2018

BloomStream: Data Temperature Identification for Flash Based Memory Storage Using Bloom Filters

Janki Bhimani, *Northeastern University*
Ningfang Mi
Bo Sheng

BloomStream: Data Temperature Identification for Flash Based Memory Storage Using Bloom Filters

Janki Bhimani
bhimani@ece.neu.edu
Northeastern University

Ningfang Mi
ningfang@ece.neu.edu
Northeastern University

Bo Sheng
bo.sheng@umb.edu
UMASS Boston

***Abstract*—**
Data temperature identification is an importance issue of many fields like data caching and storage tiering in modern flash-based storage systems. With the technological advancement of memory and storage, data temperature identification is no longer just a classification of hot and cold, but instead becomes a "multi-streaming" data categorization problem to classify data into multiple categories according to their temperature. Therefore, we propose a novel data temperature identification scheme that adopts bloom filters to efficiently capture both frequency and recency of data blocks and accurately identify the exact data temperature for each data block. Moreover, in bloom filter data structure we replace the original OR operation with the XOR masking operation such that our scheme can delete or reset bits in bloom filters and thus avoid high false positives due to saturation. We further utilize twin bloom filters to alternatively keep unmasked clean copies of data and thus ensure low false negative rate. Our extensive evaluation results show that our new scheme can accurately identify the exact data temperature with low false identification rates across different synthetic and real I/O workloads. More importantly, our scheme consumes less memory space compared to other existing data temperature identification schemes.

***Index Terms*—Data Temperature, Bloom Filters, Stream Identification, Flash Memory, Multi-stream SSDs, Caching, Tiering**

I. Introduction

The industrial and economical growth increases the data appetite of computer systems. The fundamental efficiency of all wear-leveling and garbage collection algorithms mainly depends on the data temperature identification[1]. This is because the performance and lifetime of flash-based storage devices are significantly affected by the placement of data with different temperatures [1]–[3]. Moreover, the hybrid and all flash datacenters [4] that contain layers of different storage devices for tiering or caching are becoming popular. In order to best use these storage stacks, it is very important to identify and categorize data according to their temperature, such that the frequently used data can be placed on faster storage devices. Lastly, with the recent innovations of memory solutions and storage technologies such as 3D Xpoint [5], multi-stream SSDs [6] and key-value SSDs [7] , the key towards well using these hardware advancements is also to identify and categorize data with respect to its temperature.

This project was partially supported by the NSF Career Award CNS-1452751 and NSF Grant CNS-1552525.

[1]Data temperature is the measure of the importance of a piece of data for faster overall processing of application workload.

Thus, data temperature identification is no longer just a classification problem where the data is either classified as hot or cold. Instead, it becomes a "multi-streaming" data categorization problem where we need to classify data into multiple (more than two) categories according to their temperature. For example, modern storage devices such as multi-stream SSDs would require up to 16 different data streams categorized according to data temperatures. However, we find that many of existing methods [3], [8]–[13] that are effective for hot/cold data classification, fail to be able to identify more than two categories. Moreover, although there do exist some previous techniques [1], [3], [6], [14], [15] that are capable of identifying exact access frequency for "multi-streaming" data categorization or can be modified to support it, we notice that all these existing methods either introduce significant memory-space overheads (e.g., using data structures like tables or queues to track data access time) or require considerable computing overheads (e.g., in the emulation of the Least-Recently-Used (LRU) method).

To address the above issue, bloom filter[2] (BF) [16] has been adopted by some existing techniques [17], [18] to identify and store data access frequency. However, existing BF methods either requires many BFs in order to capture high data access frequency, or inherits one main drawback of bloom filter, i.e., due to the OR operation, the BF bits that are set to 1 cannot be reset to 0. Thus, data elements cannot be removed from the BF. Consequently, when data set size increases or more data elements are added to the BF, the false positive[3] rate of that BF increases. To reduce the false positive rate, one has to increase the size of the BF (i.e., the number of bits in the BF), which thus consumes larger memory space.

Therefore, this paper strives to develop a new efficient data temperature identification scheme, called BloomStream, which is able to (1) provide the exact weighted data access frequency under the consideration of data recency, which we combinedly refer to as data temperature, and (2) reduce the false identification rate (i.e., including both false positives and false negatives) with low memory and computational overhead. In particular, the output of our new scheme is the exact data temperature for each data element and thus can be used as an

[2]Bloom filter is a space-efficient probabilistic data structure for checking if an element is a member of a set or not. The time complexity of bloom filters to check for any particular element is constant, i.e., $O(1)$.

[3]A false positive error here indicates an existence of data, when it does not.

input for any multi-category data temperature identification. More importantly, one of our main goals is to achieve a low false identification rate, where false identification error indicates a mismatch between the identified data temperature and the actual data temperature. We notice that although traditional BF-based methods can ensure false negative rates to be zero by using the OR operation to insert data, they inevitably get high false positive rates when data set sizes become large, as discussed above. Whereas, in the problem of multi-category data temperature identification, we need to reduce both false negative and false positive rates, and more precisely, reduce any mismatch between identified and actual data temperature, i.e., false identification rate. In summary, the main contributions of this paper are as follows:

- ***XOR Masking:*** We first modify the traditional bloom filter by using the XOR masking operation to replace the original OR operation when inserting data access. By this way, our scheme can delete data (i.e., reset $1 \rightarrow 0$) to avoid its saturation and high false positive. We analyze this modification in Section III-A.
- ***Twin Bloom Filters:*** We further adopt two Bloom filters to track data access alternatively and ensure that at any moment at least one BF stores the unmasked clean copy of data. By combining two bloom filters with XOR masking, our scheme correctly indicates data access (i.e., decrease false negatives) and meanwhile avoid saturating BFs (i.e., reduce false positives). The details are presented in Section III.
- ***Recency:*** Finally, we use a history BF and a history counter list to record aggregated history information for data look up. We periodically decay the access frequency in the history counter list and maintain aggregate data temperature considering frequency and recency. This transition process is described in Section III-D.
- The source code of our data temperature identification framework can be available at Github[4], hoping towards its better usability and further improvement.

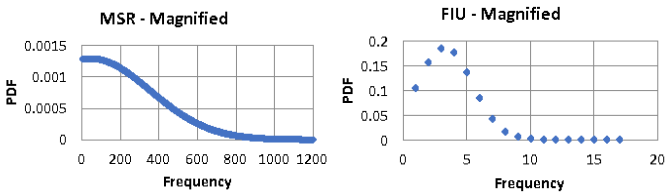
II. Motivation and Background

In this section, we present our workload analysis on data access frequency and discuss the existing data temperature identification schemes as well as the existing bloom filter related hot/cold data identification schemes.

A. Workload Analysis

As discussed in Section I, access frequency is one of the very important metrics for data categorization [2], [19], [20]. Thus, we first analyze different real I/O workload traces and study the distribution of their data access frequency. Figure 1 plots the probability density functions (PDFs) of 2 sample I/O workloads. The detailed description of these workloads will later be discussed in Section IV. From Figure 1, we can observe that the distribution of access frequency (e.g., the peak of PDFs and the range of access frequency) varies widely across different I/O workloads. For example, the MSR workload (see Figure 1 (a)) has access frequency of its

[4]https://github.com/bhimanijanki/bloomStream



(a) (b)

Figure 1: Probability density functions for access frequency of different real traces such as a) MSR, and b) FIU

majority I/Os (i.e., the peak point of its PDF) around 100 and has a long tail in the distribution with access frequency up to 1200. In contrast, the FIU workload (see Figure 1 (b)) have majority of their I/Os with access frequency around 4 and their maximum access frequency is less than 100. Given these observations, we believe that the traditional schemes [8], [17], [21], which only capture up to some limited access counts, may not be able to give a good data temperature categorization for all I/O workloads.

B. Existing Data Categorization Schemes

Now, we turn to discuss some existing schemes that are capable to separate data into multiple categories rather than just hot or cold. In [18], Hsieh et al. proposed a hot data identification algorithm, named a direct address method (DAM) as their baseline. By using a list of logical block addresses (LBA), simple counters and periodic decay, DAM captures the frequency and recency information of each LBA. We also use DAM as the baseline to compare the correctness of data temperature identification in our evaluation.

There are some machine learning based data temperature identification schemes that are also capable of identifying more than two categories of data. [1] and [14] presented the clustering schemes based on extents and features to split the address space into up to N hot and cold categories of different sizes. Although these machine learning based algorithms are very flexible in terms of number of temperature categories and number of features, they still need a large amount of computational cycles (more than 2,000 clock cycles per operation).

C. Existing Schemes Using Bloom Filters

Bloom filter (BF) [16] is an efficient data structure. It contains a bit array of m bits, initially set to 0. To insert an element to the BF, the element is first fed to k hash functions and then set the corresponding k bit positions in the BF's array to 1. To check if an element is in the set, we first get the k bit positions by feeding the key value of that element to all k hash functions. If all the k bits are equal to 1, then the element is in the BF. However, there exist two possible cases if all the k bits are 1: (1) the element is actually in the set, and (2) the element is actually not in the set. We call the latter case as a false positive that is caused due to the insertion of other elements. Given the number (n) of elements expected to have in the BF and the acceptable error rate (p),

we can use the following formulas to calculate the number (m) of bits needed in the BF and the number (k) of hash functions we should apply. Where, $m = -n * ln(p)/(ln(2)^2)$ and $k = m/n * ln(2)$. Usually, the false positive rate in the BF-based algorithms is low. However, when the number of elements stored in the BF increases, the rate of false positive also increases, i.e., indicating an existence of an element when that element is not present. Additionally, major drawback of the BF lies in its incapability of deleting data. As a result, no longer important or older information cannot be removed without deleting entire BF.

Hsieh et al. [18] presented a scheme MHF that adopts multiple hash functions and one BF with a D-bit counter for each bit position in the BF. It captures the frequency of a data block by increasing the counter values of its corresponding BF bit positions by one for each access. The recency of a hot data block is captured by dividing its corresponding counter values by 2 periodically. If the values of all $D/2$ most significant bit positions are equal to 0, then that data is considered as cold otherwise hot. To make it capable to separate data into multiple categories, we need to modify it by introducing multiple thresholds. For example, with the 4-bit counters, one can say that the data belongs to the hottest category if its access counter values are all greater than or equal to 8 (binary - 1000). If its counter values are greater than or equal to 4 (binary - 0100), but lesser than 8 (binary - 1000), then that data belongs to the next hotter category and so on. Moreover, MHF still faces all bloom filter's issues as discussed above. Later, in our evaluation, we use the modified MHF to compare with ours.

Recently, [17] proposed another BF based scheme called multiple bloom filter (MBF) to efficiently classify data into hot or cold. This scheme adopts a set of V independent BFs to capture the V times occurrences of data. In order to record higher number of occurrences of data, MBF needs to have more physical BFs, especially for the workloads with large peak access frequency such as MSR, as shown in Figure 1 (a), MBF needs to have at least 100 BFs. Given that, we note that MBF is not suitable to directly adapt to multiple categories identification.

III. Framework

In this section, we present BloomStream, a new data temperature identification scheme that aims to achieve accurate multiple stream data categorization with low overhead of computation and memory space. The goal of our scheme is to keep track of I/O requests of logic blocks, and when queried with a LBA (logic block address), return a temperature value indicating the predicted access frequency and recency of that block. The main performance metric is the accuracy, i.e., the difference between predicted and actual temperature values.

Figure 2 depicts the main structure of our proposed BloomStream. We divide a sequence of I/O requests into multiple windows each consisting of the same number of I/O requests. Our scheme records two sets of information: one for the current window (left side of the dashed line in Figure 2),

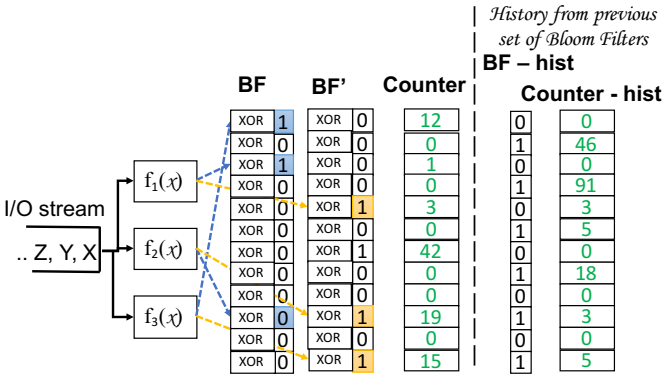


Figure 2: Our scheme BloomStream and its operation

and the other for aggregated historical requests (right side in Figure 2). Both sets consist of Bloom filter(s) and a list of counters, where each counter is associated with a bit in the Bloom filter(s). Our basic intuition is to use Bloom filter to check the existence of a LBA in the history, and if it does exist, use the counter to return a temperature value. The information for the current window consists of two Bloom filters (BF and BF') each with M bits, and a counter list ($Counter$). The aggregated information is represented by one Bloom filter and another counter list (BF_hist and $Counter\_hist$). Different from the traditional Bloom filter, our Bloom filter adopts XOR masking operations. In the rest of this section, we first explain the motivation and effects of our new XOR masking technique (Subsection III-A). Then, we describe our algorithms for the two basic operations, LBA data insert and query (Subsections III-B and III-C). Finally we present our algorithm that aggregates the historic information for access prediction considering data recency (Subsection III-D).

A. XOR Masking in Bloom Filter

In this paper, when adding a data into a Bloom filter, we use XOR masking rather than regular OR operation based on the following two motivations. First, traditional Bloom filter does not support deleting data, thus could get saturated when we keep adding new data. It results in a high false positive rate. XOR masking used in this paper can play a role of resetting the bits ($1 \rightarrow 0$) in a Bloom filter, and help mitigate the saturation problem. Second, our goal of using Bloom filter is different from regular Bloom filter usage. Traditional Bloom filter needs to guarantee 0 false negative, and the only accuracy metric is the false positive. When using XOR masking, our scheme yields both false negative and false positive. However, our goal is to return accurate temperature value upon a LBA query. Thus, a false positive is as bad as false negative. In this subsection, we first analyze the impact of XOR masking on BF saturation. Then, we define our new accuracy metric considering both false negative and false positive.

BF saturation ratio. We define *BF saturation ratio* as the ratio of the number '1's over the BF bit length. Assume that the hash functions are independent and perfectly random, we use a

BF with m bits, k hash functions, and n is the amount of data to be stored in BF. An operation of changing bit from $0 \rightarrow 1$ is refereed as *set* a bit and that from $1 \rightarrow 0$ is refereed as *reset*. In a traditional BF with OR operations, the probability that a given bit is set to 1 when adding an item with k hash functions is $1-(1-\frac{1}{m})^k$. In another word, a bit is 1 if at least one of the k hash functions select it to be set. After n data items are added, the probability that a bit is '1' is $P1_{OR} = 1 - (1 - \frac{1}{m})^{n \cdot k}$. Let $PS_{OR}(x)$ represent the probability that there are exactly x '1's in a traditional BF,

$$PS_{OR}(x) = \left(\begin{array}{c} x \\ m \end{array} \right) P1_{OR}^{x} \cdot (1 - P1_{OR})^{m-x}.$$

When we use XOR masking, among multiple arrivals of the same data, for each odd number of arrivals, XOR hash can set a bit. Therefore, the probability of setting a given bit to 1 is $P1_{XOR} = 1 - (1 - \frac{0.5}{m})^{n \cdot k}$. Similarly, the probability of having x '1's is

$$PS_{XOR}(x) = \left(\begin{array}{c} x \\ m \end{array} \right) P1_{XOR}^{x} \cdot (1 - P1_{XOR})^{m-x}.$$

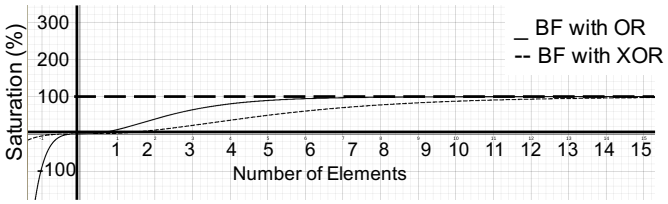


Figure 3: Comparison of the probability that a specific set of k bits are 1 in a BF of size m for traditional BF with OR and a BF with XOR masking (m = 5, k = 3)

Figure 3 shows a comparison of the saturation probability of a specific set of k bits, i.e., $P1_{OR}^{k}$ and $P1_{XOR}^{k}$, where $k = 3$ and $m = 5$. We see that traditional BF gets saturated with maximum 6 elements, while our BF with XOR masking allows more than twice the number of elements (14 elements) before getting saturated.

False identification. We use *false identification* to indicate the errors of BF caused by both false positive and false negative. If an existing element is not found, or a non-existing element is found, the false identification value is increased by one. Figure 4 illustrates an example of inserting 3 data requests and querying a set of 10 data elements (a–j). It shows a comparison of false identification and saturation of a tradition BF with OR and a BF with XOR masking. We see that traditional BF using OR gate cannot reset $1 \rightarrow 0$, thus once the BF bits are saturated then the false identification drastically increases. While the BF with XOR masking results into less number of total false identification (7 in Figure 4) than that of traditional BF (16 in Figure 4). Depending upon I/O stream the number of false identification may vary but from our experiments we notice that overall number of false identifications remains smaller compared to traditional BF.

I/O stream of data chunks - a, b, b`, # hash function – 3, # BF bits – 5, BF hash map - a: 1, 2, 3 **b:** 3, 4, 5

	BF with OR I/O streams			BF with XOR I/O streams			Other Possible Data
	a	b	b`	a	b	b`	c: 1, 3, 5
1	0	1	1	0	1	1	d: 1, 3, 4
2	0	1	1	0	1	1	e: 1, 2, 4
3	0	1	1	0	1	0	f: 1, 4, 5
4	0	0	1	0	0	1	g: 1, 2, 5
5	0	0	1	0	0	1	h: 2, 3, 4
False Positive:	**0**	**0**	**8**	**8**	**0**	**0**	**4**	**0**	i: 2, 3, 5
False Negative:	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**1**	j: 2, 4, 5
Total False Identification:	0+0+8+8=***16***				0+0+6+1=***7***			
BF Saturation Ratio:	***0, 0.6, 1, 1***				***0, 0.6, 0.8, 0.6***			

Figure 4: Comparison of false identification and saturation of a tradition BF with OR and a BF with XOR masking

B. Data Insert

In this subsection, we describe our algorithm for inserting the LBA data (see alg. 1). Overall, BloomStream uses two M-bit BFs, a list of M counters, and K hash functions to maintain access frequency of data in the current window. Once an I/O request is issued to the Flash Translation Layer (FTL), the corresponding LBA is hashed by the K hash functions. The output value of each hash function corresponds to an index ($1 \sim M$) in the Bloom filter. Instead of directly setting the corresponding bit in the bloom filter as 1, BloomStream uses an XOR operation to mask the current bit value with 1 and stores the new value into the Bloom filter.

BloomStream adopts two BFs and alternatively inserts the access information into one of these two BFs (see alg. 1). The main purpose is to help reduce the false negative caused by XOR masking. One of the Bloom filters (say BF) is initialized as the current BF. BloomStream always first checks the current BF to see if a particular LBA already exists. Later, we explain how to look up for LBAs in our modified Bloom filter in alg. 2. If that LBA does not exist in the current BF, then BloomStream adds the LBA in it by masking the values in the corresponding bits with 1 (lines 4–6). If the current BF already includes LBA, then BloomStream stores that LBA in the other alternative Bloom filter BF' (lines 7–9), and switch the role of current Bloom filter to BF' (line 10). In this way, BloomStream can ensure that the data, after insertion, always remains in one of the BFs and thus improve the identification accuracy. Additionally, the corresponding counter values in the M-bit counter list are incremented by 1 for tracking data access frequency (lines 11–12).

Algorithm 1 INSERT_BF()

```
1: procedure INSERT(X_j)
2:     Get one entry from submission queue - X_j
3:     Hash the entry with hash functions f ← f_1, f_2, f_3, ... , f_k
4:     if NOT in BF then
5:         /* Insert in BF */
6:         BF [f(X_j)] = BF [f(X_j)] XOR 1
7:     else
8:         /* Insert or Mask over in BF' */
9:         BF' [f(X_j)] = BF' [f(X_j)] XOR 1
10:        Shift the current pointer
11:        /* Increment corresponding Counter bits */
12:        C[f(X_j)] = C[f(X_j)] + 1
13:    end if
14:    return
15: end procedure
```

C. Data Query/Lookup

The other importation operation in our framework is data query/lookup, i.e., given an LBA data return its temperature value. alg. 2 describes the lookup procedure in our framework. First, we apply the K hash functions on the input LBA X_j resulting into K indexes. Then, we record the index whose corresponding counter value is the smallest. Next, we check the index bit in the Bloom filter that we are querying (BF, BF', or BF_hist). If the BFs have '1' on that bit, the algorithm considers the LBA exists, and returns the aggregated counter value ($Counter$) as its temperature value (lines 5–8). Otherwise, we consider the LBA has not been accessed, and return 0 as its temperature (lines 9–13). All the lookup to query the temperature of a data are performed on BF_hist and Counter_hist. As mentioned earlier in Section III-B, BF Lookup while inserting the data (see line 5 in alg. 1) in BF and BF' is also done with the same process of alg. 2.

Algorithm 2 LOOKUP_BF()

```
1: procedure LOOKUP_BF(X_j)
2:     Hash the entry with hash functions f ← f_1, f_2, f_3, ... , f_k
3:     /* Get index of smallest counter bit */
4:     index ← IndexOf(min(C[f_1(X_j)], C[f_2(X_j)],..., C[f_k(X_j)])
5:     if all BF[index == 1 then
6:         /* the I/O chunk exist */
7:         membership ← true
8:         temperature ← Counter[index]
9:     else
10:        /* the I/O chunk does not exist */
11:        membership ← false
12:        temperature ← 0
13:    end if
14:    return membership, temperature
15: end procedure
```

The main function of this module is to identify the existence of a given LBA in Bloom filter. In order to get aggregated data temperature, we first transition twin BFs to BF-hist that we explain in the next sub-section. The BF-hist and Counter_hist maintains the aggregated temperature values considering frequency and recency.

D. Transition to History

Periodically the current state of both the Bloom filters and the counter is preserved as the "history bloom filter" and the "history counter", that represent the aggregated historical access information. In our solution, the previous value of counter are decayed by half when aggregated with the new value. alg. 3 describes the process of this transition. For each bit in BF_hist, we first check if the counter value is less than 1 after the decay. If so, it indicates that the data has not been accessed recently, thus this bit will be reset to '0' (lines 5–7). Then, we examine if the bit is a minimum frequency bit of a data that has been added. If so, the bit value is constructed by an OR operation of BF, BF' and the previous history bloom filter (line 8–10). All other bits in BF_hist will be reset to '0' (lines 11–12). In addition, the value of the current counter is added to the decayed value of the history counter (see line 13). Then finally, after the successful transition, Bloom filters BF, BF' and Counter are reset (see line 15) and returned.

Algorithm 3 UPDATE_BF_hist()

```
1: procedure UPDATE_BF_HIST(BF, BF', Counter, BF_hist, Counter_hist)
2:     /* After every fixed decay period */
3:     if num_of_I/Os ≥ decay period then
4:     for i-th bit in BF_hist do
5:         if Counter_hist[i] / 2 < 1 then
6:             BF_hist[i] ← 0
7:         end if
8:         if i is the minimum frequency bit of a data then
9:             BF_hist[i] = BF_hist[i]||BF[i]||BF'[i]
10:        else
11:            BF_hist ← 0
12:        end if
13:        Counter_hist = Counter_hist / 2 + Counter
14:    end for
15:    BF = BF' = Counter = NULL
16:    return BF, BF', Counter, BF_hist, Counter_hist
17: end procedure
```

E. A Complete Example

In order to clearly explain overall functioning of our scheme we illustrate an example in Figure 5. In example, we consider 3 hash functions. Suppose the output value of these 3 hash functions for data chunk a, b, and c are given by hash map mentioned in line 3 of Figure 5. Initially BF, BF' and Counter are NULL. As described in subsection III-B, at the beginning current pointer is at BF. Upon arrival of a and b they are inserted into BF. The corresponding counter bits are incremented. Then after, when a arrives for second time (shown by a' in I/O stream), because BF already has a copy of a, so following alg. 1, a' is inserted into BF' and a's counter bits are incremented. The current pointer now points to BF'. Following alg. 1, the remaining inserts are performed as seen from the states of BF, BF' and $Counter$ in Figure 5. Suppose, after every 4 I/Os the periodic decay and transition to history is activated. Then BF_hist and $Counter\_hist$ are determined following alg. 3. The state of BF_hist and

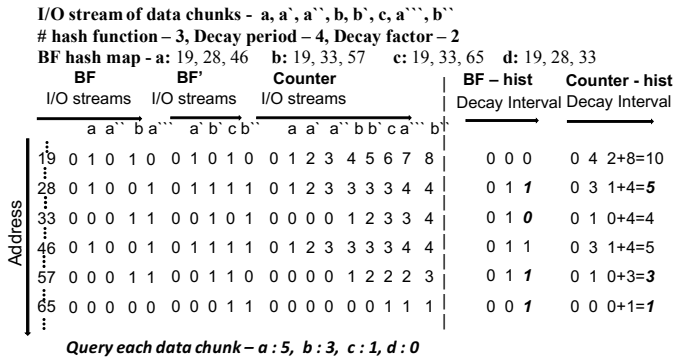


Figure 5: Illustration showing the working of our scheme with an example

$Counter\_hist$ can be seen in Figure 5 on the right side of the dashed line. Finally, if we want to lookup the membership and temperatures of a, b, c and d, then we follow alg. 2. We look the bloom filter bit at the index corresponding to the smallest counter value, among the output values of the 3 hash functions. For data chunk a, it is the bit corresponding to address 28. For data chunk b, it is the bit corresponding to address 57 and for data chunk c, it is the bit corresponding to address 65. Thus, as seen from Figure 5, finally our scheme response that the data temperature considering frequency and recency of access for data chuck a was 5, b was 3, c once and d is not seen yet.

IV. Evaluation

In this section, we present our experimental results and comparative analyses for evaluating the effectiveness of BloomStream, our new data temperature identification scheme.

A. Experimental Setup

• ***I/O Workloads:*** We evaluate the proposed BloomStream by using both synthetic workloads of different sizes and 100+ real enterprise workloads for obtaining practical in-sites. Table I shows the main characteristics of four real workloads that we chose as a representative. Specifically, `SSD` is an I/O block trace of one 450GB Samsung SSD that is collected over 1 week on a desktop computer (Xeon E5-2690, 2.9GHz, Ubuntu 16.04) in our lab. `SSD` represents the workload for general purpose programming, web surfing, data transfer from other servers, etc. `MSR` is a write intensive workload that contains 1-week block I/O traces of enterprise servers at Microsoft Research Cambridge Lab [22]. `FIU` is a read intensive trace collected from Florida International University research group [23] over 1 month. `UMASS` is a trace from a popular search engine. It is from the University of Massachusetts at Amherst Storage Repository [24]. `UMASS` has a high reuse factor as given by hit(%) in Table I. We also show the total number of requests (including both read and write I/Os) and the ratio between reads and writes of each trace in Table I. We can see that among all these four real traces, `MSR` is the most write intensive workload with the largest number of I/O requests.

Table I: Workload Characteristics

Workloads	# Total Requests	R:W (%)	Mean I/O Size (KB)	Hit (%)
SSD	2,402,296	51:49	16	70.89
MSR	30,205,489	27:73	29	91.34
FIU	10,653,454	75:25	9	72.04
UMASS	1,056,055	64:36	8	99.07

Additionally, any request for a real trace can further be divided into multiple sub-requests based on its I/O size and chunk size. For example, let us consider a write request "WRITE 200, 64K", i.e., writing data into 64K consecutive LBAs from the LBA 200. If we set chunk size to 4K, i.e., one I/O write is composed of a whole 4K block, then that write request is considered as (or divided into) 16 (i.e., 64K/4K) write sub-requests in our experiments.

• ***Baseline Algorithms:*** We consider two hot data identification schemes, i.e., the Multiple Hash Function scheme (hereafter, refer to as MHF) [18] and the Direct Address Method (hereafter, refer to as DAM) [17], as the main baseline for comparison. We modify these two schemes to report multiple data temperature categorization rather than just classifying hot or cold data and use multiple thresholds in these schemes for different data temperatures. In addition, we adopt an exponential batch decay approach [18] as a solution for a counter overflow problem in MHF that as discussed in Section II-C uses a D-bit counter for each bit position in the BF. By this way, we can fairly maintain similar delay thresholds in both MHF and our BloomStream. We also compare our BloomStream scheme with two recently proposed data temperature identification algorithms, i.e., MQ (Multi-Queue) [6] and SFR (Sequentiality, Frequency, and Recency) [6], for evaluating the memory and computing overhead.

In our experiments, we calculate the size of BF (i.e., the number of bits in BF) and the number of hash functions using equations for m and k explained in Section II. We also run MHF with the same memory space and the same number of hash functions for fair comparisons. In addition, the chunk size is set as 4KB and the decay period is 4,000 I/Os in our scheme if not explicitly mentioned.

• ***Performance Metrics:*** We use "Identification Difference" as one of the main metrics to indicate the gap (or difference) between the identified temperature and the actual temperature. For example, given a particular LBA's actual temperature is 40, we say the identification difference of this LBA is 6 if its identified temperature is either 34 or 46. Then, identification difference of 0 means that the identified temperature exactly matches the actual one. We present the distribution of identification difference by showing the proportion of data blocks (or LBAs) across various identification differences. Using this distribution result, we can report the "False Identification Ratio" (FIR) by summing the proportions of all LBAs that

Table II: Evaluating BloomStream using synthetic workload with different number of I/O requests (Req. - Requests, Amt. - Amount, Mem. - Memory, FIR - False Identification Rate)

# I/O Req.	Mean I/O Size (KB)	Amt. of Data (MB)	Mem. (KB)	# Hash Func.	FIR
2,022	64	129	1.0	4	0.041
4,007	64	256	2.0	5	0.057
6,058	64	387	3.0	6	0.070
8,089	64	517	4.0	7	0.080
10,034	64	642	5.0	8	0.083
11,948	64	764	5.9	9	0.087
13,973	64	894	6.9	10	0.086
15,971	64	1022	7.9	12	0.088
17,968	64	1149	8.9	13	0.090
19,997	64	1279	9.9	16	0.089

have their identification difference more than 0.

Another main metric we use in our evaluation is "Tolerance", which is the measure of an allowable difference in temperature identification. For example, if the tolerance is 1, then we can say a temperature identification is correct if its identification difference is smaller than or equal to 1. We further measure the "Error Rate" with respect to different tolerances to help us understand the proportion of LBAs that have their identification difference violating a given tolerance. We note that such an error rate is a more relax metric compared to the false identification ratio. Besides, we also consider memory consumption as an important factor because SRAM size is very limited in flash memory.

B. Results and Analysis

We now discuss our experimental results in terms of the correctness, efficiency and sensitivity for evaluating our scheme.

1) Synthetic Workloads

We first conduct our evaluation under a set of synthetic workloads. Table II shows the configuration of these synthetic workloads that have different number of I/O requests. Each row in the table represents one synthetic workload setup. We fix the mean I/O size, but increase the total number of I/O requests such that the total amount of data increases as well. Besides, we show the required memory space and the number of hash functions in the table. The last column in Table II gives the False Identification Rate (FIR) of our scheme by comparing our identified results with the actual ones that are obtained from trace using simple baseline DAM scheme that considers frequency and recency. We can see that our scheme in overall consumes a small amount of memory space even when we have a large workload. More importantly, the false identification rate of our scheme remains very low for all synthetic workloads, which indicates the majority ($\geq$90%) of data blocks have their identified temperature exactly matching their actual ones.

Then, we look closely to the distribution of identification difference for in-depth analysis of temperature differences. Figure 6 plots the proportion of data blocks (or LBAs)

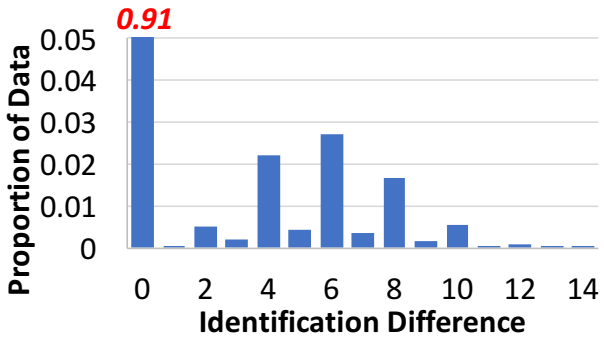


Figure 6: Frequency error Sufficient Memory

across various temperature identification differences for the last synthetic workload (i.e., with 19,997 I/O requests) shown in Table II. We can see that the temperature of 91% of data is correctly identified, which means that only 9% of data is false identified, i.e., with at least 1 identification difference. Furthermore, the identification difference is only up to 14 and most of the false identified data have their identification difference of 6.

2) Real Traces

Now, we use real I/O traces to further validate the efficiency and accuracy of our BloomStream scheme. The workload characteristics of the four I/O traces have been shown in Table I.

• ***Error Rate:*** We evaluate the performance of MHF and our scheme by plotting the error rates across different tolerances in Figure 7. In overall, BloomStream achieves lower error rates than MHF under all four real workload traces. More importantly, BloomStream significantly reduces the error rate especially when the tolerance is small. In detail, as illustrated in Figure 7(a)-(c), BloomStream obtains the error rate less than 20% for zero tolerance under the first three workloads. This signifies that our scheme could identify around 80% of data correctly even with such a strict allowance. While, the UMASS workload has a very high reuse rate (i.e., 99.07%) such that almost all LBAs are re-accessed. For such a workload with high reuse, BloomStream is able to exactly identify temperatures of 60% of LBAs without any difference. Upon increasing tolerance, the error rate decreases especially under the MHF scheme. We also notice that because MHF uses only one BF, it suffers relatively higher error rate for large real workloads, such as MSR in Figure 7(b). In contrast, our BloomStream keeps maintaining a stable, low error rate under such large workloads.

• ***Memory Size:*** We next analyze the impact of memory size on temperature identification. Figure 8 depicts the false identification ratio as a function of memory size (including the space for BFs and counters) under both MHF and BloomStream for the SSD workload. More memory space allows both schemes to obtain less false identification ratio. However, the reduction under MHF is not significant until the memory size reaches

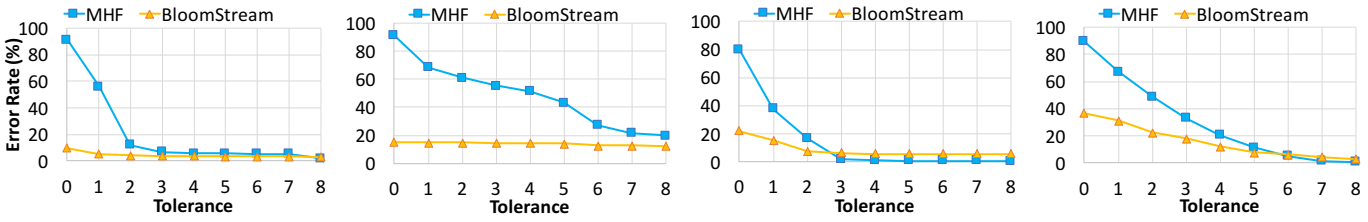


Figure 7: Error rate w.r.t. tolerance for two schemes using workloads: a) SSD, b) MSR, c) FIU, d) UMASS

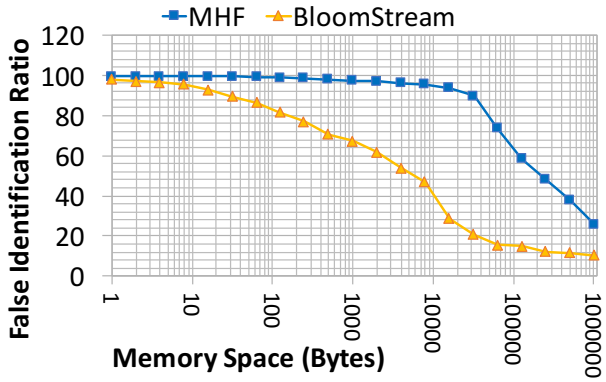


Figure 8: Our scheme BloomStream and its operation

30KB. Moreover, our proposed scheme clearly outperforms MHF under the same memory space. To get a reasonable false identification ratio (say around 20%), BloomStream just needs memory space of 45KB while MHF requires more than 1MB memory space. The other previous works of MQ and SFR does not uses BFs but instead uses queue and table as their basic data structures. These data structures consumes considerably higher memory when compared to bloom filter. The memory space required for the above discussed SSD workload with SFR scheme is 8MB and with MQ scheme is 30MB, which is much higher than that of BloomStream.

V. Conclusion

In this paper, we proposed a novel data temperature identification scheme using bloom filters for flash memory-based storage systems. Our scheme overcame the short-comings of traditional bloom filter by using two bloom filters along with the XOR masking operation for inserting new data. As a result, our scheme is able to delete (i.e., reset to 0) data in the BFs for avoiding the saturation and meanwhile obtain the unmasked clean copy of data from one of the two BFs. Our scheme also captures recency and allows simultaneous insert and look-up operations by using an additional history BF and counter. We extensively evaluated the efficiency and accuracy of our framework under both synthetic and real I/O workloads. We showed that our scheme outperforms the state-of-the-art schemes with lower false identification rates, and lower memory overhead.

References

[1] M. Shafaei, P. Desnoyers, and J. Fitzpatrick, "Write amplification reduction in flash-based ssds through extent-based temperature identification." in *HotStorage*, 2016.

[2] K. Kim, S. Jung, and Y. H. Song, "Compression ratio based hot/cold data identification for flash memory," in *ICCE*. IEEE, 2011, pp. 33–34.

[3] D. Park, B. Debnath, Y. Nam, D. H. Du, Y. Kim, and Y. Kim, "Hotdatatrap: a sampling-based hot data identification scheme for flash memory," in *SIGAPP*. ACM, 2012, pp. 1610–1617.

[4] D. G. Andersen and S. Swanson, "Rethinking flash in the data center," *IEEE micro*, vol. 30, no. 4, pp. 52–54, 2010.

[5] T. Morgan, "Intel shows off 3d xpoint memory performance," 2015.

[6] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "Autostream: automatic stream management for multi-streamed ssds," in *Proceedings of the 10th ACM SYSTOR*. ACM, 2017, p. 3.

[7] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A Flexible, High-Performance Key-Value SSD," in *HPCA*. IEEE, 2017.

[8] H.-S. Lee, H.-S. Yun, and D.-H. Lee, "Hftl: hybrid flash translation layer based on hot data identification for flash memory," *TCE*, vol. 55, 2009.

[9] M.-L. Chiang, P. C. Lee, and R.-C. Chang, "Managing flash memory in personal communication devices," in *ISCE*. IEEE, 1997, pp. 177–182.

[10] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *RTAS*. IEEE, 2002.

[11] 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*. IEEE, 2009, pp. 1–9.

[12] H. Kim and S. Ahn, "Bplru: A buffer management scheme for improving random writes in flash storage." in *FAST*, vol. 8, 2008, pp. 1–14.

[13] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du, "Bloomflash: Bloom filter on flash-based storage," in *IEEE ICDCS*, 2011.

[14] J. Bhimani, J. Yang, Z. Yang, N. Mi, N. K. Giri, R. Pandurangan, C. Choi, and V. Balakrishnan, "Enhancing ssds with multi-stream: What? why? how?" in *36th IEEE International Performance Computing and Communications Conference (IPCCC), Poster Paper. IEEE*, 2017.

[15] I. Choi and D. Shin, "Wear leveling for pcm using hot data identification," in *Proceedings of the International Conference on IT Convergence and Security 2011*. Springer, 2012, pp. 145–149.

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[17] D. Park and D. H. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *MSST, 2011 IEEE 27th Symposium on*. IEEE, 2011, pp. 1–11.

[18] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 1, pp. 22–40, 2006.

[19] L.-P. Chang and T.-W. Kuo, "Efficient management for large-scale flash-memory storage systems with resource conservation," *ACM Transactions on Storage (TOS)*, vol. 1, no. 4, pp. 381–418, 2005.

[20] T. Chilimbi and M. Hirzel, "Dynamic prefetching of hot data streams," Jun. 6 2006, uS Patent 7,058,936.

[21] C.-I. Lin, D. Park, W. He, and D. H. Du, "H-SWD:Incorporating hot data identification into shingled write disks," in *MASCOTS'12*. IEEE.

[22] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," *ACM TOS*, vol. 4, no. 3, pp. 10:1–10:23, 2008.

[23] *SNIA Iotta Repository*. [Online]. Available: http://iotta.snia.org

[24] *UMass Trace Repository*. [Online]. Available: http://traces.cs.umass.edu