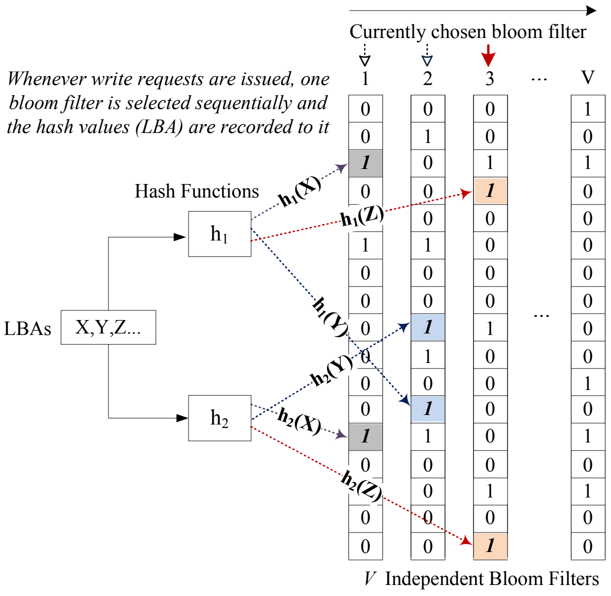
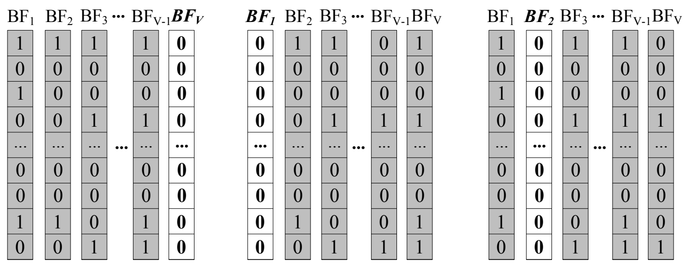

Currently chosen bloom filter
Whenever write requests are issued, one bloom filter is selected sequentially and the hash values (LBA) are recorded to it
1
2
3
...
V
Hash Functions
h1
h2
LBAs
X,Y,Z...
h1(X)
h1(Z)
h1(Y)
h2(Y)
h2(X)
h2(Z)
V Independent Bloom Filters

Fig. 3. Our Framework and Its Operations

recorded the corresponding LBA. The number of BF retaining the LBAs can show its frequency.

For precise frequency capturing, when it chooses one of the V BFs and marks the corresponding bits to 1, if the selected BF has already recorded the hash values of the LBA, it sequentially (in a round robin manner) examines other BFs until it finds a new one that has not recorded the LBA. This sequential examination minimizes disruption of our recency analysis. If it finds such a new one, it records the hash values to the BF. If it turns out that all (or predefined number of) BFs have already contained the LBA information, our scheme simply defines the data as hot and skips its further processes such as a BF checking or a recency weight assignment since this will be definitely over the threshold. This shortcut decision reduces its overhead (this will be demonstrated in our experiment section). Consequently, assuming the hash values of an LBA appear in r $(0 \le r \le V)$ numbers of the BFs out of V BFs, we can say the corresponding LBA has appeared r times before. On the other hand, if all bloom filters have already recorded the LBA, we do not know its precise frequency number from then on. However, for hot data identification in our scheme, the information required is simply the fact if the value r is larger than the threshold, not the precise counter value. If this r is larger than a threshold, it passes the frequency check.

• ***Recency:*** Even though the total access counters of two LBAs are equivalent in a period, the one accessed heavily in the further past and has never been accessed afterwards should be classified differently from the other one heavily accessed recently. If a hot data identifier depends only on counter values (frequency information), it cannot make distinction between them. Even though it has a decay mechanism, it cannot classify them well if the identifier captures coarse-grained recency.



BF1 BF2 BF3 ··· BFV-1 BFV
BF1 BF2 BF3 ··· BFV-1 BFV
BF1 BF2 BF3 ··· BFV-1 BFV

(a) First T Interval (b) Second T Interval (c) Third T Interval

Fig. 4. Our Aging Mechanism. Here, a white (i.e., not-shaded) bloom filter corresponds to a reset bloom filter

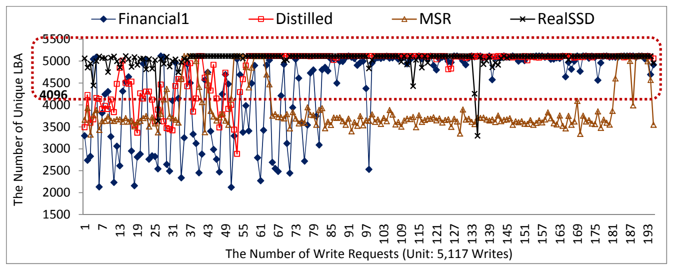
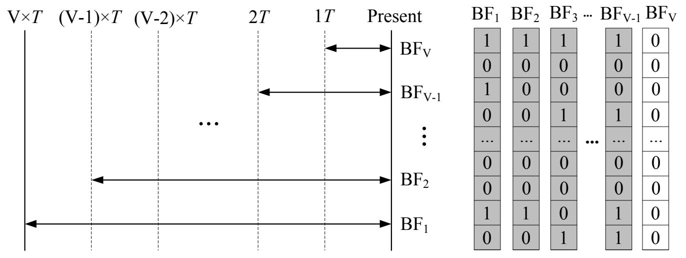


Fig. 6. The Number of Unique LBAs within Unit Periods under Various Workloads (Unit: 5,117 Writes). A large number of unique LBAs exceed the bloom filter size of 4,096 (a dotted square).



(a) Recency Coverage (b) Bloom Filter Status

Fig. 5. Recency Coverage for Each Bloom Filter

Since our scheme does not maintain LBA counters, we need to devise a different aging mechanism to capture recency information. Figure 4 illustrates our aging mechanism to decay old information. Consider that we adopt V independent BFs and the hash values of LBAs are recorded to each BF in a round-robin manner during a predefined interval T. An interval T represents a fixed number of consecutive write requests, not the time interval. As shown in Figure 4 (a), after the interval T, the BF that has not been selected in the longest time interval is selected and all bits in the BF (i.e., BF_V) are reset to 0. As soon as it is reset, the hashed LBA values start to be recorded again to all BFs including the reset BF. After the interval T, the next BF (BF_1) is selected and all the bits are reset (shown in Figure 4 (b)). Similarly, after the next T interval, the next BF (BF_2) is chosen in a right cyclic shift manner and all information is erased (shown in Figure 4 (c)) as time goes on.

Figure 5 shows the recency coverage after the interval T as soon as (BF_V) is reset. The reset BF (BF_V) can remember LBA information accessed during only the last one interval T (i.e., latest $1T$ interval). The previously reset BF (BF_{V-1}) can record the LBA information accessed during the last two intervals. Similarly, the BF 1 (BF_1) which will be chosen as a next reset BF after this period can cover the longest interval $V \times T$. This means BF_1 records all LBA information for the last $V \times T$ intervals.

Our proposed scheme assigns a different recency weight to each BF so that recency value is combined with frequency value for hot data decision. The reset BF (BF_V) records most recent access information; so highest recency weight has to be assigned to it, whereas lowest recency weight is allotted to the BF that will be chosen as a next reset BF (BF_1) because it has recorded the access information for the longest time. We intend to use recency value as a weight to the frequency value such that a final value combining both can be produced. Consequently, even though two different LBAs have appeared once in BF_V and BF_1 respectively, both frequency values are regarded differently as follows: assuming BF_V is a current reset BF, we first choose one BF ($BF_{\lfloor V/2 \rfloor}$) that has medium recency coverage and then assign a weight value of 1 to it. Next, we assign a double weight value (this is explained in subsection III-B) to the reset BF (BF_V), which means the LBA appearance in this BF counts as a double frequency. Third, we calculate the difference of recency weights between each BF by $1/(V - \lfloor V/2 \rfloor)$ and assign evenly decreasing weight value from BF_{V-1} to BF_1. For example, consider we adopt 4 BFs ($V = 4$) and BF_4 is a current reset BF, we first select BF_2 and assign a weight value of 1 to it. Then, we allot a double weight value (i.e., 2) to the reset BF (BF_4). Next, we can get 0.5 ($= 1/(4-2)$) as a difference of recency weights and finally distribute different recency weight values 2, 1.5, 1, and 0.5 to BF_4, BF_3, BF_2, and BF_1 respectively. Here, if any LBA appears both in BF_3 and BF_4, the original frequency value would be 2, but our scheme regards the final value as 3.5 because each BF has different recency weights. In this paper, we define this combination value of frequency and recency as a *hot data index*. Thus, if the value of hot data index is greater than or equal to a predefined threshold value, we regard the data as hot.

• ***A Bloom Filter Size and Decay Period:*** According to [26], we can estimate an optimal BF size (M) using the following formula, $M = KN/ln2$, given K and N. It is clear that the BF size (M) needs to be quite large compared to the size of the element set (N). For example, assuming we adopt two hash functions ($K = 2$) and there are 4,096 unique elements, we should adopt an 11,819-bit vector as an appropriate size of the BF to minimize a false positive probability. This basic BF allows the elements to be only added to it, but not removed from it. Thus, as the number of input elements grows, the data recorded in the BF are also continuously accumulated, which causes a higher false positive probability. To reduce this, the traditional BF scheme requires a large BF size as described in the formula. However, we cannot simply apply this relationship to both our proposed scheme and multihash function scheme. The BFs in both schemes retain a distinguishing feature from the basic BF–information in the bloom filters can be removed. Especially our proposed scheme completely removes all information in one of the V BFs periodically. This prevents continuous data accumulation to the BF. Therefore, we can adopt a smaller BF than the aforementioned traditional BF.

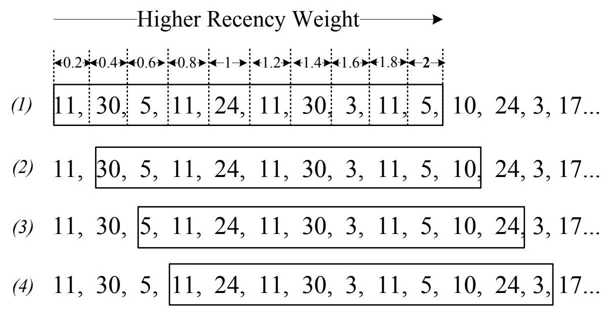
Multihash function scheme consists of 4,096 entries of a BF each of which is composed of a 4-bit counter. It also adopts 5,117 write requests (N) as its decay period. This is based on their expectation that the number of hash table entry (M) can accommodate all those LBAs which correspond to cold data within every N (where, $N \leq M/(1-R)$) write request (here, R is the hot ratio of a workload and the authors assumed R is 20%) [20]. To verify their assumption, as displayed in Figure 6, we measured the number of unique LBA for every 5,117 write request under several real traces such as Financial1, Distilled, MSR and RealSSD (these traces will be explained in our experiment section). Figure 6 clearly shows that the number of unique LBA very frequently exceeds their BF size (4,096) for each unit period (every 5,117 write request) under all four traces. This necessarily causes hash collisions so that it results in a higher false positive probability.

We also measured the average number of unique LBA within both 2,048 and 4,096 write request periods under the same workloads. They vary from 1,548 to 1,991 for 2,048 unit and from 3,014 to 3,900 for 4,096 unit. This is closely related with average hot ratios of each workload: intuitively the higher average hot ratio, the less number of unique LBA. Based on these observations, as our decay interval T, we adopt M/V numbers of write requests, where M is a BF size and V is the number of BF. To reduce the probability of a false positive, the BF size (i.e., the number of hash table entries) must be able to accommodate at least all unique LBAs that came in for the last $V \times T$ interval. Thus, whenever M/V numbers of write requests are issued, one of the BFs are selected in a round robin fashion and all the bits in the BF are reset to 0. For example, assuming our BF size is 2,048, our decay period corresponds to 512 write requests. Since memory consumption is also very important factor, we adopt even less (a half) size of BF than the BF size in multihash scheme.

B. WDAC: A Window-based Direct Address Counting

Hsieh et al. proposed an approximated hot data identification algorithm named a direct address method (hereafter, we refer to this as DAM) as their baseline. By using counters DAM can capture the frequency information well, whereas, with respect to recency, it retains the same limitation as the multihash function scheme: all LBAs accessed within the same decay period have an identical recency regardless of their access time or sequence. For instance, assuming the decay period is 4,096 write requests, the LBA accessed in the first request within this period is considered as having the same recency as the LBA accessed in the last ($4,096^{th}$) request.

To resolve this limitation, in this subsection, we propose a more reasonable baseline algorithm to approximate ideal



(a) LBAs and Sliding Window

(1)

LBA	HDI
11	4.0
30	1.8
5	2.6
24	1.0
3	1.6

(2)

LBA	HDI
11	3.2
30	1.4
5	2.2
24	0.8
3	1.4
10	2.0

(3)

LBA	HDI
11	2.6
30	1.0
5	1.8
24	2.6
3	1.2
10	1.8

(4)

LBA	HDI
11	2.0
30	0.8
5	1.4
24	2.2
3	3.0
10	1.6

(b) Total Hot Data Index for Each LBA

Fig. 7. Working Process of WDAC Algorithm. Here, the window size is 10. HDI corresponds to the total hot data index value.

hot data identification named Window-based Direct Address Counting (WDAC). As shown in Figure 7 (a), WDAC maintains a specific size of buffer like a sliding window. In addition, it maintains total hot data index values for each LBA (shown in Figure 7 (b)).

Within this window, all elements have a different recency value according to their access sequences: the closer to the present, the higher recency weight is assigned to the LBA. That is, highest recency value (i.e., 2) is assigned to the most recently accessed element (head) in the window, whereas, the lowest recency is allotted to the last element (tail) in the window. All intermediate LBAs in the window have all different recency values with an evenly decreasing manner. Assuming the window size is W, we can get the recency difference between two adjacent elements in the window as $2/W$. Thus, all recency weights assigned to each LBA evenly decrease by $2/W$ from the head to the tail in the window. Whenever a new LBA comes in, all recency values are reassigned to all the LBAs shifted in the window and the last one is evicted from the window.

Like our proposed multiple bloom filter-based scheme, as our highest recency weight, we choose 2 (a double weight value) since the total average of all recency values in the window is equivalent to that within the same decay period in the DAM. Instead, unlike the DAM, we assign a higher recency weight to the recently accessed LBAs and vice versa. Consequently, our proposed WDAC can properly identify hot data thereby using very fine-grained recency information.

Figure 7 illustrates a simple example for our WDAC process. This window can contain 10 LBAs and recency difference corresponds to 0.2 (= 2/10). Whenever a write request comes

TABLE I
SYSTEM PARAMETERS AND VALUES

System Parameters	MBF	MHF	WDAC	DAM
Bloom Filter Size	2^{11}	2^{12}	N/A	N/A
Number of Bloom Filter	4	1	N/A	N/A
Decay (Window Size)	2^{9}	2^{12}	2^{12}	2^{12}
Number of Hash Function	2	2	N/A	N/A
Hot Threshold	4	4	4	4
Recency Weight Difference	0.5	N/A	0.000488	N/A

in to the flash translation layer (FTL), this LBA is stored in the head of the window and the last one is evicted. At the same time, all the others are shifted toward the tail of the window and their recency values are reevaluated by decreasing their values by the recency difference (0.2). All of the total hot data index values also need to be updated accordingly. Finally, if the total hot data index value of the corresponding LBA is greater than or equal to a predefined hot threshold, we regard it as hot data.

In our proposed WDAC scheme, since window size is associated with recency and frequency, a proper window size is an important parameter to this scheme. Therefore, the impact of this window size will be discussed in our experiment part.

IV. EXPERIMENTAL RESULTS

This section provides diverse experimental results and comparative analyses.

A. Evaluation Setup

We compare our Multiple Bloom Filter-based scheme (hereafter, refer to as MBF) with three other hot data identification schemes: Multiple Hash Function scheme (hereafter, refer to as MHF) [20], Direct Address Method (refer to as DAM) [20], and our proposed baseline scheme named WDAC. We, however, focus particularly on the MHF since it is the state-of-the-art scheme. We adopt a freezing approach [20] as a solution for a counter overflow problem in MHF since it, in our experiments, showed a better performance than the other approach (i.e., exponential batch decay). DAM is an aforementioned baseline proposed in MHF scheme. Table I shows system parameters and their values. Our scheme (MBF) adopts a half size of a bloom filter as MHF because ours shows a better performance than MHF even with a smaller one. However, we also evaluate both schemes with the same size of a bloom filter for clearer understanding.

For fair evaluation, we assign the same number of write requests (4,096) for a decay interval in DAM as well as for a window size in our proposed WDAC algorithm. Moreover, we employ 4,096 write requests for the decay interval in MHF scheme to synchronize with DAM and WDAC, while we adopt 512 write requests for our decay period. We also adopt identical hash functions for both MBF and MHF schemes. Lastly, since window size is 4,096 in WDAC, there exist about 0.488×10^{-3} ($= 2/4,096$) weight difference for each element and we also assign 0.5 weight difference for each BF in MBF.

TABLE II
WORKLOAD CHARACTERISTICS

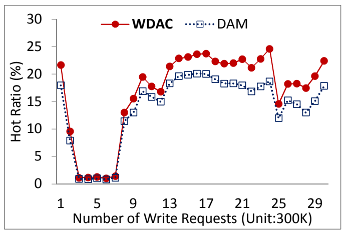
Workloads	Total Requests	Request Ratio (Read:Write)	Inter-arrival Time (Avg.)
Financial1	5,334,987	R:1,235,633(22%) W:4,099,354(78%)	8.19 ms
MSR	1,048,577	R:47,380(4.5%) W:1,001,197(95.5%)	N/A
Distilled	3,142,935	R:1,633,429(52%) W:1,509,506(48%)	32 ms
RealSSD	2,138,396	R:1,083,495(51%) W:1,054,901(49%)	492.25 ms

For more objective evaluation, we adopt four real workloads (Table II). Financial1 is a write intensive trace file from the University of Massachusetts at Amherst Storage Repository [27]. This trace file was collected from an Online Transaction Processing (OLTP) application running at a financial institution. Distilled trace file [19] shows a general and personal usage patterns in a laptop such as web surfing, watching movies, playing games, documentation work, etc. This is from the flash memory research group repository at National Taiwan University, and since the MHF scheme uses only this trace file, we also adopt this trace for fair evaluation. We also adopt MSR trace file made up of 1-week block I/O traces of enterprise servers at Microsoft Research Cambridge Lab [28]. We select, in particular, *prxy volume 0* trace since it exhibits a write intensive workload [29]. Lastly, we employ a real Solid State Drive (SSD) trace file that is 1-month block I/O traces of a desktop computer (AMD X2 3800+, 2G RAM, Windows XP Pro) in our lab (hereafter, refer to as RealSSD trace file). We installed Micron's C200 SSD (30G, SATA) to the computer and collected personal traces such as computer programming, running simulations, documentation work, web surfing, watching movies etc.

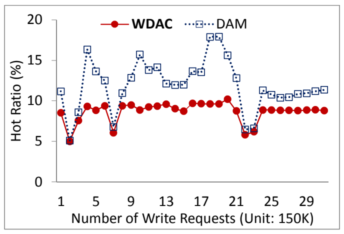
The total requests in Table II correspond to the total number of read and write requests in each trace. These requests can also be subdivided into several or more sub-requests with respect to LBA accessed. For example, let us consider such a write request as *WRITE 100, 5*. This means 'write data into 5 consecutive LBAs from the LBA 100'. In this case, we regard this request as 5 write requests in our experiments.

B. Performance Metrics

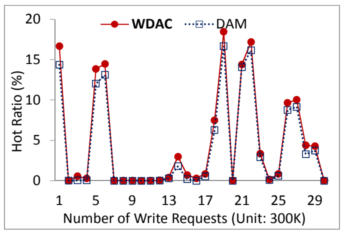
A hot ratio is a ratio of hot data to all data. First of all, we choose this *hot ratio* to compare each performance of those four hot data identification schemes and to examine closeness among them. However, even though both hot ratios of two algorithms are identical, hot data classification results of both schemes may be able to be different since an identical hot ratio means the same number of hot data to all data and does not necessarily mean all classification results are identical. Thus, in order to make up for this limitation, we employ another evaluation metric: *false identification rate*. Whenever write requests are issued, we try to compare each identification result of each scheme. This enables us to make a more precise



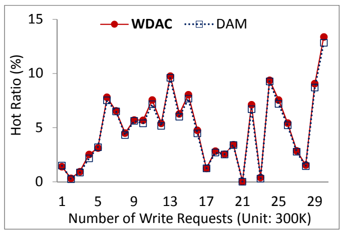
(a) Financial1



(b) MSR

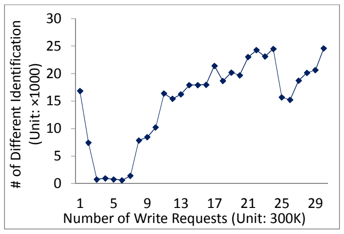


(c) Distilled

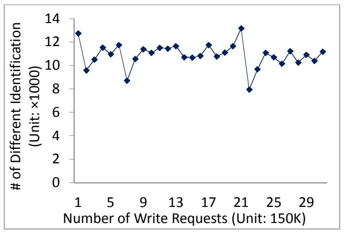


(d) RealSSD

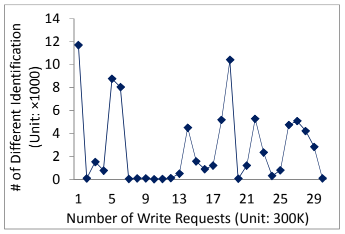
Fig. 8. Hot Ratios of Two Baseline Algorithms



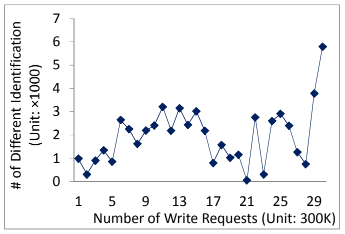
(a) Financial1



(b) MSR



(c) Distilled



(d) RealSSD

Fig. 9. The Number of Different Identification between Two Baseline Algorithms

analysis of them. *Memory consumption* is another important factor to be discussed since SRAM size is very limited in flash memory. Finally, *runtime overhead* also must be taken in account. To evaluate it, we choose two main operations and measure CPU clock cycles per operation in each scheme.

C. Results and Analysis

We discuss our evaluation results in diverse respects.

• ***Baseline Algorithm:*** We first start to evaluate two baseline schemes: our proposed WDAC and DAM. As shown in Figure 8 (a) and (b), the hot ratios of both schemes exhibit considerably different results under Financial1 and MSR, whereas they display almost identical patterns under Distilled and RealSSD traces. However, as mentioned before, the identical hot ratios do not necessarily mean the same performance of them. Thus we make another experiment for a more comparative analysis.

Figure 9 plots the number of different identification results between WDAC and DAM. We count the number of different hot classification results during a specified unit time (150K or 300K write requests) between them. Thus, zero value means all the identification results are identical between each scheme during the corresponding unit time. Particularly, Figure 9 (c) and (d) illustrate well the performance of both schemes can be different even though two hot ratios look very similar each other as shown in Figure 8 (c) and (d).

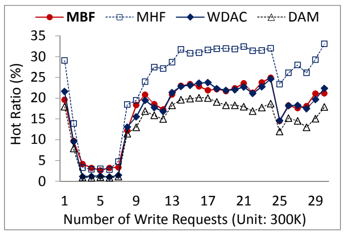
• ***Our Scheme (MBF) vs. MHF:*** Now, we make an attempt to evaluate performance of our proposed scheme (MBF) and the multiple hash function scheme (MHF). For references, we include our aforementioned two baselines. In particular, we try to compare with DAM as well as WDAC in order to demonstrate that our proposed WDAC is not an unfairly customized baseline algorithm which is best fit for our scheme.

As illustrated in Figure 10, our MBF presents a very close approach to our baseline, even to DAM, while the MHF has a tendency to exhibit considerably higher hot ratios than MBF as well as two baselines under all traces. This results from a higher ratio of false identification in MHF. That is, since the BF size (i.e., hash table size) is limited, even though write requests of cold data are increased, the chance of incorrect counter increments also grows due to hash collisions. As a result, it causes higher hot ratios. As Hsieh et al. are mentioned in their paper, the MHF scheme does not show a good performance especially when the hot data ratio in traces is very low. To get over this limitation, they suggested two solutions: a larger size of the hash table or a more frequent decay operation. However, both suggestions cannot be fundamental solutions since not only does a larger hash table require more memory consumption, but also a more frequent decay produces a significant performance overhead.

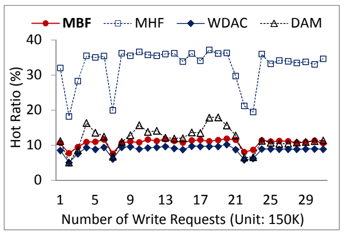
Our scheme, on the other side, resolves that limitation by adopting multiple BFs. Unlike MFH adopting the exponential batch decay after a longer decay period (here, 4,096 write requests), our scheme erases all information only in one BF out of V BFs (here, V=4) after a shorter decay period (here, 512 write requests). Furthermore, a smaller BF results in lower computational overhead as well as less memory space consumption. Both memory consumption and runtime overhead will be discussed in the next subsections in more detail.

Figure 11 shows the false identification rates of both our scheme and MHF scheme. We compare both MBF and MHF with our proposed ideal scheme (WDAC). Thus, we can call these results false identification rates. As presented in Figure 11, our MBF scheme exhibits much lower false identification rates than MHF. It improves its performance by an average of 41%, 65%, 59%, and 36% under the four workloads respectively.

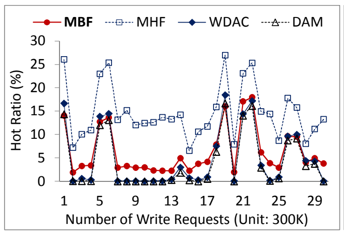
• ***Runtime Overhead:*** Computational overhead is another



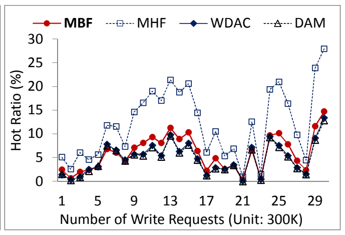
(a) Financial1



(b) MSR

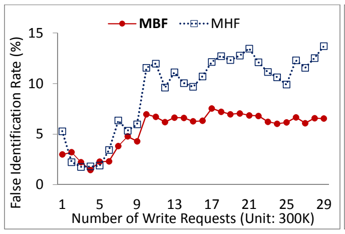


(c) Distilled

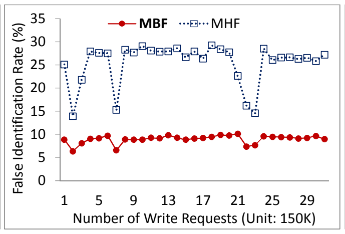


(d) RealSSD

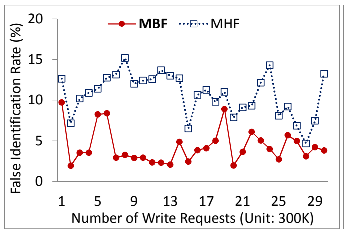
Fig. 10. Hot Ratios of Four Schemes under Various Traces



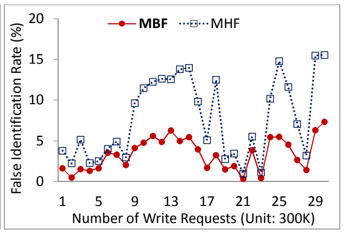
(a) Financial1



(b) MSR



(c) Distilled



(d) RealSSD

Fig. 11. False Identification Rates of Both MBF and MHF

important factor to evaluate hot data identification scheme. Figure 12 displays runtime overheads for a checkup and decay operation in MBF and MHF. A checkup operation means the verification process to check if the data in the corresponding LBA are hot whenever write requests are issued. A decay operation is the aforementioned aging mechanism. Since both are the most representative operations in hot data identification schemes, we choose and evaluate them. To evaluate each operation, we measure CPU clock cycles for them under the configurations in Table I. This measuring is done over an AMD X2 3800+ (2GHz) system with 2G RAM under Windows XP Professional platform. For fair and precise evaluation of each operation in both schemes, we feed many write requests from MSR traces (100K numbers of write requests) and then measure their average CPU clock cycles since a CPU clock cycle is dependent on cache misses (such as code miss and data miss) in CPU caches. In other words, when we measure CPU clock cycles of the operations, a lot more clock cycles are required at the beginning stage due to CPU cache misses, while the clock cycles start to be significantly reduced and stabilized soon afterwards.

As presented in Figure 12, the runtime overhead of a checkup operation in MBF is comparable to the checkup operation in MHF because both schemes adopt identical hash functions and BF data structures. However, our scheme requires about 6% less computational overhead than MHF since our scheme (MBF) sets only one bit for each hash function, while MHF frequently needs to set more than one bit. That is, since MHF scheme consists of 4-bit counters, one or more bits can need to be set in order to increase the corresponding counters. In the case that a chosen BF in MBF has already recorded the corresponding LBA information, it tries to choose another BF until it finds one available. Intuitively, this process may require extra overhead in comparison with the MHF.

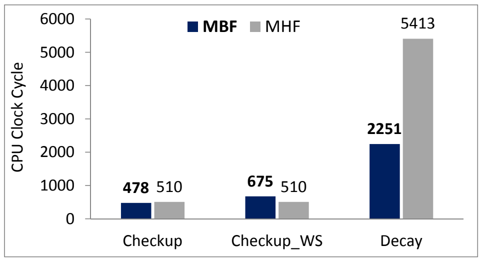
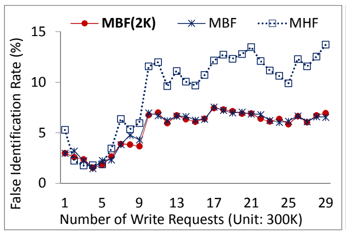


Fig. 12. Average Runtime Overhead per Operations. Here, Checkup_WS means the checkup operation *without* a shortcut decision.

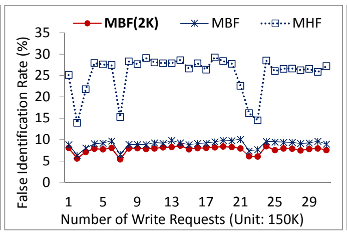
However, when it turns out that all BFs (here, 4 BFs) have already included the LBA information, MBF simply defines the data as hot and skips further processes such as a BF checking or recency weight assignment. This shortcut decision in the checkup operation considerably reduces its overhead by an average of 29% compared to the checkup operation without shortcut decision (478 *vs.* 675) in our scheme.

On the other hand, a decay operation of our scheme outperforms that of MHF. Our scheme consists of 4 numbers of 2,048 sizes of 1-bit array, while MHF is composed of 4,096 sizes of 4-bit array. Thus, whenever a decay operation is executed in MHF, all 4,096 numbers of 4-bit counters must be right-shifted by 1-bit at once. Our scheme, however, needs to reset only 2,048 sizes of 1-bit array. Therefore, although our scheme requires more frequent (here, 4 times more frequent) decay operations than MHF, it requires almost a half (58%) less runtime overhead than MHF.

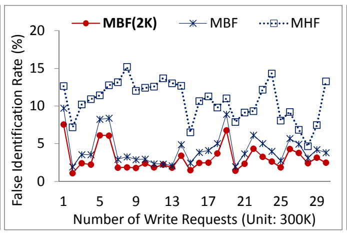
• ***Impact of Memory Size:*** Our scheme consumes only 1KB, while MHF requires a double (2KB). As mentioned before, our proposed scheme comprises 4 BFs each of which consists of 0.25KB (i.e., 2K×1-bit). On the other hand, MHF is composed of 4K numbers of 4-bit counters (i.e., 4K×4-bit). Therefore,



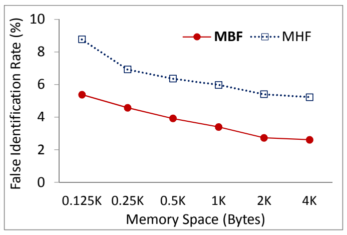
(a) Financial1



(b) MSR

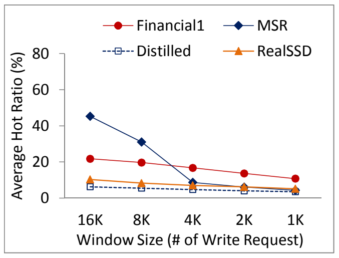


(c) Distilled

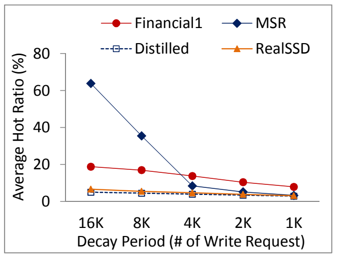


(d) RealSSD

Fig. 13. False Identification Rates between Two Schemes with Same Memory Space. Here, original MBF and MHF requires 1KB and 2KB respectively. Figure (d) shows the performance change over various memory space under RealSSD traces.

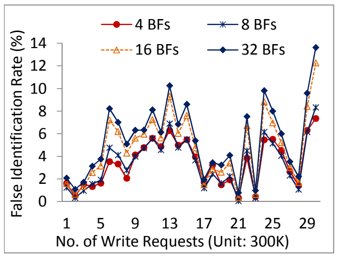


(a) WDAC

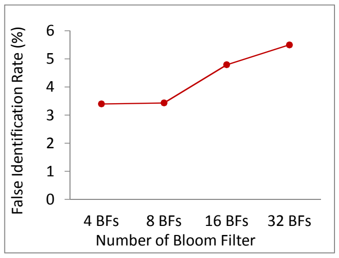


(b) DAM

Fig. 14. Changes of Average Hot Ratios over Various Window Sizes in WDAC and Decay Periods in DAM



(a) MHF



(b) Change of Total False Identification Rates

Fig. 15. Performance Change over Various Numbers of a Bloom Filter in Our Scheme under RealSSD trace

our scheme consumes only a half of memory space of MHF scheme.

Figure 13 illustrates false identification rates between two schemes with same memory space. Since MHF originally requires 2KB, we double each BF size in our scheme from 0.25KB to 0.5KB (i.e., 0.5KB×4=2KB). For reference, we include our original MBF (1KB) into each plot. As shown in Figure 13, MBF with 2KB improves its performance by lowering its false identification rates further than original MBF (1KB), which means our proposed scheme clearly outperforms MHF under the same memory space condition. MBF with 2K benefits from a larger BF size so that it can reduce the possibility of a false positive in each BF.

Different memory spaces (i.e., different bloom filter sizes) will have an effect on the performance of each scheme. We now explore the impact of memory space on both MBF and MHF. Figure 13 (d) exhibits performance changes of both schemes over various memory spaces under RealSSD trace file. Both schemes with larger memory space show a better performance than those with smaller memory space. As a memory space grows, although each performance is also improved accordingly, our proposed scheme still exhibits better performance than MHF throughout all memory spaces from 0.125KB to 4KB. Since other experimental results under the other traces such as Financial1, MSR and Distilled also show almost identical patterns with this experiment, we do not provide the other ones.

• ***Impact of Window Size:*** Intuitively, a larger window will cause a higher hot ratio because a window size is directly associated with frequency information. All experiments in Figure 14 demonstrate well this intuition. Since total hot data index values for each LBA are the summation of all weighted frequency values of the corresponding LBA within the window, the frequency value has a great impact on the hot data decision. A larger window can contain a more number of LBA access information so that it necessarily causes a higher frequency value for each LBA. This results in a higher hot ratio. Conceptually the decay period in DAM is similar to the window size in WDAC. Thus, we also explore the relationship between various decay periods and hot ratios in DAM. Figure 14 presents the changes of average hot ratios over a different window size in WDAC and a different decay period in DAM. Like WDAC, a longer decay period causes a higher hot ratio in DAM. Thus, overall trends of hot ratios in DAM exhibit very similar results.

• ***Impact of the Number of a Bloom Filter:*** In our proposed scheme, the number of BF corresponds to the granularity of recency. To explore its impact, we make experiments with various numbers of BF in our scheme. For more objective comparison, we not only assign the same memory space (1KB) and the same number of hash function (2) to each scheme, but also configure BF sizes and decay periods accordingly. As presented in Figure 15 (b), as the number of BF grows, the false identification rates also increase. This result is closely related with BF sizes of each scheme. Since each configuration has the same total memory consumption, smaller BFs have to be assigned to the scheme with a more number of BF, which results in higher false identification rates. Even though the scheme with a more number of BF can capture finer-grained recency information, this benefit is offset by its higher false identification rates. To verify this, we make another experiment with the same configurations except for memory consumption. When we double the number of BF from 4 to

8 while remaining the same BF size (i.e., 1KB to 2KB), the performance is improved by an average of 18% since it can benefit from capturing its finer-grained recency.

V. Conclusion

In this paper, we proposed a novel hot data identification scheme for flash memory-based storage systems. Unlike the multihash framework, our scheme adopts multiple bloom filters and each bloom filter has a different weight and a different recency coverage so that it can capture finer-grained recency information. Furthermore, multiple and smaller bloom filters empower our scheme to achieve not only lower runtime overheads, but also less memory consumption.

In addition to this novel scheme, we proposed a more reasonable baseline algorithm to approximate an ideal hot data identification named Window-based Direct Address Counting (WDAC). The existing algorithm (i.e., direct address method) cannot properly catch recency information because it assigns an identical recency weight to all LBAs accessed within a decay period. However, our WDAC allots all different recency weights to all LBAs within a window according to their access sequences so that it can capture precise recency as well as frequency information. Consequently, WDAC can properly identify hot data.

We made experiments in many respects under diverse real traces including real SSD traces. All hot data identification results (i.e., hot ratios) of our scheme display much closer results to those of the baseline scheme than the other one. Furthermore, to make up for the limitation of a hot ratio-based analysis, we also compared the number (or rate) of false identification by making one-to-one comparison of each identification result. This pinpoint evaluation not only enables us to make a comparative analysis of each performance, but also demonstrates that our scheme more precisely identifies hot data. Lastly, we carried out experiments on variable memory size, window size, and the number of a bloom filter in our scheme to explore their impacts. Our experiments present that our proposed scheme improves the performance up to 65% although its runtime overhead of, in particular, decay operation in our scheme requires less CPU clocks up to 58% and less memory space, by an average of 50%.

Acknowledgment

This research was partially supported by the National Science Foundation (NSF) grants 0960833 and 0934396.

References

[1] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *EuroSys '09*, 2009, pp. 145–158.

[2] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song, "A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, 2007.

[3] C.-H. Wu and T.-W. Kuo, "An adaptive two-level management for the flash translation layer in embedded systems," in *ICCAD*, 2006.

[4] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J. Kim, "A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 4, 2008.

[5] J. Shin, Z. Xia, N. Xu, R. Gao, X. Cai, S. Maeng, and F. Hsu, "FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications," in *ICS*, 2009.

[6] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *ASPLOS*, 2009.

[7] D. Park, B. Debnath, and D. Du, "CFTL: A Convertible Flash Translation Layer Adaptive to Data Access Patterns," in *SIGMETRICS*, 2010.

[8] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *USENIX*, 2008.

[9] L.-P. Chang and T.-W. Kuo, "Efficient Management for Large-scale Flash Memory Storage Systems with Resource Conservation," in *ACM Transactions on Storage*, vol. 1, no. 4, 2005.

[10] B. Debnath, S. Subramanya, D. Du, and D. J. Lilja, "Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks," in *MASCOTS*, 2009.

[11] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *FAST*, 2008.

[12] S. Nath and A. Kansal, "FlashDB: Dynamic Self-tuning Database for NAND Flash," in *IPSN*, 2007.

[13] Y. Chang, J. Hsieh, and T. Kuo, "Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design," in *DAC*, 2007.

[14] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," in *FAST*, 2010.

[15] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD Lifetimes with Disk-Based Write Caches," in *FAST*, 2010.

[16] G. Sun, Y. Joo, Y. Chen, D. Niu, Y. Xie, Y. Chen, and H. Li, "A Hybrid Solid-State Storage Architecture for Performance, Energy Consumption and Lifetime Improvement," in *HPCA*, 2010.

[17] L.-P. Chang, "Hybrid solid-state disks: combining heterogeneous NAND flash in large SSDs," in *Proceedings of the Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '08, 2008, pp. 428–433. [Online]. Available: http://portal.acm.org/citation.cfm?id=1356802.1356908

[18] M.-L. Chiang, P. C. H. Lee, and R. chuan Chang, "Managing flash memory in personal communication devices," in *Proceedings of the 1997 International Symposium on Consumer Electronics*, 1997, pp. 177–182.

[19] L.-P. Chang and T.-W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," in *RTAS*, 2002.

[20] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient Identification of Hot Data for Flash Memory Storage Systems," *ACM Transactions on Storage*, vol. 2, no. 1, 2006.

[21] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 4, pp. 837–863, 2004.

[22] A. Ban, "Wear Leveling of Static Areas in Flash memory," *US Patent,6732221, M-Systems*, 2004.

[23] L. Chang, "An Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems," in *SAC*, 2007.

[24] M. Bauer, R. Alexis, G. Atwood, B. Baltar, A. Fazio, K. Frary, M. Hensel, M. Ishac, J. Javanifard, M. Landgraf, D. Leak, K. Loe, D. Mills, P. Ruby, R. Rozman, S. Sweha, S. Talreja, and K. Wojciechowski, "A multilevel-cell 32 mb flash memory," feb. 1995, pp. 132–133, 351.

[25] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, 1970.

[26] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397 – 409, April 2006.

[27] UMass, "OLTP Trace from UMass Trace Repository," http://traces.cs.umass.edu/index.php/Storage/Storage, 2002.

[28] Microsoft, "SNIA IOTTA Repository: MSR Cambridge Block I/O Traces," http://iotta.snia.org/traces/list/BlockIO, 2007.

[29] D. Narayanan and A. Donnelly, "Write Off-Loading: Practical Power Management for Enterprise Storage," in *FAST*, 2008.