ABSTRACT

Efficient hot and cold data identification in computer systems has been a fundamental issue. However, it has been least investigated. In this paper, we propose a novel online hot data identification scheme for flash-based storage named HotDataTrap. The main idea is to maintain a working set of potential hot data items in a cache based on a sampling mechanism. This sampling-based scheme enables HotDataTrap to early discard some of the cold items so that it can reduce runtime overheads as well as a waste of memory spaces. Moreover, our two-level hash indexing scheme helps HotDataTrap directly look up a requested item in the cache and save a memory space further by exploiting spatial localities. Both our sampling approach and hierarchical hash indexing scheme empower HotDataTrap to precisely and efficiently identify hot data with even less memory. Our extensive experiments with various realistic workloads demonstrate that our HotDataTrap outperforms the state-of-the-art scheme by an average of 335% and our two-level hash indexing scheme considerably improves further HotDataTrap performance up to 50.8%.

Categories and Subject Descriptors
D.4.7 [Operating Systems]: Organization and Design—Real-time systems and embedded systems

General Terms
Design, Performance

Keywords
Hot Data Identification, HotDataTrap, Flash Memory, SSD

1 Introduction

Over the years, researchers have explored various alternative storage technologies to possibly replace conventional magnetic disk drives. With the recent technological break-throughs, a NAND flash-based storage has emerged to overcome the shortcomings of them. These days, the flash-based storages like a Solid State Drive (SSD) have been increasingly adopted as main storage media especially in mobile devices such as laptops, and even in data centers [1, 2, 3, 4, 5, 6]. Although they provide a block I/O interface like magnetic disk drives, due to the lack of their mechanical head movement, not only is random read performance comparable to the sequential read performance, but also shock resistance, light weight, and low power consumption can be achieved. However, flash memory has two main limitations in terms of a write operation. First, once data are written in a flash page, an erase operation is required to update them, where the erase is almost 10 times slower than write and 100 times slower than read [7]. Thus, frequent erases severely degrade the overall performance of flash-based storage. Second, each flash block cell can be erased only for a limited number of times (i.e., 10K-100K) [7].

To get over these limitations, a flash-based storage uses an intermediate software/hardware layer named Flash Translation Layer (FTL) which hides these limitations so that existing disk-based applications can use flash memory without any modifications. However, internally FTL needs to deal with these physical limitations. To avoid an expensive erase operation for any logical page update operation, FTL treats flash memory as a log device, stores updated data to a new physical page, marks the old physical page as invalid for future garbage collection, and maintains logical-to-physical page address mapping information in order to track the latest location of a logical page [8]. Periodically, FTL invokes garbage collection to reclaim the free space resulted from the earlier update operations. In addition, during garbage collection, FTL employs wear leveling algorithms to ensure that all blocks are erased out evenly in order to improve overall longevity of the flash memory.

One of the critical issues in designing flash-based storage systems and integrating them into the storage hierarchy is how to effectively identify hot data that will be frequently accessed in near future as well as have been frequently accessed so far. Since future access information is not given as a priori, we need an on-line algorithm which can predict the future hot data based on past behaviors of a workload. Thus, if any data have been recently accessed more than

Copyright is held by the authors. This work is based on an earlier work: SAC’12 Proceedings of the 2012 ACM Symposium on Applied Computing, Copyright 2012 ACM 978-1-4503-0857-1/12/03.
http://doi.acm.org/10.1145/2245276.2232034
the threshold number of times, we can consider the data hot data, otherwise, cold data. This definition of hot, however, is still ambiguous and takes only frequency (i.e., how many times the data have been accessed) into account. However, there is another important factor—recency (i.e., when the data have been accessed)—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 identification. This hot definition can be different for each application. Now, we briefly describe some applications of a hot data identification scheme in the context of the flash-based storage systems.

**Garbage Collection and Wear Leveling:** Both have a critical impact on the access latency and lifetime of flash memory. By collecting hot data to the same block, we can considerably improve the garbage collection efficiency. In addition, we can improve its reliability by allocating hot data to the flash blocks with a low erase count [10, 11, 9, 12, 2].

**Flash Memory as a Cache:** There are some attempts [13, 14] to use flash memory as an extended cache between DRAM and HDD. In this hierarchy, they tried to keep track of page histories to decide the placement and migration of data pages between DRAM and flash memory by adopting the hot data identification concept.

**Hybrid SSD:** Hybrid SSDs such as SLC-MLC hybrid [15] and Flash-PCM hybrid [16] are another application. We can store hot data into SLC (Single-Level Cell) or PCM (Phase Change Memory), while we can store cold data into MLC (Multi-Level Cell) or NAND flash. They, consequently, can improve both performance and reliability of SSDs.

**Address Mapping Scheme:** Convertible-FTL (CFTL) is a novel hybrid address mapping algorithm designed for flash-based storage [17]. CFTL classifies all flash blocks into the hot and cold blocks according to data access patterns. It improves the address translation efficiency in a way that hot data are managed by a page-level mapping, while cold data are maintained by a block-level mapping.

**Buffer Replacement Algorithm:** LRU-WSR algorithm is specially designed for flash memory considering its asymmetric read and write latencies [18]. For a victim selection, if any page selected by LRU algorithm is defined as clean and cold, it is chosen as a victim. On the other hand, if it is defined as dirty and hot, it is moved to MRU position instead.

**Sensor Networks:** FlashDB is a flash-based B-Tree index structure optimized for sensor networks [5]. In FlashDB, the B-Tree node can be stored either in a disk mode or in a log mode. These mode selections are based on the hot data identification algorithm.

**Shingled Magnetic Recording Disks:** A Shingled Magnetic Recording (SMR) HDD is a magnetic hard disk drive that adopts the shingled magnetic recording technology to overcome the areal density limit faced in conventional hard disk drives [19]. Although this is not flash-based storage, its physical characteristics are very similar to the NAND flash [20, 21, 22, 23]. Recently, Lin et al. proposed H-SWD adopting hot data identification mechanism to improve the SMR HDD performance [24, 25].

In addition to these applications, hot data identification has a big potential to be utilized by many other fields. Although it has a great impact on designing and integrating flash memory, it has been least investigated [26]. The challenges are not only the classification accuracy, but also a limited resource (i.e., SRAM) and computational overheads. Existing schemes for flash memory either suffer from high memory consumption [27] or incur excessive computational overheads [28]. To overcome these shortcomings, Hsieh et al. recently proposed a multiple hash function framework [29]. This adopts a counting bloom filter and multiple hash functions so that it can reduce memory consumption as well as computational overheads. However, it considers simply a frequency factor and cannot identify hot data with accuracy due to a false positive probability that is an inborn limitation of a bloom filter. Moreover, it is heavily dependent on the workload characteristic information (i.e., an average hot data ratio of a workload) which cannot be given as a priori in an on-line algorithm.

Considering these observations, an efficient on-line hot data identification scheme has to meet the following requirements: (1) capturing recency as well as frequency information, (2) low memory consumption, (3) low computational overheads, and (4) independence of prerequisite information. Based on these requirements, we propose a novel on-line hot data identification scheme named HotDataTrap. The main idea is to cache and maintain a working set of potential hot data by sampling LBA access requests.

HotDataTrap captures recency as well as frequency by maintaining recency bit like a CLOCK cache algorithm [30], which results in a more accurate hot data identification. It periodically resets the recency information and decays the frequency information, which is an aging mechanism. Thus, if an item has not been accessed for a long time, eventually it is considered as cold data and will become a victim candidate for eviction from the cache. Moreover, HotDataTrap can achieve the direct cache lookup as well as less memory consumption with the help of its hierarchical two-way hash indexing scheme. Consequently, our HotDataTrap provides a much more accurate hot data identification than the state-of-the-art scheme [29] even with a very limited memory space. The main contributions of this paper are as follows:

- **Sampling-based Hot Data Identification:** Unlike all other schemes, HotDataTrap does not initially insert all requested items into the cache. Instead, it decides whether to cache them or not with our simple sampling algorithm. Once an item (i.e., requested LBA) is selected for caching, HotDataTrap maintains it as long as it is stored in the cache. Otherwise, HotDataTrap simply ignores it from the beginning. This sampling-based scheme enables HotDataTrap to early discard some of the cold items so that it can reduce the computational overhead and memory consumption.

- **Two-Level Hash Indexing:** To reduce memory consumption further, HotDataTrap does not maintain whole meta data information for each accessed LBA in the cache. Instead, it stores partial LBA information by using the hierarchical hash indexing scheme. That is, for sequential accesses, HotDataTrap stores starting accessed LBA information and only its offset information so that it can take advantage of spatial localities. Moreover, this hierarchical hash indexing enables HotDataTrap to directly access the
corresponding meta data information in the cache. This can significantly reduce the cache lookup overhead.

The rest of the paper is organized as follows. Section 2 gives a brief overview of flash memory and existing hot data identification schemes and Section 3 describes our proposed HotDataTrap scheme. Section 4 provides diverse our experiments and analyses. Finally, Section 5 concludes this research.

2 Related Work

This section describes NAND flash memory characteristics and existing hot data identification schemes for flash memory. In addition, we discuss their advantages and disadvantages.

2.1 Characteristics of Flash Memory

Flash memory is organized as an array of blocks each of which consists of either 32 pages (small block) or 64 pages (large block). It provides three types of unit operations such as read, write, and erase. These operations are asymmetric: the read and write operations are performed on a page basis, while the erase operates on a block basis. Data erase (1,500μs) is the most expensive operation in flash memory compared to data read (25μs) and write (200μs) [7].

Figure 1 shows a typical architecture of flash memory-based storage systems. Both Memory Technology Device (MTD) and Flash Translation Layer (FTL) are two major parts of flash memory architecture. The MTD provides aforementioned three primitive flash operations. 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 [31]. 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. Its motivation is to prevent any cold data from staying at any block for a long 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.

Another crucial issue in flash memory is wear leveling. Wear leveling is to evenly distribute the number of erase count for each block since, unlike DRAM, each block cell in flash memory has a limited life span. That is, after a limited number of erases to each block, errors can start to occur. Currently, the most common allowable number of write per block is typically 10K for MLC (Multi-level Cell) and 100K for SLC (Single-level Cell). Wear leveling is largely categorized into dynamic wear leveling and static wear leveling. Dynamic wear leveling tries to balance the block erase count by allocating hot data to the blocks with low erase counts. However, this will not be able to guarantee a complete balance of block erase counts since some blocks with cold data may not be erased forever. On the contrary, static wear leveling can evenly distribute the erase counts by migrating cold data into blocks with high erase counts. Dynamic wear leveling has a shorter life expectancy than static wear leveling since a complete balance of block erase counts is hard to achieve.

Both garbage collection and wear leveling play a critical role in longevity and reliability of flash memory. Since both algorithms are fundamentally based on hot and cold data identification scheme, an efficient hot data identification scheme is very critical in flash memory design.

2.2 Hot Data Identification Schemes

This subsection describes the existing hot data identification schemes and explores their merits and demerits.

Chiang et al. [27] classified data into read-only, hot, and cold data for designing a garbage collection algorithm. This scheme separates read-only data from other data and chose a block containing many hot data since hot data will soon become invalidated, which will result in higher reclamation efficiency. For data classification, this scheme keeps track of the hotness of each LBA (Logical Block Address) in a lookup table by maintaining some information such as data access timestamp, counters, valid block counters, etc. However, this approach introduces significantly high memory consumption.
Chang et al. proposed a two-level LRU (Least Recently Used) discipline for hot and cold identification [28]. As shown in Figure 2, it maintains two fixed-length LRU lists for LBAs: one for a hot data list and another for a hot candidate list. When the Flash Translation Layer (FTL) receives a write request, this scheme checks whether the requested LBA already exists in the first-level LRU list (i.e., a hot list) or not. If it is found in the hot list, the data are considered hot data, otherwise, cold data. If the LBA hits the candidate list instead, it is promoted to the hot list. This approach reduces memory space consumption by keeping track of only hot or potentially hot LBAs. However, it still requires considerable running overheads to emulate the LRU discipline and its performance is sensitive to the sizes of both lists.

To resolve memory space overheads and computational complexity, Hsieh et al. proposed a Multi-hash function scheme by adopting a counting bloom filter [29]. Since this is the state-of-the-art algorithm, we describe it in more detail. A counting bloom filter (for short, BF) is an extension of a basic BF. The basic BF is widely used for a compact representation of a set of items. According to Border’s definition [32], a BF for representing a set \( S = \{x_1, x_2, x_3, \cdots, x_n\} \) of \( n \) items is described by \( m \) bits, all bits are initially set to 0. A BF employs \( k \) independent hash functions \( h_1, h_2, \cdots, h_k \) with a range \( 1, 2, \cdots, m \). Each item \( x_i \) passing through the hash functions is remembered by marking the \( k \) bits corresponding to the \( k \) hash values. If all \( k \) bits have already been marked to 1, it indicates that the item \( x_i \) has appeared before, assuming \( m \) is large enough. Otherwise (i.e., at least one bit out of \( k \) bits has not been marked yet), \( x_i \) is regarded as a new item. Although a basic BF can indicate whether an item \( x \) has appeared before or not, it cannot provide the frequency information (i.e., how many times \( x \) has appeared).

To capture this frequency information, a counting BF uses \( m \) counters instead of an array of \( m \) bits. Consequently, when an item \( x_i \) appears, all corresponding counter values are incremented by 1. Among all corresponding counter values, the minimum counter value indicates an upper bound of the frequency of \( x_i \), because each of them can be also incremented by other items. If this minimum frequency value is greater than a threshold value, \( x_i \) is reported as hot data.

As shown in Figure 3, the Multi-hash function scheme adopted this basic counting BF concept to identify hot data. When a write request is issued, its LBA is hashed by \( k \)-hash functions and the \( k \)-hash values correspond to their bit positions of the BF. Then, the corresponding counters are increased by 1. If all \( K \)-counter values are larger than the predefined hot threshold value, the data are classified as hot data (note: if any one bit of \( D \)-most significant bits is set to 1, this means its counter value is greater than or equal to at least \( 2^{D - B} \)).

This scheme achieves less memory space consumption and lower computational complexity; however, an appropriate bloom filter size is its big concern. The bloom filter size is closely related to the number of a unique LBA to be accommodated, which depends heavily on the workload characteristics. If the bloom filter size is not properly configured, its performance is severely degraded (this will be addressed in our experiment section in detail). Moreover, its decay interval is based on the average hot data ratios in workloads, which cannot be provided to the on-line algorithm in advance.

Hsieh et al. also presented an approximated hot data identification algorithm named a direct address method (hereafter, we refer to this as DAM) as their baseline algorithm. DAM assumes that the unlimited memory space is available to keep track of hot data. It maintains a counter for all LBAs and periodically decays each LBA counter value by dividing by two at once.

### 3 HotDataTrap

This section describes our proposed hot data identification scheme named HotDataTrap.

#### 3.1 Architecture

Most of the I/O traces exhibit localities [11, 9] and in case of sequential accesses, only a few LSBs (Least Significant Bits) of LBAs are changed, while most of the other bits of them are not affected. The basic architecture of HotDataTrap is inspired by these observations.

HotDataTrap caches a set of items in order to identify hot items. For each item, it maintains an ID, a counter, and a recency bit (shown in Figure 4). The counter is used to keep track of the frequency information, while the recency bit is adopted to check whether the item has been recently accessed or not. To reduce memory consumption, HotDataTrap uses only partial bits (16 bits) out of 32-bit LBA (assuming LBA is composed of 32 bits) to identify LBAs. This 16-bit ID consists of a primary ID (12 bits) and a sub ID (4 bits).

Our proposed scheme adopts two hash functions: one for a primary ID access and the other for a sub ID access. As shown in Figure 4, one hash function for the sub ID access captures only the last 4 LSBs, while the other one for the
primary ID access takes the following 12 LSBs. This two-level hierarchical hash indexing scheme can considerably reduce cache lookup overheads by allowing HotDataTrap to directly access LBA information in the cache. Moreover, it empowers HotDataTrap to exploit spatial localities. That is, many access patterns in workloads generally exhibit high spatial localities as well as temporal localities [9]. For sequential accesses, HotDataTrap stores the starting LBA in a sequential access to both the primary ID and the sub ID, and its offset addresses only to the sub IDs, not to the primary IDs. Consequently, it can significantly reduce memory space consumption.

However, this partial LBA capturing can cause a false LBA identification problem. To verify that our partial ID can provide enough capabilities to appropriately identify LBAs, we made an attempt to analyze four workloads (i.e., Financial1, MSR, Distilled, RealSSD). These traces will be explained in detail in our evaluation section (Section 4.3).

Figure 5 shows false LBA identification rates for each trace. We first captured the aforementioned 16 LSBs to identify LBAs and then made a one-to-one comparison with the whole 32-bit LBAs. As plotted in Figure 5, MSR, Distilled, and RealSSD traces present very low false LBA identification rates, which means we can use only 16 LSBs to identify LBAs. On the other hand, Financial1 trace file shows a different characteristic from the others. When we captured the 16 LSBs of the Financial1 traces, it exhibited a relatively higher false identification rate than those of the other three traces. However, when we adopted 24 LSBs instead, it showed a zero false identification rate, which means the 24 LSBs can perfectly identify all LBAs in the Financial1 traces. To clarify this reason, we made another analysis of those four traces.

Figure 6 represents a 100% stacked column chart that is generally used to emphasize the proportion of each data. As displayed in this figure, unlike the others, the Financial1 traces tend to access only a very limited range of LBAs (a dotted square part), whereas MSR, Distilled, and RealSSD traces access much wider ranges of LBAs. This means Financial1 traces intensively access only a very limited space so that the variance of all accessed LBAs in the workloads is even smaller than the others. Therefore, capturing 16 LSBs in Financial1 causes a relatively higher false LBA identification rate than the others. In this case, if we need a more precise identification performance according to the applications, we can increase our primary ID size, not the sub ID size. Even though we increase our primary ID size, we still can save a memory space with the sub ID by exploiting spatial localities. Moreover, although HotDataTrap adopts 16 LSBs capturing in our experiments, our scheme outperforms the state-of-the-art scheme even in the Financial1 traces. This will be verified in our performance evaluation section (Section 4.3) later.

3.2 Overall Working Processes

HotDataTrap works as follows: whenever a write request is issued, the request LBA is hashed by two hash functions as described in the section 3.1 to check if the LBA has stored in the cache. If the request LBA hits the cache, the corresponding counter value is incremented by 1 to capture the frequency and the recency bit is set to 1 for recency capturing. If the counter value is greater than or equal to the predefined hit threshold value, it is classified as hot data, otherwise cold data. In case of a cache miss, HotDataTrap makes a decision on inserting this new request LBA into the cache by using our sampling-based approach.

Unlike other hot data identification schemes or typical cache algorithms which initially insert all missed items into the cache and then selectively evict a useless item from the cache.
afterwards, our HotDataTrap initially stores only selected items in the cache from the beginning. Due to a limited cache space, it would be ideal if we could store only an item that will become hot in near future. However, it is impossible unless we can see the future. Even though we try to predict the future accesses based on the access history, we must pay extra overheads to manage considerable amount of the past access information. This is not a reasonable solution especially for hot data identification scheme because it must be triggered whenever every write request is issued.

To resolve this problem, HotDataTrap tries to do sampling with 50% probability which is conceptually equivalent to tossing a coin. This simple idea is inspired by the intuition that if any LBAs are frequently accessed, they are highly likely to pass this sampling. This sampling-based approach helps HotDataTrap early discard an infrequently accessed LBA. It, consequently, can reduce not only memory consumption, but also computational overheads. The effectiveness of this sampling-based approach will be verified in our experiments (Section 4.3).

A hot data identification scheme needs an aging mechanism which decays the frequency information. Like the Multi-hash function scheme [29], HotDataTrap periodically divides the access counter values by two for its aging mechanism, which enables hot data identification schemes to partially forget their frequency information as time goes on. Thus, if any data have been heavily accessed in the past, but not afterwards, they will eventually become cold data and will be selected as victims in the cache soon.

Moreover, HotDataTrap retains another mechanism for better hot data identification by capturing recency as well as frequency information. Both a recency capturing mechanism and an aging mechanism look similar, but are clearly different: the main goal of the aging mechanism is to regularly decay the frequency information over time as well as to prevent a counter overflow problem. On the other hand, a recency capturing mechanism intends to reflect recent access information. Thus, the aging mechanism in Multi-hash cannot appropriately capture recency information. For instance, the Multi-hash scheme does not provide any mechanism to check if any data have been recently accessed because its aging mechanism simply divides all access counts by two periodically. However, the aging mechanism in HotDataTrap periodically resets all recency bits to 0, and its recency capturing mechanism sets the corresponding recency bit to 1 if any data are accessed.

Figure 7 illustrates our victim selection process. If the cache is full and we need to insert a newly sampled LBA into the cache, HotDataTrap chooses a victim. If we perform a sequential search in the cache for the victim selection, it causes significant overheads. To reduce these overheads, HotDataTrap maintains a victim candidate list. If an access counter value is smaller than the predefined hot threshold value and its recency bit is reset to 0, such LBA is classified into a victim candidate. Whenever HotDataTrap executes its decay process, it stores those victim candidates into the list. That is, the victim list is periodically updated for each decay period to reflect the latest information. Thus, when HotDataTrap needs to evict a cold item from the cache, it first selects one victim candidate from the list and directly checks if the candidate can be removed (i.e., it should meet two aforementioned requirements for an eviction). If the candidate is still cold, HotDataTrap deletes the candidate and inserts the new item into the cache. If the candidate has turned to hot data since the last decay period, we cannot evict it. In this case, HotDataTrap checks another candidate from the list.

The victim candidate list can notably save the victim search overheads but does not guarantee that all candidates in the list are currently victims in the cache because the cold data (i.e., victims at that time) may have changed to hot data since the last decay period. To reduce this error, HotDataTrap periodically updates the candidate information for each decay period. However, even though there may exist errors in the victim selection, they are not so much common because, in general, most of the workloads do not dramatically change their access patterns or workload characteristics and are likely to retain localities [11].

This victim candidate list enables HotDataTrap to directly look up a victim by using aforementioned two-level hierarchical hash indexing scheme, which can significantly reduce victim search overheads.

---

**Algorithm 1 HotDataTrap**

1: Input: write requests for LBAs
2: Output: hot or cold data classification of the LBAs
3: A write request for an LBA $x$ is issued
4: if (Cache Hit) then
5: Increase the counter for $x$ by 1
6: Set the recency bit to 1
7: if (Counter $\geq$ HotThreshold) then
8: Classify $x$ as hot data
9: end if
10: else
11: // Cache Miss
12: if (Passed a sampling test) then
13: if (Cache is not full) then
14: Insert $x$ into the cache
15: Set the counter for $x$ to 1
16: Set the recency bit to 1
17: else
18: // Need to evict data
19: while (The current candidate is not a victim) do
20: Move to the next candidate in the list
21: Check if the candidate can be evicted
22: end while
23: Evict the victim and insert $x$ into the cache
24: Set the counter for $x$ to 1
25: Set the recency bit to 1
26: end if
27: else
28: // If failed in the sampling test
29: Skip further processing of the $x$
30: end if
31: end if
### 3.3 Algorithm

Algorithm 1 provides a pseudocode of our HotDataTrap algorithm. When a write request is issued, if the LBA hits the cache, its corresponding access counter is incremented by 1 and its recency bit is set to 1. If the counter value is equal to or greater than a predefined HotThreshold value, it is identified as hot data, otherwise, cold data. If it does not hit the cache, HotDataTrap tries to do sampling by generating a random number between 0 and 1. If the number passes the sampling test (i.e., belongs to its sampling range), the LBA is ready to be added into the cache. If the cache has a space available, HotDataTrap just inserts it into the cache and sets both the access counter and recency bit to 1. Otherwise, HotDataTrap needs to select and remove a victim whose access counter value is less than HotThreshold value and receny bit was reset to 0. Then, it stores the newly sampled LBA into the cache. If the new request fails to pass the sampling test, HotDataTrap simply discards it.

### 4 Performance Evaluation

This section provides extensive experimental results and comparative analyses.

#### 4.1 Evaluation Setup

We compare our HotDataTrap with a Multi-hash function scheme [29] that is the state-of-the-art scheme and a direct address method (refer to as DAM) [29]. We select a freezing approach (i.e., when an LBA access counter reaches its maximum value due to a heavy access, it does not increase the counter any more even though the corresponding LBA is continuously accessed) as a solution for a counter overflow problem both in Multi-hash scheme and HotDataTrap since it, according to our experiments, showed a better performance than the other approach (i.e., exponential batch decay: divides all counters by two at once). DAM is an aforementioned baseline algorithm. Table 1 describes system parameters and their values.

For fair evaluation, we adopt an identical decay interval (4,096 write requests) as well as aging mechanism (i.e., exponential batch decay) for all those schemes. For more objective evaluation, we employ four realistic workloads (Table 2). Financial1 represents OLTP application traces of enterprise servers at Microsoft Research Cambridge Lab [34]. We selected, in particular, prn volume 0 trace since it exhibits a write intensive workload [35]. Lastly, we adopt 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 2 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 1000 5. This means write data into 5 consecutive LBAs from the LBA 1000’. In this case, we regard this request as 5 write requests in our experiments.

#### 4.2 Performance Metrics

We first select a hot ratio to evaluate each performance of those schemes. A hot ratio represents a ratio of hot data to all data. Thus, by comparing each hot ratio of both schemes (HotDataTrap and Multi-hash) with that of the baseline scheme (DAM), we can observe their closeness to the baseline result. However, even though both hot ratios of two schemes are identical, hot data classification results of both schemes may not be able to be identical since an identical hot ratio means the same number of hot data to all data, which does not necessarily mean all classification results are identical. Thus, we need another different metric to make up for this limitation. To meet this requirement, we adopt a false identification rate. Whenever write requests are issued, we try to make a one-to-one comparison between two hot data identification results of each scheme. This allows us to make a more precise analysis of them. Memory consumption is another important factor to be discussed since the SRAM size is very limited in flash memory and we need to exploit it. Lastly, we must take runtime overheads into account. To evaluate these overheads, we measure CPU clock cycles per each representative operation.

#### 4.3 Evaluation Results

- **Overall Performance:** We first start to discuss hot ratios of each scheme under four realistic workloads. As shown in Figure 8, our HotDataTrap exhibits very similar hot ratios to the baseline scheme (DAM), while the Multi-hash scheme has a tendency to show relatively even higher hot ratios than DAM. This results from the inherent limitation of a bloom filter-based scheme. If a bloom filter size is not large enough to accommodate all distinct input values, the bloom
filter necessarily causes a false positive problem. There exists a fundamental trade-off between a memory space (i.e., a bloom filter size) and its performance (i.e., a false positive rate) of the bloom filter-based scheme. To verify this, we measure the number of LBA collision for each unit period (i.e., decay period) under those traces. In other words, although the working space of the LBAs is much larger than the size of bloom filter (4,096) in the Multi-hash scheme, all of them have to be mapped to the bloom filter by using hash functions. Thus, many of them must be overlapped to the same positions of the bloom filter so that it causes hash collisions. As plotted in the Figure 10, we can observe many collisions (on average, 877) for each unit request (4,096 writes). All of these collisions do not necessarily mean the possibility by adopting multiple hash functions. However, considering this factor, many collisions necessarily have considerable effect on false identification rates in the Multi-hash scheme. Figure 9 demonstrates well this observation by showing that Multi-hash shows much higher false identification rates than the others. Moreover, according to the Figure 5 in Section 3.1, MSR also accesses a narrower working space than Distilled and RealSSD (even though it is wider than Financial1). Figure 9 (b) also clearly supports this observation by showing that its overall false hot data identification rate is slightly higher than that of Distilled and RealSSD in HotDataTrap. However, even though these two traces (particularly, Financial1) intensively accesses only a very limited working space, it shows a relatively higher false LBA identification rate than the other traces when we capture the 16 LSBs (Least Significant Bits) to identify LBAs. Figure 9 verifies our aforementioned discussion: Financial1 shows a higher false hot data identification rate than the others. Moreover, according to the Figure 5 in Section 3.1, MSR also accesses a narrower working space than Distilled and RealSSD (even though it is wider than Financial1). Figure 9 (b) also clearly supports this observation by showing that its overall false hot data identification rate is slightly higher than that of Distilled and RealSSD in HotDataTrap. However, even though these two traces (particularly, Financial1) display higher false identification rates, they still outperform the Multi-hash.

**Runtime Overheads:** We evaluate runtime overheads of two major operations in hot data identification scheme: a checkup and decay operation. A checkup operation corresponds to a verification process to check whether the data are hot or not. A decay operation is the aforementioned aging mechanism that regularly divides all access counts by two. To evaluate the runtime overheads, we measure CPU clock cycles of them. They are measured on the system with AMD X2 3800+ (2GHz, Dual Core) and 2G RAM under
Windows XP Professional platform. For a fair and precise measurement, we fed 100K numbers of write requests from Financial1 traces to each scheme and ran 100 times of each operation to measure average CPU clock cycles. Since the CPU clock count depends heavily on level-1 cache in the CPU, if both codes and data are not in the level-1 cache, the CPU clock count is enormously high. This is the main reason the CPU clock count at the first run is excessively higher than the subsequent counts that represent the time it takes when everything is in the level-1 cache.

As shown in Figure 11 (a), the runtime overheads of a checkup operation in HotDataTrap is almost comparable to that of the Multi-hash checkup operation. Although the HotDataTrap requires a few more cycles than Multi-hash scheme, considering both standard deviations (28.2 for Multi-hash and 25.6 for HotDataTrap) for each scheme, they can be ignorable. This result is intuitive because both schemes not only adopt the same number of hash functions, but also can directly look up and check the data in the cache. However, the decay operation of HotDataTrap requires much lower (40.7%) runtime overheads than that of Multi-hash since Multi-hash must decay 4,096 entries, while HotDataTrap only has to treat even less number of entries (approximately 1/3 of the Multi-hash, but it depends on workload characteristics) for each decay interval. For each entry, Multi-hash requires 4 bits, while our scheme needs 20 bits or 8 bits. Assuming HotDataTrap entries are 1/3 of Multi-hash entries to decay, its runtime overheads would also be around 1/3 of Multi-hash overheads. However, our scheme requires extra processes to update its victim candidate list during the decay process. Thus, its overall runtime overheads reach almost 60% of Multi-hash scheme’s. In addition, HotDataTrap must consider extra overheads: cache management. Multi-hash does not retain this overhead since it uses a bloom filter, while our scheme manages a set of potential hot items in the cache. However, we can significantly reduce these overheads with the help of our sampling approach since the requests failing to pass the sampling test will be discarded without any further process. Figure 11 (b) plots the impact of various sampling rates. As the sampling rate grows, the runtime overheads also increase.

- **Impact of Memory Spaces**: In this subsection, we assign a different memory space to each scheme from 0.5KB to 8KB and observe their impacts on both schemes. The Multi-hash scheme consumes 2KB (4 × 4,096 bits). As plotted in Figure 12 (a), both schemes exhibit a performance improvement as a memory space grows. However, HotDataTrap outperforms Multi-hash, and moreover, the Multi-hash scheme is more sensitive to a memory space. This result is closely related to the false positives of a bloom filter since the false positive probability decreases as a bloom filter size increases. On the other hand, HotDataTrap shows a very stable performance even with a smaller memory space, which means our HotDataTrap can be well employed even in embedded system devices with a very limited memory space. Figure 12 (b) exhibits hot ratios of each scheme over various memory spaces. All hot ratios increase as a memory size increases because their decay periods also increase accordingly. A longer decay period allows each scheme to accept more requests, which causes higher hot ratios.

- **Impact of Decay Intervals**: A decay interval has a significant impact on the overall performance. Thus, finding an appropriate decay interval is another important issue in the hot data identification scheme design. Multi-hash adopts a formula, \( N \leq M/(1 - R) \), as its decay interval, where \( N \) is a decay interval, \( M \) and \( R \) correspond to its bloom filter size and an average hot ratio of a workload respectively. The authors say this formula 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 \( N \) write request (they assumed \( R = 20\% \)). According to the formula, assuming the bloom filter size \( M \) is 4,096 bits, they suggested 5,117 write requests as their optimal decay interval. To verify their assumption, we tried to measure the number of a unique LBA for every 5,117 write request under all those four traces. According to our observation (due to a space limit, we do not display this plot here), a large number of unique LBAs exceeded the bloom filter size of 4,096, which necessarily causes many hash collisions so that it results in a higher false positive probability. Moreover, initially fixing an average hot ratio of a workload to 20% is not reasonable since each workload has different characteristics and we cannot know their average hot ratios in advance. Lastly, even though the authors recommended 5,117 writes as their optimal decay interval, our experiments demonstrated that Multi-hash showed a better performance when we adopted 4,096 writes as its decay interval. Thus, we chose 4,096 writes as its decay interval due to not only a performance issue, but also a fairness issue.

Now, in order to observe the impact of a decay interval on each scheme, we fix the memory space (2KB, i.e., a bloom filter size of 4,096 bits for Multi-hash) for each scheme and change the decay interval from 2,048 to 16,384 write requests. Figure 13 displays a performance change over diverse decay intervals. As shown in Figure 13 (b), the performance of Multi-hash is very sensitive to the decay intervals. Its false
hot data identification rate almost exponentially increases as a decay interval grows. For instance, when the interval increases 4 times (from 4,096 to 16,384), its false identification rate increases approximately 10 times (9.93 times). In particular, the false identification rate reaches even on average 72.4% if the decay interval is 16,384. This results from a higher false positive probability of the bloom filter. On the other side, our HotDataTrap shows a stable performance irrespective of decay intervals (Figure 13 (a)), which demonstrates well that the decay interval does not have a critical influence on HotDataTrap performance. Therefore, even though we cannot find an optimal decay interval, it is not a critical issue in HotDataTrap unlike Multi-hash.

- **Effectiveness of Hierarchical Hash Indexing:** HotDataTrap is designed to exploit spatial localities in data accesses by using its two-level hierarchical hash indexing scheme. In this subsection, we explore how much we can benefit from it. To evaluate its benefit, we prepared two different HotDataTrap schemes: one for our original HotDataTrap (which initially retains the indexing scheme) and the other for HotDataTrap without two-level indexing scheme (referred to as HotDataTrap_WT). That is, HotDataTrap_WT always use a primary ID as well as its sub ID even though the traces show sequential accesses, which can require a more memory space than the HotDataTrap. Figure 14 compares both performances and demonstrates the effectiveness of our two-level hierarchical hash indexing scheme. As shown in the plots, HotDataTrap improves its performance, by an average of 19.7% and up to 50.8% thereby lowering false hot data identification.

- **Impact of Sampling Rates:** When a request LBA does not hit the cache, it may or may not be inserted into the cache according to a sampling rate of HotDataTrap. Thus, the sampling rate is another factor to affect an overall HotDataTrap performance. Figure 15 exhibits a performance change for each sampling rate under a variety of traces and Figure 16 plots overall impacts of the sampling rates. As shown in Figure 16 (a), average hot ratios for each trace gradually approach to those of DAM since a higher sampling rate can reduce a chance to drop potential hot items by mistake. Although this higher sampling may cause higher running overheads due to a more frequent cache management process (Figure 11), it can be expected to achieve a better performance as displayed in Figure 16 (b). False identification rates, intuitively, decrease as the sampling rate grows. Since HotDataTrap with 10% sampling rate shows an even better performance than the Multi-hash scheme by an average of 65.1%, it is acceptable that we select a 50% sampling rate as our initial sampling rate for HotDataTrap. Instead, we can choose a different sampling rate according to application requirements. If an application requires a precise hot data identification performance at the expense of runtime overheads, it can adopt a 100% sampling rate, not lower sampling rates, and vice versa.

5 Conclusion

Hot data identification is of paramount importance in flash-based storage devices since not only does it have a great impact on the overall performance of them, but also it has a big potential to be applicable to many other fields. We, in this paper, proposed a novel efficient on-line hot data identifica-
tion algorithm named HotDataTrap. HotDataTrap adopts a sampling mechanism unlike other existing schemes. This sampling approach helps HotDataTrap early discard some of the cold items so that it can reduce runtime overheads as well as a waste of a memory space. Moreover, its two-level hierarchical hash indexing enables HotDataTrap to directly look up items in the cache and to reduce a memory space further by exploiting spatial localities.

We made diverse experiments in many respects with various realistic workloads. Based on various performance metrics, we carried out experiments on the runtime overheads and the impacts of memory sizes, sampling rates, and decay periods. In addition, we also explored the effectiveness of our hierarchical indexing scheme by comparing two different HotDataTrap schemes with and without the indexing scheme respectively. Our diverse experiments presented that HotDataTrap achieved a better performance up to a factor of 20 and by an average of 335%. The two-level hash indexing scheme improved a HotDataTrap performance further up to 50.8%. Moreover, HotDataTrap was not sensitive to its decay interval as well as a memory space. Consequently, HotDataTrap can be well employed even in small embedded devices with a very limited memory space.

6 Acknowledgments

This work was partially supported by grants from NSF (NSF Awards: 0960833, 0934396, 1127829 and 115471). This research was also supported in part by GLORY project from the Ministry of Knowledge Economy (MKE: Grant No. K1001703), Korea. GLORY is named from GLObal Re-Source conservation, enhancing a hybrid flash-memory storage systems through resource conservation, “ACM Transactions on Storage”, vol. 1, no. 4, 2005.


7 References


[34] “SNIA IOTTA Repository: MSR Cambridge Block I/O Traces,” http://iotta.snia.org/traces/list/BlockIO.