An Empirical Study of Hot/Cold Data Separation Policies in Solid State Drives (SSDs)

Jongsung Lee
Sungkyunkwan university
Suwon 440-746
South Korea
leitia@csl.skku.edu

Jin-Soo Kim
Sungkyunkwan university
Suwon 440-746
South Korea
jinsookim@skku.edu

ABSTRACT

Separating hot data from cold data is known to allow for efficient management of NAND flash memory in Solid State Drives (SSDs). However, most of previous work has been evaluated with the trace-driven simulations under different workloads and testing conditions. The goal of this paper is to empirically study the performance, computation overhead, and memory consumption of the existing hot/cold data separation policies on a real SSD platform. After devising a general framework where a different policy can be easily plugged in, we have evaluated three hot/cold data separation policies: 2-level LRU (LRU), Multiple Bloom Filter (MBF), and Dynamic dAta Clustering (DAC). Our evaluation results show that DAC performs best, improving the performance by up to 58% in real workloads with a reasonable computation and memory overhead.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary Storage*

General Terms

Experiment, Measurement, Performance

Keywords

Hot/Cold data separation, 2-Level LRU, Multiple Bloom Filter, Dynamic dAta Clustering

1. INTRODUCTION

Recently, solid state drives (SSDs) have been considered as the potential alternative of traditional hard disk drives (HDDs) in favor of faster performance, lower power consumption, smaller size, and better shock resistance. These benefits of SSDs are largely originated from their storage media, NAND flash memory [14]. NAND flash memory consists of a series of *blocks* and each block in turn contains 64∼256 *pages*, supporting three types of operations: read, write (or program), and erase. Pages are the unit of read/write operations, while erase operations wipe out all the pages within a block.

NAND flash memory has several restrictions to be used as a conventional block device. First, data can be written only in the clean page. The previously programmed page cannot be overwritten until the block containing the page is cleaned by the erase operation. Second, there is a limit in the number of program/erase cycles that can be performed on each block. If a block is used beyond this limit, the block is regarded as being worn out and the reliability of the block is not guaranteed.

To handle these problems, an intermediate layer called Flash Translation Layer (FTL) [7, 11] is introduced. FTL emulates the traditional block device on top of NAND flash memory, hiding the aforementioned peculiarities of NAND flash memory. Basically, FTL writes the incoming data into one of clean pages and keeps track of the mapping information between the logical address given by the host into the physical flash address. The old data becomes obsolete and it is later reclaimed by the procedure known as garbage collection (GC). As the erase operation is performed on a block basis, any valid pages contained in a block should be copied to another clean pages before the victim block is erased. The amount of valid pages copied during GC directly affects the performance and lifetime of SSDs.

It is known that separating hot data (i.e., the data which is likely to be updated soon) from cold data (i.e., the data which is not updated frequently) is very effective in improving the efficiency of GC [9]. This is because the number of valid pages within a victim block diminishes if we identify hot data and gather them in the same block. There are various studies which attempt to identify hot/cold data and use this information during GC. However, most of them evaluates their policies with the trace-driven simulations under different workloads and testing conditions.

The aim of this paper is to empirically analyze the performance of the existing hot/cold data separation policies on a real SSD platform. The policies we have evaluated include 2-level LRU (LRU) [5], Multiple Bloom Filter (MBF) [12], and Dynamic dAta Clustering (DAC) [6]. For fair comparison, we have devised a general FTL framework, on which only the hot/cold data management can be varied according to each policy, fixing other parameters such as SSD capacity, the amount of over-provisioned area, victim selection policy, etc. Using six synthetic and four real workloads, we have compared the performance, computation overhead, and memory consumption of each policy. Our evaluation results show that all the hot/cold data separation policies considered in this paper are helpful in most cases. In particular, DAC shows the best performance improving the overhead of GC, measured in WAF (Write Amplification Factor) [15], by up to 58% for real workloads with a reasonable memory and computation overhead.

The rest of this paper is organized as follows. Section 2 briefly overviews FTL. Section 3 describes LRU, MBF, and DAC in detail. Our evaluation methodology and results are presented in Section 4. Finally, Section 5 concludes the paper.

2. BACKGROUND

2.1 Flash Translation Layer

Every SSD is equipped with a special firmware layer called Flash Translation Layer (FTL). As in-place update is not allowed in NAND flash memory, FTL relies on an address mapping technique which translates the logical page number (LPN) from the host into the physical page address on NAND flash. Although various address mapping techniques are proposed for FTL, we only consider page-level mapping [8] where the mapping entry is maintained for each logical page. Due to the flexibility in NAND flash management, page-level mapping exhibits higher performance, resulting in wide spread adoption in commercial SSDs.

Algorithm 1 outlines the overall GC procedure in our general FTL framework. We assume that there is a separate *update block* $B_{Update(h)}$ associated with each data *hotness* level h. For each write request on an LPN l, FTL estimates the hotness level h of l and stores the incoming data to $B_{Update(h)}$. If $B_{Update(h)}$ is full, FTL allocates a clean block as the new $B_{Update(h)}$. In case all the clean blocks are exhausted, the GC routine in Algorithm 1 is initiated. The argument $Type$ denotes the hotness level of the required update block (set to 0 for FTLs which do not distinguish hot data from cold data).

During GC, FTL first selects a victim block by calling SelectVictim() and then reclaims it with ReclaimBlock(). The victim block is erased after each valid page is moved to the update block corresponding to its hotness level. ReclaimBlock() returns the number of clean blocks it actually generates. FTL reserves a certain number of blocks to cope with the situation where another clean block is needed while copying valid pages. In such a case, the return value of ReclaimBlock() can be less than one. The main GC routine continues block reclamation until a total of *nBlk* (1 by default) new blocks are successfully generated without changing the number of reserved blocks. Note that IsFull() in Algorithm 1 returns 1 when the block is full and 0 otherwise.

2.2 Victim selection policies

The performance of GC greatly depends on which block is selected as a victim of GC. A traditional and simple way is the greedy policy [13] where a block with the smallest number of valid pages is chosen. Another victim selection policy is the cost-benefit policy [13] which considers not only the amount of valid pages but also the age of each block. It is based on the observation that pages in young blocks (i.e., recently written blocks) are very likely to be overwritten. Under the cost-benefit policy, the block which maximizes

Algorithm 1 Garbage collection

```
 1: function GarbageCollection($Type$)
 2:     $n \leftarrow 0$
 3:     while $n < nBlk - 1 +$ IsFull($B_{Update(Type)}$) do
 4:         $B_{Victim} \leftarrow$ SelectVictim()
 5:         $n \leftarrow n +$ ReclaimBlock($B_{Victim}$)
 6:     if IsFull($B_{Update(Type)}$) then
 7:         $B_{Update(Type)} \leftarrow$ Get a reserved block
 8: function ReclaimBlock($B_{Victim}$)
 9:     $n \leftarrow 1$
10:     for all valid page P in $B_{Victim}$ do
11:         $H \leftarrow$ Hotness($LPN(P)$)
12:         if IsFull($B_{Update(H)}$) then
13:             $B_{Update(H)} \leftarrow$ Get a reserved block
14:             $n \leftarrow n - 1$
15:         Write $P$ to $B_{Update(H)}$
16:     Erase($B_{Victim}$)
17:     Add $B_{Victim}$ to reserved blocks for the next GC
18:     return $n$
```

$cb = \frac{1-u}{u} * age$ is selected as a victim, where u and age denote the utilization of the block and the age (= current time - last written time) of the block, respectively. Since the cost-benefit policy is known to perform better when hot data is mixed with cold data [13], we have used the cost-benefit policy for all experiments described in Section 4.

3. HOT/COLD DATA SEPARATION POLICIES

In this section, we briefly overview three hot/cold data separation policies evaluated in this paper.

3.1 2-Level LRU

The 2-Level LRU (LRU) policy [5] uses two LRU-based lists: the hot list and the candidate list. Each list contains LPNs. Basically, only the LPNs that are on the hot list are regarded as hot data. In this policy, when a write operation is received, FTL finds the corresponding LPN in both lists. If the LPN exists in the hot list, it is promoted to the front of the hot list to reduce the chance of being evicted. If the LPN is on the candidate list, it is deleted from the candidate list and inserted to the front of the hot list. When the LPN does not exist in any list, it is inserted to the front of the candidate list. The size of each list (HOT_SIZE or CANDIDATE_SIZE) is fixed. If the hot list is full, the last LPN in the hot list is evicted and inserted to the front of the candidate list. If the candidate list is full, the last LPN in the candidate list is simply dropped.

This policy is quite simple but has some problems. The fixed size of the hot list means that the amount of hot data is always fixed and it cannot adapt to various workloads where the amount of hot data exceeds the hot list size. Another problem is that the lookup operation in the LRU-based list for the given LPN takes a lot of time due to linear search. Algorithm 2 presents the pseudo code of Hotness_LRU() and Write_LRU() which are used in our FTL framework to implement the 2-level LRU policy. Write_LRU() is called for every write operation to adjust the LRU lists. Hotness_LRU() replaces the Hotness() function shown in line 14 of Algorithm 1.

Algorithm 2 Hotness_LRU() and Write_LRU()

```
1: function HOTNESS_LRU($Lpn$)
2:     if $Lpn \in$ HotList then
3:         return HOT
4:     else
5:         return COLD
6: function WRITE_LRU($Lpn$)
7:     if $Lpn \in$ HotList then
8:         Move $Lpn$ to the front of HotList
9:     else if $Lpn \in$ CandidateList then
10:        if |HotList| $\geq$ HOT_SIZE then
11:            $E\_Lpn \leftarrow$ Evict the last one of HotList
12:            Insert $E\_Lpn$ to the front of CandidateList
13:        Delete $Lpn$ from CandidateList
14:        Insert $Lpn$ to the front of HotList
15:    else
16:        if |CandidateList| $\geq$ CANDIDATE_LIST then
17:            Evict the last one of CandidateList
18:        Insert $Lpn$ to the front of CandidateList
```

3.2 Multiple Bloom Filter

The Multiple Bloom Filter (MBF) policy [12] uses bloom filters (BF) to test whether the current LPN belongs to a set of hot data. A BF consists of a bit array of BLOOM_FILTER_SIZE bits. Each LPN is hashed by NUM_HASH_FUNC different hash functions and the corresponding bits in the BF are set to one.

To minimize false positive errors where some cold data are wrongly classified as hot data, the MBF policy utilizes multiple BFs. The actual number of BFs (NUM_FILTERS) is configurable and it is recommended to have at least four BFs. At a certain point, one of BFs is designated as the current filter or CUR_FILTER. For each write request, if the corresponding bit in the current filter is already set to one, the MBF policy tries to set the bit in the same location of the next BF in a round-robin fashion. After every DECAY_PERIOD, the current filter is set to the BF that has not been selected in the longest time interval and all bits in that BF are erased to drop old information. Under the MBF policy, the hotness of an LPN is estimated by hashing the LPN with the same hash functions and then counting the number of bits set to one in the corresponding bit position of all BFs. If the count is over the predefined THRESHOLD for all hashed locations, the LPN is classified as hot data.

One of the advantages of the MBF policy is that it consumes a very small amount of memory. However, the performance of the MBF policy significantly depends on various factors such as the choice of hash functions, NUM_FILTERS, BLOOM_FILTER_SIZE, THRESHOLD, etc. In Algorithm 3, we outline Hotness_MBF() and Write_MBF() functions implemented in our FTL framework.

3.3 Dynamic dAta Clustering

In the LRU and MBF policies, the data is simply classified as either hot or cold. However, the Dynamic dAta Clustering (DAC) policy [6] uses the notion of *regions* to provide more fine-grained classification on the data. All LPNs are initially set to the region 0 (coldest) and it will be gradually promoted to upper regions as there are subsequent writes into the same LPN. On the other hand, when a block is selected as a victim during GC, all valid pages in the victim

Algorithm 3 Hotness_MBF() and Write_MBF()

```
1: function HOTNESS_MBF($Lpn$)
2:     $Hotness \leftarrow$ HOT
3:     for $H$ from 0 to NUM_HASH_FUNC do
4:         $Bit \leftarrow$ Hash($H, Lpn$)
5:         $Cnt \leftarrow 0$
6:         for $F$ from 0 to NUM_FILTERS do
7:             if TestBit($F, Bit$) = TRUE then
8:                 $Cnt \leftarrow Cnt + 1$
9:         if $Cnt <$ THRESHOLD then
10:            $Hotness \leftarrow$ COLD
11:    return $Hotness$
12: function WRITE_MBF($Lpn$)
13:    for $H$ from 0 to NUM_HASH_FUNC do
14:        $Bit \leftarrow$ Hash($H, Lpn$)
15:        $F \leftarrow$ CUR_FILTER
16:        loop
17:            if TestBit($F, Bit$) = FALSE then
18:                SetBit($F, Bit$)
19:                Break
20:            $F \leftarrow (F + 1)$ % NUM_FILTERS
21:            if $F =$ CUR_FILTER then
22:                Break
23:    if CurrentTime() % DECAY_PERIOD = 0 then
24:        CUR_FILTER $\leftarrow$ (CUR_FILTER + NUM_FILTERS - 1)
25:                        % NUM_FILTERS
26:        Clear CUR_FILTER
```

block is downgraded to the lower region.

The number of regions (NUM_REGIONS) in the DAC policy is normally set to four or more, but that works best varies from workload to workload. In addition, as a separate update block is allocated to each region, the DAC policy may not perform well especially when the number of clean blocks is tight. Algorithm 4 shows the pseudo code of Hotness_DAC() and Write_DAC() functions. Note that GetRegion(*Lpn*) returns the current region number of *Lpn* and SetRegion(*Lpn*, *Region*) sets the region number of *Lpn* to *Region*.

Algorithm 4 Hotness_DAC() and Write_DAC()

```
1: function HOTNESS_DAC($Lpn$)
2:     $Region \leftarrow$ GetRegion($Lpn$)
3:     if $Region > 0$ then
4:         $Region \leftarrow Region - 1$
5:     return $Region$
6: function WRITE_DAC($Lpn$)
7:     $Region \leftarrow$ GetRegion($Lpn$)
8:     if $Region <$ NUM_REGIONS then
9:         $Region \leftarrow Region + 1$
10:    SetRegion($Lpn, Region$)
```

4. EVALUATION

4.1 Methodology

We evaluate each hot/cold data separation policy on a real SSD called the Jasmine OpenSSD platform [1]. The Jasmine OpenSSD platform consists of Indilinx's Barefoot SSD controller, 64MB Mobile SDRAM, and two flash mem-

Policy	Name	Value
LRU	HOT_LIST_SIZE(α)	512
	CAND_LIST_SIZE (β)	1532
MBF	BLOOM_FILTER_SIZE (γ)	4096
	NUM_FILTERS (δ)	4
	NUM_HASH_FUNC	2
	THRESHOLD	2
	DECAY_PERIOD	512
DAC	NUM_REGIONS (η)	4

Table 1: Parameter values of each policy.

ory modules attached to different flash channels. As each flash module supports four 8GB NAND packages, the total capacity is 64GB. We combine two flash memory dies together in each NAND package and enable 2-plane operation with channel-level interleaving [4]. Hence, eight physical pages (each 4KB) are read or programmed at once, effectively forming a 32KB *virtual page*. The Jasmine OpenSSD platform has been connected to a PC via the SATA2 interface, which has Intel Core i5-3570 3.40GHz CPU and 8GB RAM running Linux 3.5.0.

We develop six synthetic workloads named SKEW70, SKEW90, SKEW95, SKEW99, SKEWINC, and SKEWDEC. Each SKEWX workload uniformly issues X% of the total writes to the (100-X)% *hot area* of the logical disk area. The higher skew rate indicates that write requests are much more concentrated on the smaller area. SKEWINC and SKEWDEC workloads dynamically vary the amount of skewness from 70% to 99% and vice versa, respectively, to see whether each policy is adaptive to changing workloads.

In addition to synthetic workloads, we replay four traces of real workloads, FINANCIAL, WEB, GENERAL, and TPC-C. FINANCIAL is the trace collected from OLTP (On-Line Transaction Processing) applications running at a financial institution [3]. Other traces are collected by in-house tools. WEB is obtained while surfing the web during one day. GENERAL models a general desktop computing workload where a user runs office suite, downloads mp3 files, and plays music and movies during five days. Finally, TPC-C is gathered from a commercial DBMS while running the TPC-C benchmark [2] for three hours.

We use Write Amplification Factor (WAF) [15, 10] as a performance metric to evaluate each policy. WAF is calculated by dividing the actual amount data written to flash memory by the amount of data written by the host. Since FTL generates additional writes during GC, WAF becomes greater than one and a smaller WAF value represents better performance. As described in Section 3, each policy is influenced by the setting of various parameters. Table 1 summarizes the parameters of each policy which are set to the same values as in the original papers.

4.2 Synthetic Workloads

Figure 1 compares the WAF value of each policy in synthetic workloads. The Baseline policy treats all LPNs equally as cold. For comparison, we also evaluate the Oracle policy which statically determines those LPNs belonging to the (100-X)% hot area as hot. Compared to Baseline, we can observe that all the hot/cold separation policies are quite effective in improving the WAF values. Not surprisingly,

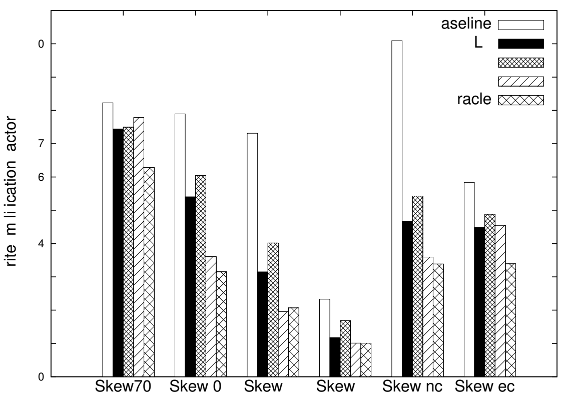

Figure 1: Write Amplification Factor of Synthetic Workloads

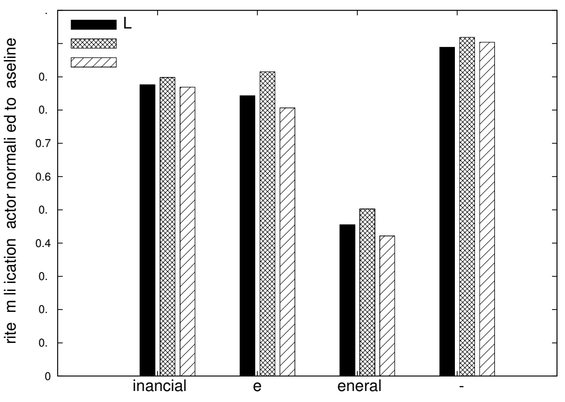

Figure 2: Write Amplification Factor of Real Workloads

Oracle shows the best performance in most cases. Overall, DAC closely catches up the performance of Oracle when the skew rate is greater than 70%.

We note that WAF tends to be decreased as the skew rate rises. This is because the higher skew rate invalidates a smaller number of hot data fast, reducing the number of valid pages in a victim block during GC. However, Baseline does not perform well until the skew rate reaches 99%. We can also see that Baseline fails to cope with workloads where the amount of hot data varies.

MBF shows the worst performance among the evaluated policies. Compared to Oracle, MBF has classified only 36% of hot data as hot in SKEW99 and less than 36% in other cases. This is due to the DECAY_PERIOD parameter value which seems to be too short. When DECAY_PERIOD is too short, most of data will be regarded as cold since the contents of BFs are frequently cleared. Similar to MBF, the performance of LRU depends on its parameter values. LRU is based on fixed-size lists and only the data included in the hot list is treated as hot. If the amount of hot data fits in the hot list, which is the case in SKEW99, LRU exhibits comparable performance to DAC. Otherwise, the performance of LRU degrades severely.

4.3 Real Workloads

Figure 2 depicts the WAF values of real workloads nor-

Financial					
Policy	Write Count	GC Count	Time elapsed (sec)		Ratio
			Policy	Total	
LRU	4840118	37736	229.22	7151.90	3.21%
MBF	4959035	38673	47.95	7298.61	0.66%
DAC	4799136	37387	4.61	7074.66	0.07%

TPC-C					
Policy	Write Count	GC Count	Time elapsed (sec)		Ratio
			Policy	Total	
LRU	4588624	35736	52.88	7534.94	0.70%
MBF	4728355	36870	46.60	7795.48	0.60%
DAC	4658552	36279	1.41	7857.75	0.02%

Table 2: Computation overheads in financial and tpc-c

Policy	Usage	Memory (bytes)	Total (bytes)
LRU	Hot list	α * 8	14360
	Candidate list	β * 8	
	Misc.	56	
MBF	Bloom filter	γ / 8 * δ	2052
	Misc.	4	
DAC	Region ID	υ * $\lceil \log_2 \eta \rceil$ / 8	2048

Table 3: Memory consumption

malized to the results of Baseline. Similar to synthetic workloads, hot/cold data separation policies are helpful in reducing the WAF values, and DAC performs reasonably well improving the WAF value of GENERAL by 58%. The only exception is the TPC-C workload; all policies were useless or even harmful. The reason is that TPC-C has low skewness and written LPNs are more uniformly distributed than other workloads.

4.4 Overheads

To evaluate the computation overhead of each policy, we measure the total elapsed time and the time for executing Algorithms 2, 3, and 4 for LRU, MBF, and DAC, respectively. Table 2 presents the results for FINANCIAL and TPC-C. The computation overhead of LRU is significant, spending the total 229 seconds while replaying the FINANCIAL trace. This is because LRU manipulates LRU-based lists on each write request and performs linear search on the hot list to see whether the given LPN is hot or not. MBF also requires considerable computation for calculating hash values and accessing BFs. On the contrary, DAC has the lowest computation overhead as it just needs to get or set a region number for the given LPN. As Table 2 shows, the ratio of the time taken for hot/cold data separation over the total elapsed time is at most 3.2% and less than 0.1% for DAC.

Table 3 compares the amount of memory required to implement each policy. The symbols α, β, γ, δ, and η represent the corresponding parameters in Table 1 and υ indicates the total number of blocks which is set to 8192 in our evaluation platform. In LRU, most of memory is consumed by two doubly-linked lists, where each entry contains 4-byte LPN and two 2-byte pointers. The memory consumption of MBF depends on the number of BFs and the size of each BF. DAC is most efficient in terms of space overhead as it only needs to save the region number for each block.

5. CONCLUSIONS

This paper aims at quantitatively evaluating three hot/cold data separation policies, LRU, MBF, and DAC, on a real SSD platform. We have devised a general FTL framework, where each policy can be easily plugged in. Our evaluations with six synthetic workloads and four real workloads indicate that separating hot data from cold data is quite effective in improving the performance and lifetime of SSDs. Among the evaluated policies, DAC performs best improving the WAF value by up to 58% in real workloads with a reasonable computation and memory overhead. However, there is still room for improvement especially when the workload has low skewness and/or the amount of hot data changes over time. We plan to pursue a more adaptive hot/cold data separation policy that can handle these situations.

6. ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MEST) (No. 2010-0026511). This work was also supported by the IT R&D program of MKE/KEIT (No. 10041244, SmartTV 2.0 Software Platform).

References

[1] Jasmine OpenSSD Platform. `http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform/`.

[2] Transaction Processing Performance Council. `http://www.tpc.org/tpcc/`.

[3] UMass Trace Repository. `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[4] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proc. Annual Technical Conference on Annual Technical Conference*, pages 57–70, 2008.

[5] L.-P. Chang and T.-W. Kuo. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proc. Real-Time and Embedded Technology and Applications Symposium*, pages 187–196, 2002.

[6] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software – Practice & Experience*, 29(3):267–290, March 1999.

[7] CompactFlash Association. `http://www.compactflash.org/`.

[8] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. Conference on Architectural support for programming languages and operating systems*, pages 229–240, February 2009.

[9] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage*, (1):22–40, February 2006.

[10] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proc. Conference on Systems and Storage*, 2009.

[11] Intel Co. Understanding the flash translation layer (FTL) specification. `http://developer.intel.com/`.

[12] D. Park. Hot data identification for flash-based storage systems using multiple bloom filters. In *Proc. Mass Storage Systems and Technologies*, pages 1–11, 2011.

[13] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proc. SOSP*, pages 1–15, October 1991.

[14] Samsung Electronics Co. NAND flash memory & smartmedia data book, 2005.

[15] Wikipedia. Write amplification. `http://en.wikipedia.org/wiki/Write_amplification`, 2010.