Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 11-009

HotDataTrap: A Sampling-based Hot Data Identification Scheme for Flash Memory

Dongchul Park, Biplob Debnath, Young Jin Nam, David Du,
Youngkyun Kim, and Youngchul Kim

April 20, 2011

HotDataTrap: A Sampling-based Hot Data Identification Scheme for Flash Memory

Dongchul Park, Biplob Debnath,
Young Jin Nam and David H.C. Du
University of Minnesota–Twin Cities, USA
park@cs.umn.edu, biplob@umn.edu,
{youngjin, du}@cs.umn.edu

Youngkyun Kim and Youngchul Kim
Electronics and Telecommunications Research
Institute (ETRI), Korea
{kimyoung, kimyc}@etri.re.kr

ABSTRACT

Hot data identification is an issue of paramount importance in flash-based storage devices since it has a great impact on their overall performance as well as retains a big potential to be applicable to many other fields. However, it has been least investigated. In this paper, we propose a novel on-line hot data identification scheme named HotDataTrap. The main idea is to maintain a working set of potential hot data items in a cache based on a sampling approach. This sampling scheme enables HotDataTrap to early discard some of the cold items so that it can reduce runtime overheads and a waste of memory spaces. Moreover, our two-level hierarchical 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 even with a very limited memory space. 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%.

1. INTRODUCTION

These days, flash-based storage devices 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 with respect to write operations. 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 resolve these limitations, a flash-based storage device adopts an intermediate software/hardware layer named Flash Translation Layer (FTL). The FTL treats flash memory as a log device; so it stores updated data to a new physical page, marks the old physical page as invalid for future reclamation (called a garbage collection), and maintains logical-to-physical page address mapping information to keep track of the latest location of the logical page [8]. Periodically, FTL invokes a garbage collection algorithm to reclaim those outdated spaces and employs a wear leveling algorithm to improve the life span of the flash memory by evenly distributing block erases over the entire flash memory space.

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 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 recently 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.

In addition to these applications in flash memory, 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. 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 [19] or incur excessive computational overheads [20]. To overcome these shortcomings, Hsieh et al. recently proposed a multiple hash function framework [21]. This adopts a counting bloom filter and multiple hash functions so that it can reduce memory consumption as well as computational overheads. However, this scheme considers simply a frequency factor by maintaining access counters and cannot achieve high accuracy of the hot data identification 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. Although a few years have passed since this scheme was proposed, to the best of our knowledge, this is the state-of-the-art scheme.

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 a recency bit like a CLOCK algorithm [22]. It periodically resets the recency information and decays the frequency information, which is an aging mechanism. Thus, if data have not been accessed for a long time, eventually they are considered cold data and will become victim candidates for an eviction from the cache. Moreover, HotDataTrap can achieve direct cache lookup as well as lower memory consumption with the help of its two-level hierarchical hash indexing scheme. Consequently, our HotDataTrap provides more accurate hot data identification than the state-of-the-art scheme [21] even with a very limited memory. The main contributions of this paper are as follows:

- ***Sampling-based Hot Data Identification:*** Unlike other schemes, HotDataTrap does not initially insert all cache-missed items into a cache. Instead, it decides whether to cache them or not with our simple sampling algorithm. Once the item 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 approach enables HotDataTrap to early discard some of the cold items so that it can reduce computational overheads and memory consumption.
- ***Two-Level Hierarchical Hash Indexing:*** HotDataTrap takes advantage of spatial localities. For sequential accesses, HotDataTrap maintains only their starting (partial) LBAs and its offset information by adopting an hierarchical data structure and two hash functions. These enable HotDataTrap to achieve low runtime overheads (due to its direct cache lookup) as well as lower memory consumption.

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. BACKGROUND AND RELATED WORK

This section describes NAND flash memory characteristics and existing hot data identification schemes for flash memroy. 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 ($1{,}500\mu s$) is the most expensive operation in flash memory compared to data read ($25\mu s$) and write ($200\mu s$) [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 [23]. This garbage collection is performed on a block basis; so all valid pages in the victim block must be copied to other clean

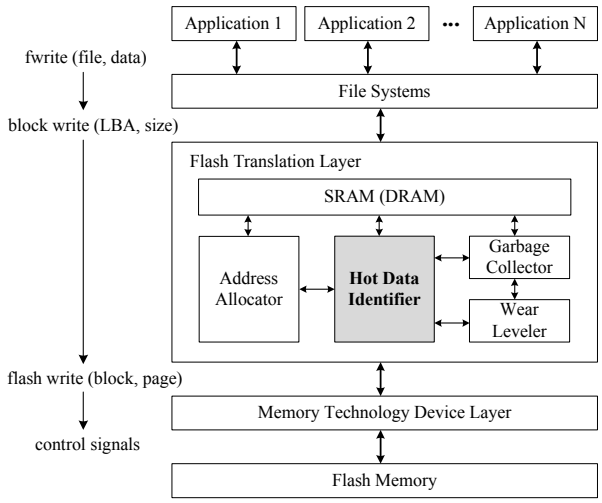


Figure 1: Typical System Architecture of Flash Memory-based Storage

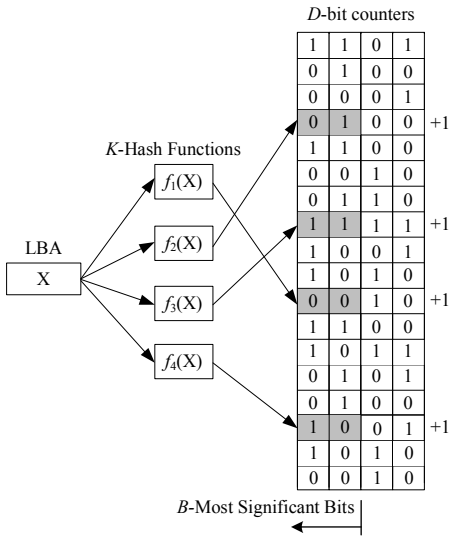


Figure 2: Multi-hash Function Framework (K=4, D=4, and B=2)

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.

2.2 Hot Data Identification Schemes

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

Chiang et al. [19] 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 [20]. 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, the two-level LRU list 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 first-level list, the data are considered hot data, otherwise, cold data. The second-level LRU list (i.e., a candidate list) maintains the LBAs of recently written data. If those LBAs are written again within a short period of time, the LBAs are classified as hot. Thus, those are promoted from the candidate list to the hot list. If the hot list is full of hot LBAs, the last entry (i.e., tail) in the hot list is demoted to the first (i.e., head) slot in the candidate list. This approach tries to reduce 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.

To resolve memory space overheads and computational complexity, Hsieh et al. proposed a Multi-hash function scheme by adopting a counting bloom filter [21]. 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 [24], 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.

As shown in Figure 2, 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 B-most significant bits is set to 1, this means its counter value is greater than or equal to at least 2^{D-B}). Although this Multi-hash function scheme can capture well the frequency information, it does not provide an explicit recency capturing mechanism. Instead, it provides an aging mechanism by dividing all counter values

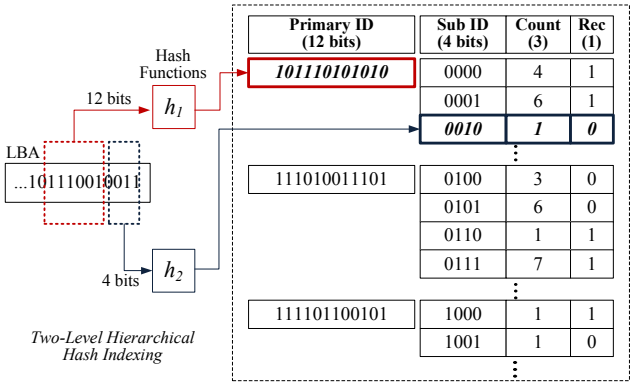


Figure 3: HotDataTrap Framework

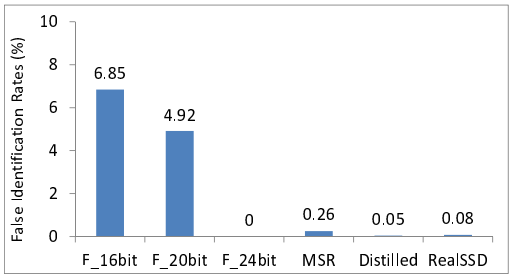


Figure 4: False LBA Identification Rates for Each Workload (F_16(20, 24)bit stands for Financial1 traces capturing 16(20, 24) LSBs in LBAs.)

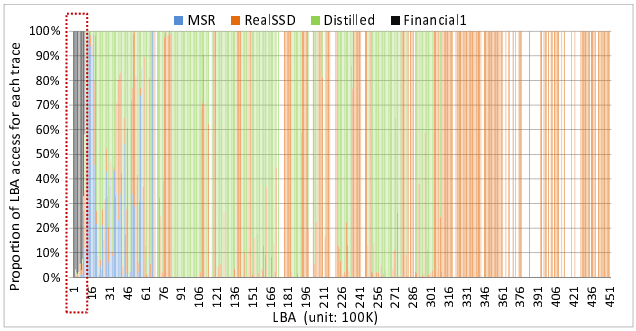


Figure 5: Proportion of LBA Access for Each Trace

by two periodically.

This scheme achieved lower memory space consumption and low computational complexity; however, an appropriate BF size is its big concern. The BF size is closely related to the number of a unique LBA to be accommodated, which depends heavily on the workload characteristics. If the BF 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. HOT DATA TRAP

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 3). 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 3, 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, this hierarchical indexing scheme 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.

HotDataTrap captures only the aforementioned 16 LSBs of LBAs to identify LBA information in a cache. 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., Financial 1, MSR, Distilled, RealSSD). These traces will be explained in detail in our evaluation section (Section 4.3).

Figure 4 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 4, 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 5 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 hot 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, our HotDataTrap initially stores only selected items in the cache from the beginning. In other words, most of them have interest only in a victim selection (i.e., how to effectively choose and evict a useless item from the cache), not in an item insertion into the cache (i.e., which item must be initially inserted into the cache). Since our main goal is to identify whether the requested data are hot or not, HotDataTrap does not insert all missed LBAs in the cache. 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 almost impossible unless we can see the future. Even though we make an attempt to predict the future accesses based on the access history, we have to pay extra overheads to manage considerable amount of the past access information. This is not a reasonable solution especially for the 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).

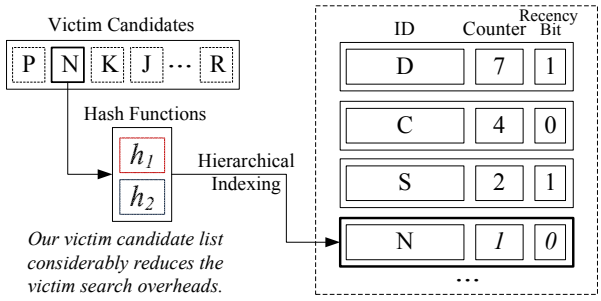


Figure 6: HotDataTrap Victim Selection

A hot data identification scheme needs an aging mechanism which decays the frequency information. Like the Multi-hash function scheme [21], 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. Unlike other schemes, our HotDataTrap resets all recency bits to 0 in order to reflect a recency factor. Thus, if any item has been heavily accessed in the past, but not recently, it will eventually become cold data and will be selected as a victim in the cache soon.

Figure 6 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 (i.e., can be evicted), 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 and follows the same process described above.

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 our 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 ≥ 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 recency 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 [21] that is the state-of-the-art scheme and a direct address method (refer to as DAM) [21]. 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.

Table 1: System Parameters

System Parameters	Values
Bloom Filter Size	2^{11}
Decay Interval	2^{11}
Hot Threshold	4
Number of Hash Function	2
Initial Memory Space	2KB

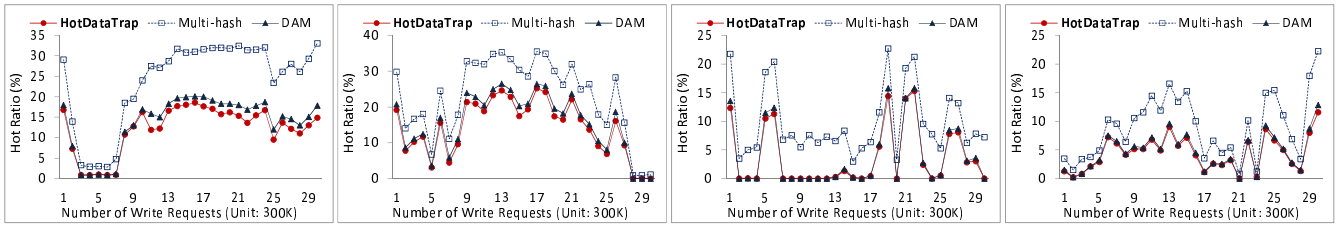
Table 2: Workload Characteristics

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

For 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 is write intensive block I/O traces from the University of Massachusetts at Amherst Storage Repository [25]. This trace file was collected from an On-Line Transaction Processing (OLTP) application running at a financial institution. Distilled trace file [20] represents a general and personal usage patterns in a laptop such as web surfing, documentation work, watching movies, playing games, etc. This is from the flash memory research group repository at National Taiwan University, and since Multi-hash scheme employed only this trace file for its evaluation, we also adopt this trace for fair comparison. We also select Microsoft Research Trace (refer to as MSR) made up of 1-week block I/O traces of enterprise servers at Microsoft Research Cambridge Lab [26]. We selected, in particular, *prn volume 0* trace since it exhibits a write intensive workload [27]. 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 requests in Table 2 can be also 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.

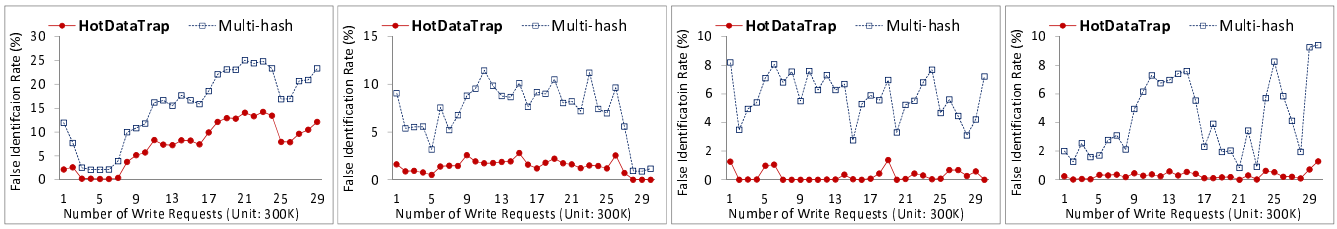
4.2 Performance Metrics

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 Multh-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,



(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Figure 7: Hot Ratios of Each Scheme under Various Traces



(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Figure 8: False Hot Data Identification Rates of Both HotDataTrap and Multi-hash

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.

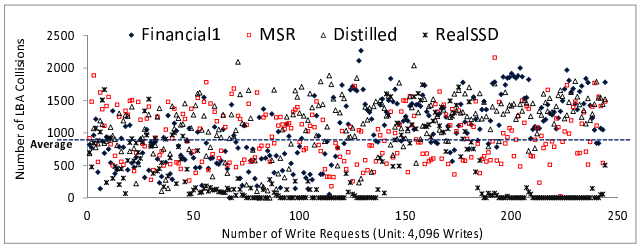


Figure 9: Number of LBA Collisions for Each Unit (4,096 writes) Period

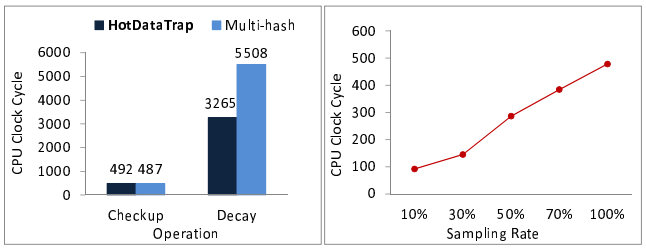
4.3 Evaluation Results

We discuss our evaluation results in diverse respects.

• ***Overall Performance:*** We first start to discuss hot ratios of each scheme under four realistic workloads. As shown in Figure 7, 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 inborn 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 9, we can observe many collisions (on average, 877) for each unit request (4,096 writes). All of these collisions do not necessarily mean the number of false identification of the Multi-hash because it can reduce 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 8 demonstrates well this analysis. Multi-hash shows much higher false identification rates than our HotDataTrap. HotDataTrap makes a even more precise decision on hot data identification by an average of 335%.

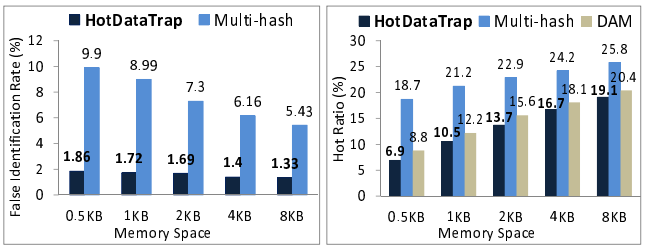
As we discussed in Section 3.1, due to the distinguished workload characteristics of Financial1 (i.e., Financial1 intensively accesses only a very limited working space), it shows a relatively higher false LBA identification rate than the other traces when we capture the 16 LSBs (Least Significant Bits) to identify LBAs. Figure 8 verifies our aforementioned discussion. In particular, Financial1 shows a higher false hot data identification rate than the others. Moreover, according to the Figure 4 in Section 3.1, MSR also accesses a narrower working space than Distilled and RealSSD (even though it is wider than Financial1). Figure 8 (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



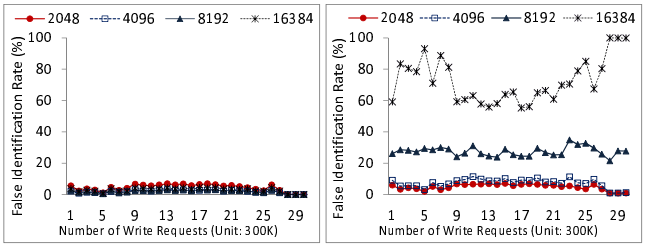
(a) Overheads for Each Operation (b) Cache Overheads

Figure 10: Average Runtime Overheads



(a) False Identification Rates (b) Hot Ratios

Figure 11: Impact of Memory Spaces



(a) HotDataTrap (b) Multi-hash

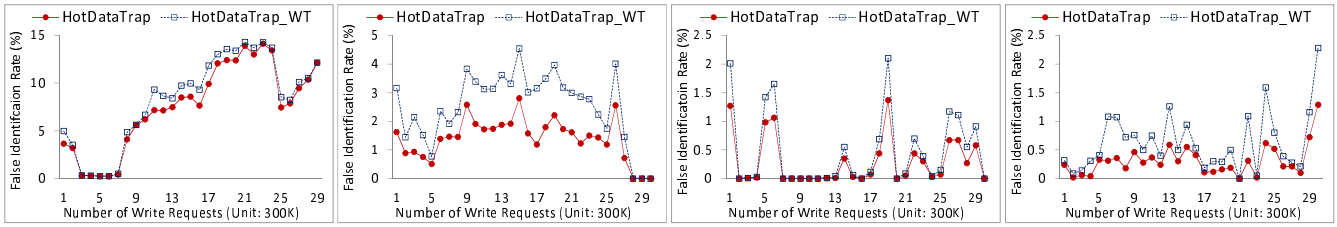
Figure 12: Impact of Decay Intervals

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 an aging mechanism that regularly divides all access counts by two. To evaluate the runtime overheads, we measure CPU clock cycles of them under the configurations in Table 1. 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 10 (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 10 (b) plots the impact of various sampling rates. As the sampling rate grows, the runtime overheads also increase.

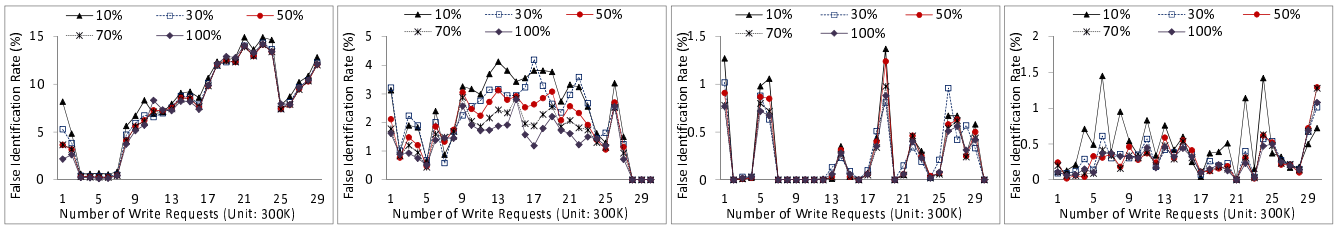
• ***Impact of Memory Spaces:*** In this subsection, we assign a different memory space to each scheme from 0.5KB to 8KB and observe their impacts on both schemes. The Multi-hash scheme consumes 2KB ($4 \times 4,096$ bits). As plotted in Figure 11 (a), both schemes exhibit a performance improvement as 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 11 (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 is 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



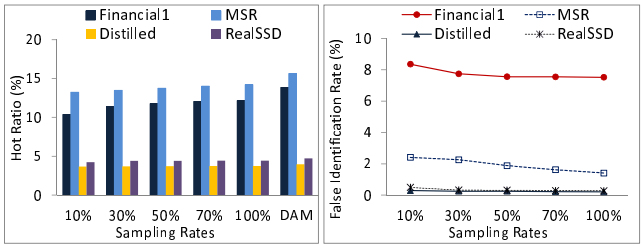
(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Figure 13: Performance Improvement with Our Two-Level Hierarchical Hash Indexing Scheme



(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Figure 14: Performance Change over Sampling Rates under Diverse Workloads



(a) Average Hot Ratios (b) Avg. False ID Rates

Figure 15: Impact of Sampling Rates

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 12 displays a performance change over diverse decay intervals. As shown in Figure 12 (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 12 (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 13 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 14 exhibits a performance change for each sampling rate under a variety of traces and Figure 15 plots overall impacts of the sampling rates. As shown in Figure 15 (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 10), it can be expected to achieve a better performance as displayed in Figure 15 (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

We, in this paper, proposed a novel on-line hot data identification scheme named HotDataTrap. HotDataTrap is a sampling-based hot data identification scheme. 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, the 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. REFERENCES

[1] A. Caulfield, L. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," in *ASPLOS*, 2009.

[2] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Survey*, vol. 37, no. 2, 2005.

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

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

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

[6] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. Najjar, "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices," in *FAST*, 2005.

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

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

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

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

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

[12] Y. Chang, J. Hsieh, and T. Kuo, "Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Leveling Design," in *DAC*, 2007.

[13] T. Kgil, D. Robert, and T. Mudge, "Improving NAND Flash Based Disk Caches," in *ISCA*, 2008.

[14] J. Mogul, E. Argollo, and M. S. P. Faraboschi, "Operating System Support for NVM+DRAM Hybrid Main Memory," in *HotOS*, 2009.

[15] L.-P. Chang, "Hybrid solid-state disks: combining heterogeneous NAND flash in large SSDs," in *ASP-DAC*, 2008.

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

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

[18] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR: integration of LRU and writes sequence reordering for flash memory," *IEEE Transactions on Consumer Electronics*, vol. 54, no. 3, 2008.

[19] M. Chiang, P. Lee, and R. Chang, "Managing flash memory in personal communication devices," in *ISCE*, 1997.

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

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

[22] F. Corbato, "A Paging Experiment with the Multics System," in *MIT Project MAC Report MAC-M-384*, 1968.

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

[24] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002.

[25] "University of Massachusetts Amhesrst Storage Traces," http: //traces.cs.umass.edu/index.php/Storage/Storage.

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

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