An Efficient SSD-based Hybrid Storage Architecture for Large-scale Search Engines

Ruixuan Li, Chengzhou Li
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, P.R.China
Email: rxli@hust.edu.cn, chengzhouli@smail.hust.edu.cn

Weijun Xiao
Department of Electrical and Computer Engineering
University of Minnesota
Twin Cities, USA
Email: wxiao@umn.edu

Hai Jin, Heng He, Xiwu Gu, Kunmei Wen
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, P.R.China
Email: hjin@hust.edu.cn, henghe@smail.hust.edu.cn,
guxiwu@hust.edu.cn, kmwen@hust.edu.cn

Zhiyong Xu
Department of Mathematics and Computer Science
Suffolk University
Boston, USA
Email: zxu@mcs.suffolk.edu

***Abstract*—Large-scale search engines use hard disk drives (HDD) to store the mass index data for their capacity, whose performances are limited by the relatively low I/O performance of HDD. Caching is an effective optimization, and many caching algorithms have been proposed to improve retrieval performance. Considering the high cost of memory and huge amounts of data, the limited capacity of cache in memory cannot resolve the above problem thoroughly. In this paper, we adopt a solid state disk (SSD) based storage architecture, which uses SSD as a secondary cache for memory. We analyze the I/O patterns of search engines and propose SSD-based data management policies based on the hybrid storage architecture, including data selection, data placement and data replacement. Our main goal is to improve the performance of search engines while reducing operation cost inside SSD. The experimental results demonstrate the proposed architecture improves the hit ratio by 13.31%, the performance by 41.05%, the average access time inside SSD by 43.83%, and reduces block erasure operations by 71.52%.**

***Index Terms*—search engine; solid state disk; hybrid storage architecture; caching**

I. Introduction

A web search engine is designed to search information on World Wide Web servers. Large search engines need to process hundreds of queries per second on collections of millions of documents. As a result, query processing is a major performance bottleneck and cost factor in the current search engines. A number of techniques have been employed to increase query throughput, including massively parallel processing, index compression, early termination, and caching.

With the development of computer technology in hardware, especially the rapid development of CPU technology, the low I/O performance of hard disk drive (HDD) becomes the major bottleneck in modern large-scale search engines. For the last two decades, researchers have made continuous efforts to address several open issues of HDD, such as long latencies of handling random accesses, excessively high power consumption, audible noise and uncertain reliability, etc. Considering that these issues are essentially rooted in the mechanical nature of HDD, they are inherently difficult to be solved by localized improvements of disks.

Fortunately, the emerging solid state disk (SSD) technology brings new and promising opportunities to those I/O-intensive applications. Unlike traditional rotating media, SSD is based on semiconductor chips, which provides many desired technical merits, such as low power consumption, compact size, shock resistance, and most importantly, ultra-high performance for random data access. Consequently SSD has been called as a "pivotal technology" to revolutionize storage systems. In fact, two leading on-line search engine service providers, namely google.com and baidu.com, announced their plans to migrate part of their data from HDD to SSD [1].

However, two potential issues may complicate the full adoption of SSD in large-scale search engines. First, current average cost per GB of SSD is 10 times more than that of HDD. In addition, the existing data in large-scale search engines are extraordinarily large, it is not suitable to store all the data on SSD. For example, Google makes use of a large number of cheap servers with HDD to provide high quality services [2]. Second, since SSD has a limit of block erasure count, the combination of a more stressful workload and fewer available erasure cycles reduces the useful lifetime of SSD, in some cases to less than one year [3].

In this paper, we aim to further improve the performance of large-scale search engines by using SSD without obviously increasing the cost of servers. Without any major change to the existing HDD-based storage systems used in search engines, we propose an SSD-based hybrid storage architecture with memory as the first-level cache and SSD as the second-level cache. In search engines, two types of cached data dominate, namely results and inverted lists. There are some differences between caching results and inverted lists. First, the result entries are small and similar in size, the inverted list entries are usually large and variable in size. In addition, only part of the inverted lists are required during computing.

Second, the results are prominently relevant to the queries and time-sensitive, while the inverted lists are relatively stable. Considering these characteristics, we propose different policies to manage the two types of cached data respectively, which are specifically designed for search engines. To the best of our knowledge, although this hybrid SSD-based two-level cache architecture has been evaluated in database systems [4], we are the first to comprehensively analyze this problem and propose an efficient solution for the application of search engines.

We have made three main contributions in this work. First, we propose a data selection policy, which carefully places the data to be cached in memory or SSD. Second, we propose an improved log-based method to organize the cached data on SSD so as to ensure the performance of write and read operations, which is a data placement issue. Third, we propose different overwriting policies for result cache and inverted list cache in SSD so as to avoid expensive random writes and reduce block erasure operations, which is a data replacement issue.

The rest of the paper is organized as follows. Section II describes the background and discusses the related work. Section III analyzes the characteristics of I/O trace in search engines. Section IV characterizes the problem that we are discussing in this paper. Section V describes the two-level cache architecture. Section VI introduces our proposed cache algorithms. Section VII presents the results of performance evaluation, and Section VIII concludes this study and discusses future work.

II. Background and Related Work

In this section, we focus on previous work which is closely relevant to our work, including related work on SSD, SSD-based hybrid storage, SSD-based buffer management, and caching in search engines.

A. Solid State Disk

Over the past decades, the use of flash memory has evolved from specialized applications in hand held devices to primary system storage. Flash memory can be written or read a single page at a time, but it has to be erased in granularity of block, which is much larger than page [5].

Inside an SSD, there is a kind of special software called flash translation layer (FTL). Researchers have done a lot of work on FTLs. In 1998, Intel proposes a page-mapped FTL for the first time [6], however, the mapping table takes up large space of SRAM in SSD, which makes it not suitable for SSD of large capacity. In order to overcome this problem, a block-mapped FTL is proposed in [7]. As a trade-off, the block-mapped FTL has generally a lower read and garbage collection performance. To address the above shortcomings, researchers have come up with a log-based FTL scheme, which is a hybrid between page-level and block-level schemes [8][9]. In 2009, a page-level mapping scheme called DFTL is proposed to reduce memory requirement and lookup overhead by loading partial of mapping table into SRAM [10]. All the FTLs above are designed to resolve the issue of "erase-before-write" and limited life-span inside SSD. Although the FTLs are completely transparent to users, different FTLs may suffer a big difference in the same application. In this paper, we take the ideal page-based FTL [6] as the base line.

B. SSD-based Hybrid Storage

Considering the special I/O performance of SSD, researches on SSD-based hybrid storage architecture has attracted much focus from both academic and industrial fields. Given its excellent random read performance, SSD can work well as a read cache in front of a larger HDD [11][12]. G. Soundararajan et al. propose Griffin, a hybrid storage device that uses HDD as a write cache for a Solid State Device (SSD) [3]. Although SSD is also used as a read cache in our proposed hybrid storage architecture, we focus the special application (large-scale search engines) and optimize the performance of SSD-based read cache according to its characteristics.

C. SSD-based Buffer Management

The traditional virtual memory system has been designed for decades assuming a magnetic disk as the secondary storage. Park et al. [13] propose a new replacement policy called CFLRU to reduce writes to SSD by keeping dirty pages in memory as long as possible. Jung et al. [14] propose an LRU algorithm called LRU-WSR that enhances LRU by reordering writes of not-cold dirty pages from the buffer cache to flash storage. Kim et al. [15] propose a new write buffer management scheme called BPLRU, which significantly improves the random write performance of flash storage. CFLRU and LRU-WSR are oriented to the application layer, while BPLRU is an internal buffer management algorithm inside SSD. The above algorithms are designed for general purpose applications, while we focus on large-scale search engines.

D. Caching Techniques in Search Engines

In search engine, caching is an effective optimization. The caching techniques in search engines can be classified as follows.

Result caching: Result caching filters out repetitions in the query stream by caching the complete results of previous queries for a limited time window [16][17].

List caching: At the lower level inside each index server, list caching is used on a lower level in each participating machine to keep the inverted lists of frequently used search terms in main memory [18][19].

Two-level caching: Saraiva et al. [18] evaluate a two-level caching architecture using result and list caching on the search engine TodoBR.

Three-level caching: Long et al. [19] propose and evaluate a three-level caching scheme that adds an intermediate level of caching.

In [20], Huang et al. point out that it is possible to substantially improve the performance of web index server by using flash-aware storage management approaches. Pritchett

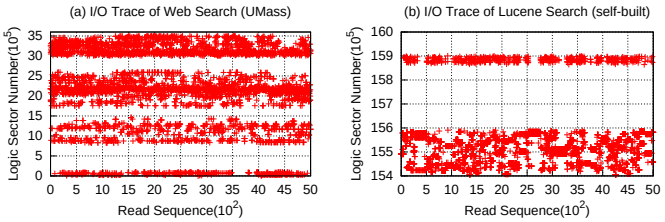

Fig. 1. The I/O trace of search engines

et al. [21] propose a new and cost-effective architecture, namely "SieveStore". However, "SieveStore" does not take the characteristics of caching in large-scale search engines into account.

As far as we know, few studies focus on the optimization in search engines by using SSD. In [20], Huang et al. consider the optimal policy that allocates the DRAM portion of inverted index into flash memory as much as possible. But they do not discuss how to place the data on SSD for higher I/O performance and how to replace the data on SSD when the available free space of SSD is exhausted. In addition, they do not take the cost effectiveness into consideration under the circumstances of large-scale search engines. In this paper, we try to discuss the problem of caching with SSD, design several policies and give a simple implementation so as to improve the performance of large-scale search engines.

III. I/O PATTERNS OF SEARCH ENGINES

In this paper, we have collected two kinds of I/O traces in search engines, one is downloaded from "UMass Trace Repository" [22], and the other is collected from a simulative search engine built on Lucene [23] by ourselves. In self-built search engine, we collect the disk I/O trace on Windows Server 2003 with DiskMon during the process of retrieval. Figure 1 shows the I/O patterns of the search engines, where the x-axis denotes the read sequence, and the y-axis denotes the logic sector number.

Four obvious characteristics in the I/O patterns of search engines can be figured out: read-dominant, locality, random reads and skipped reads, which are described in detail as follows.

- **Read-dominant**. According to the analysis using the web search trace of UMass, read operations take up more than 99%, which indicates that read performance takes a key role in search engines.
- **Locality**. Although the data scale is tremendously large, only part of the data is frequently accessed. Locality is quite obvious during the process of retrieval.
- **Random reads**. As large amount of irregular queries are processed at the same time, random reads are inevitable.
- **Skipped reads**. In practice, search engines adopt many technologies to optimize the I/O performance, such as skipped reads. Take Lucene as an example, there are several "skip list" in the index of Lucene. During the process of retrieval, although the "docId" lists are stored sequentially in the inverted lists, they are more likely to be read in skip order rather than in sequential order. The reasons are as follows: first, different documents have different term frequency ("tf"), only the documents that have higher term frequency will be accessed during computing; second, considering early termination during the process of retrieval, only part of the inverted list are required, therefore skipped reads may take place frequently

Unlike the other read-dominant data centers, the search engine applications have their own particular characteristics in terms of data structures and access patterns. Furthermore, during the process of retrieval, the access frequency of terms follows Zipf-like distribution [18]. In addition, although sometimes the inverted list of a term is fairly large, only small part of the inverted list is required during computing.

IV. PROBLEM DEFINITION

A. Caching Scheme

Considering the shared data by memory and SSD, we divide the two-level cache scheme into three caching schemes: inclusive scheme, exclusive scheme and hybrid scheme. With a page as the smallest cache unit, the three schemes can be described as follows.

- **Inclusive Scheme**. Whenever a page is in memory, it is also cached on SSD. That is to say, if the system caches a page in memory, it should also write the page to SSD.
- **Exclusive Scheme**. No page is stored on both memory and SSD at the same time. A page brought from SSD to memory is removed from SSD, and vice versa.
- **Hybrid Scheme**. A page in memory may or may not be cached on SSD, depending on criteria either set by the user or decided based on the current workload.

In this paper, we adopt the hybrid scheme. The main reasons are as follows. If we use the inclusive cache scheme, SSD and memory share most of the cached data, which can not bring the expected advantages of SSD into full play. If we take the exclusive scheme, the data should be removed when they are read from SSD, which will result in a number of block erasure operations inside SSD and shorten the life-span of SSD. In the hybrid scheme, all the hot data will be cached in memory first. Once the cache in memory is full, some least recently used data will be evicted and then written into SSD according to our proposed eviction policies. When the available capacity of SSD is exhausted, the fresh data evicted from memory will overwrite the cold data in SSD. Note that if the data cached in SSD are hit, they will be read from SSD to memory without deleting.

B. Problem Consideration

Table I presents 9 different kinds of situations during the process of retrieval. As shown in Table I, note that "R" denotes the results, "I" denotes the inverted lists. "memory", "SSD" and "HDD" denotes where the data are read from respectively. "Probability" denotes the probability that the corresponding situation takes place. "Time Cost" denotes the average time cost of reading data from corresponding storage device.

TABLE I
RETRIEVAL UNDER DIFFERENT SITUATIONS

Situation	memory	SSD	HDD	Probability	Time Cost
S_1	R			P_1	T_1
S_2	I			P_2	T_2
S_3		R		P_3	T_3
S_4	I	I		P_4	T_4
S_5		I		P_5	T_5
S_6	I	I	I	P_6	T_6
S_7	I		I	P_7	T_7
S_8		I	I	P_8	T_8
S_9			I	P_9	T_9

In order to minimize the average response time, the following conclusions are reached by analyzing the different situations in Table I.

(1) As our goal is to reduce read operations on HDD and write operations to SSD, it is reasonable to place read-intensive data in memory or SSD and write-intensive data on HDD. That is, the probability of "S1", "S2", "S3", "S4", and "S5" should be increased.
(2) Considering that the inverted lists are variable in size while the capacity of cache is limited, it is important to balance the access frequency and the size of inverted list. Therefore, an advanced algorithm is needed to identify which part of the inverted list is worth caching.
(3) Considering the special characteristics of SSD (e.g. asymmetrical write and read performance, limited block erasure count) and limited capacity, it is a challenge to determine whether the data evicted from memory should be flushed to SSD.

Considering that the typical large-scale search engines are usually read-dominant applications, for simplicity, we limit our discussion in the static scenario in this paper. If the dynamic situation is considered, we can also have a similar solution. Suppose that each cached data has a "TTL" (Time-to-Live), when the cached data expire, the search engines will read the latest data from HDD for computing. The dynamic situation is one of the research contents in our future work.

V. ARCHITECTURE

Our proposed hybrid SSD-based storage architecture for large-scale search engines can be illustrated as Figure 2, which is similar to the multi-level cache (L1, L2, L3 etc.) in CPU. In this architecture, the first-level cache is memory, which is adopted to store the most frequently used data, and the second-level cache is SSD, which is used as a complement for memory and mainly stores the data evicted from memory to SSD when the cache in memory is full.

In Figure 2, all the queries from users will be pretreated by query processor first, and then they will be forwarded to cache manager. The cache manager is the most important part in this architecture, whose functions include selection management (SM), query management (QM), and replacement management (RM). All the data evicted from memory should be transferred to write buffer for assembling before they are flushed to SSD.

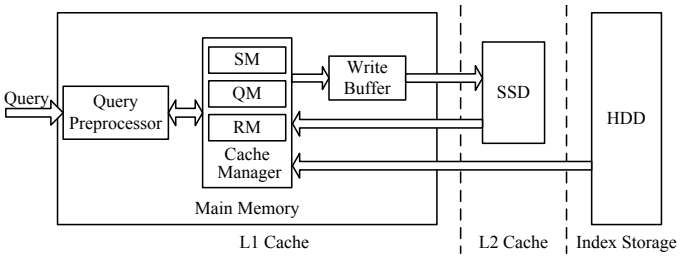

Fig. 2. The SSD-based cache architecture for search engines

The functions of "SM", "QM", and "RM" can be described in detail as follows.

Selection Management (SM). Considering the limited capacity and special I/O characteristics of SSD, not all the data evicted from memory will be flushed to SSD. It is important to select the data to be cached in memory or SSD.

Query Management (QM). When a query arrives, the cache manager first checks whether the required data have been cached in memory or SSD. If neither, the cache manager has to read data from HDD for retrieval computing and then cache the used data in memory if necessary.

Replacement Management (RM). When the available cache space in memory or SSD is exhausted, data replacement is inevitable. The functions of replacement management comprise two aspects: first, when the cache in memory is full, some data should be evicted from memory to SSD and the fresh data will replace the victim data in memory; second, when the available free space in SSD is exhausted, the cache manager has to select the victim data on SSD for replacement.

We will present the data selection, data placement and data replacement policies in Section VI.

VI. DATA MANAGEMENT POLICIES

In this paper, we take a two-level cache scheme in search engines, which combines result and inverted list caches together. In realistic search engines, the size of each cached result may be approximate or identical, and we take it as fixed-length cache entry. However, the cached inverted lists are usually variable in size, thus we call it as variable-length cache entry.

In our research, we assume the parameters of SSD as follows: the size of a page is 2KB, and one block contains 64 pages, namely the size of a block is 128KB. Each read, write and erasure operation may take 20μs, 250μs and 1.5ms respectively.

In cached query results, each result entry only caches the Top-K documents, and in this paper we assume K equals 50. Suppose that the size of each document (i.e. URL, snippet, date, etc.) in a result entry approaches 400B, the size of each result entry (50 documents) is nearly 20KB. When the result cache buffer in memory is full, the evicted result entries will be transferred to write buffer, waiting to be flushed to SSD.

The size of an inverted list is a function of both the term popularity in the collection and the number of documents being indexed. For large collections, these inverted lists may be very large, which makes it impossible to cache most of the valuable inverted lists in memory. To address this problem,

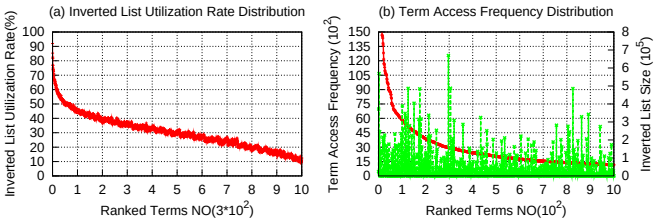

Fig. 3. The inverted list utilization rate distribution and the term access frequency distribution (5 million documents indexed from enwiki data set, and the query log is from AOL)

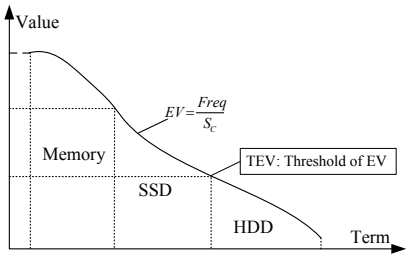

Fig. 4. The relationship between efficiency values and terms

we turn to an important characteristic of the filtered vector model processing technique, which has been discussed in [18]. In this technique, the inverted lists are sorted according to the frequency of the term occurrence in each document, and the query processing exploits the frequency variance by using the documents in which the term is most frequent. As a consequence, the lists are not fully traversed or are not traversed at all, depending on the relevance of the term in the query and the document collection. Since lists are almost always partially processed, we set out to cache part of the inverted lists. The frequency-sorted inverted lists can be partitioned in different ways with different tradeoffs. In our research, we divide the inverted lists based on block size (i.e. 128KB, 512KB, or 1024KB).

A. Data Selection

Compared with inverted list entries, result entries are quite small and similar in size, so we can take the common policy to deal with result selection. In this part, we mainly focus on the inverted list selection policy.

Figure 3 represents the inverted list utilization rate distribution and the term access frequency distribution. It is obvious that only part of the inverted lists are used during query processing (Figure 3(a)) and only a small part of inverted lists are frequently accessed (Figure 3(b)). Considering these characteristics, it is worthy studying the data selection policy for the inverted list cache carefully.

When the inverted list cache in memory is full, the evicted inverted lists will also be transferred to a write buffer, waiting to be flushed into SSD. Formula 1 presents how to determine the size of an inverted list which will be flushed to SSD. In Formula 1, S_C denotes the size of the inverted list to bes cached in SSD, S_I denotes the size of the total used inverted list in memory, P_U denotes the utilization rate of the inverted list and S_B denotes the block size in SSD (in this paper, S_B equals 128KB). For example, if S_I is 1000KB, P_U is 50%, the S_C is 4 (512KB, namely 4 blocks). In this way, all the cached data are of integral blocks ($128 * N$ KB). Note that the value of P_U can be obtained by analyzing the query log or computing during the process of retrieval. In this paper, we assume that P_U is already known by analyzing the query log.

$$S_c = \lceil \frac{S_I * P_U}{S_B} \rceil \tag{1}$$

Considering the inverted lists are variable in size, and the access frequency of terms obeys Zipf-like distribution, we take "size" and "frequency" into consideration in inverted list cache replacement. Formula 2 defines the efficiency value of the cached inverted list, which is directly proportional to "$Freq$" (the access frequency of an inverted list) and inversely proportional to "S_C" (the size of a cached inverted list).

$$EV = \frac{Freq}{S_C} \tag{2}$$

Figure 4 shows the relationship between the efficiency value of inverted lists and terms. In our research, we assume that the cache manager will keep the most efficient data in memory, and second most efficient data in SSD. If the efficiency value of an inverted list is less than a specified threshold (the threshold "TEV" can be determined by analyzing the query log), it will be discarded directly, rather than flushed to SSD. In inverted list cache replacement policy, we choose the victim entry by using the efficiency value of inverted list, rather than the access frequency of inverted list merely.

B. Data Placement

Figure 5 shows how data are flushed to SSD from the write buffer. We assign each result cache block a logic block number named RB_n. The primary benefit is two-fold: first, the larger read and write operations can improve the performance of SSD [5]; second, this kind of data placement policy can avoid data fragment during the process of data replacement.

The cache manager should maintain two types of mappings, one is for memory, and the other is for SSD. Figure 6 shows the memory cache mappings. In Figure 6(a),"key" denotes the cached query, and "value" comprises the top 50 documents (one result entry) and access frequency of the corresponding query. In Figure 6(b), "term" denotes the cached term, "I" denotes the inverted list, "freq" denotes the access frequency, "size" denotes the size of the corresponding inverted list, "P_U" denotes the utilization rate of the inverted list. Figure 7 shows the SSD cache mapping. In Figure 7(a), "key" denotes the cached query, "value" comprises the pointer of an result entry in the cache file of SSD, access frequency and result block number. In Figure 7(b), "key" denotes the result block number, "value" comprises the pointer of an RB in the cache file of SSD and result flag (a bitmap, one bit for each result entry in an RB, "1" denotes valid and "0" denotes invalid "10110000" denotes the first, third and fourth result entries are valid). In Figure 7(c), "key" denotes the cached term, "value" comprises the pointer of an inverted list in the cache file of SSD, access frequency and size. It is obvious that our proposed

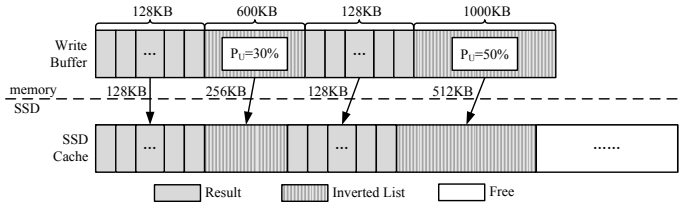

Fig. 5. The process of how data are flushed to SSD

key	value
$query_1$	$<R_1,freq_1>$
$query_2$	$<R_2,freq_2>$
...	...

(a) Result Mapping in Memory

key	value
$term_1$	$<I_1,freq_1,size_1,P_{U1}>$
$term_2$	$<I_2,freq_2,size_2,P_{U2}>$
...	...

(b) Inverted List Mapping in Memory

Fig. 6. The memory cache mappings

key	value
$query_1$	$<ptr_1,freq_1,RB_1>$
$query_2$	$<ptr_2,freq_2,RB_2>$
...	...

(a) Result Mapping in SSD

key	value
RB_1	$<ptr_1,flag_1>$
RB_2	$<ptr_2,flag_2>$
...	...

(b) Result Block Mapping

key	value
$term_1$	$<ptr_1,freq_1,size_1>$
$term_2$	$<ptr_2,freq_2,size_2>$
...	...

(c) Inverted List Mapping in SSD

Fig. 7. The SSD cache mappings

data placement policy can save memory space substantially (compared with memory merely).

C. Data Replacement

In order to improve the performance of write operations and reduce block erasure operations in SSD, we adopt an improved log-based file management policy. In this paper, we divide the space of SSD into three states logically (Figure 8), respectively "normal state", "replaceable state" and "free state".

The "normal state" denotes that the data block is valid and read-only. The "replaceable state" denotes that the data block are replaceable (Figure 8(a)), as they have been read back from SSD to memory or marked as invalid. If the available space of SSD is abundant, the replaceable state blocks are just read-only without overwriting (Figure 8(b)). But if the available space of SSD is exhausted, the replaceable state data blocks are more likely to be replaced first (Figure 8(c)). The "free state" denotes that the data block is free and available for writing.

Figure 9 shows the state transition among the above three states. Once a free block is written, it will change from "free" to "normal". When a normal block is read from SSD to memory or marked as invalid, it will change from "normal" to "replaceable". However, when a replaceable block is overwritten by fresh data, it will change from "replaceable" to "normal" again. In practice, the data cached in SSD may become cold as time goes on, thus it's better to delete the cold data at a proper time for two reasons: one is to make full use of the limited capacity of SSD, the other is to improve the write performance of SSD. Note that some types of SSD support "Trim" [24] operation during deleting, which can improve the write performance in the large-scale I/O-intensive applications.

Considering the different characteristics between results and inverted lists, we propose two kinds of replacement policies in

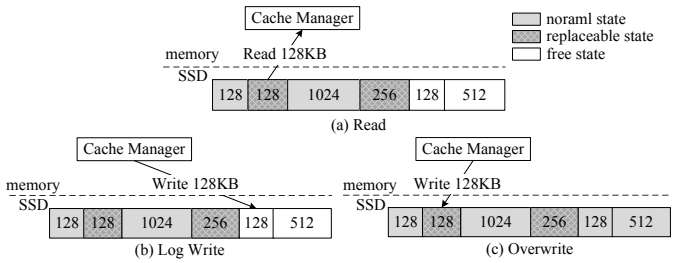

Fig. 8. The log-based cache files management

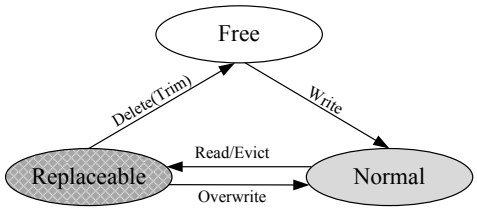

Fig. 9. The free, normal and replaceable state change

our research, one is for results, the other is for inverted lists. Although the two algorithms behave differently, they are both based on "cost", which considers the I/O performance and block erasure inside SSD. We call the proposed algorithms as CBLRU (Cost-based LRU).

1) Result Cache Replacement Policy: To distinguish the two-level cache in memory and SSD, we call the result cache in memory and SSD as "L1 RC" and "L2 RC" respectively.

When "L1 RC" is full, the cache manager will choose the victim result entries according to the LRU algorithm. If the available free space of SSD is exhausted, the cache manager has to choose some result entries in SSD to replace with the fresh result entries evicted from memory. On the condition of the traditional LRU algorithm, the small write operations are random. Small writes are the worst case in SSD, because it may bring low write performance, large numbers of block erasure, and it also impacts on the performance of read operation inside SSD. Therefore, our main idea is to change small random writes to large sequential writes during the process of result replacement. In result cache, the smallest unit of write and overwrite operation is RB (128KB, a block size), rather than a result entry (about 20KB) merely.

Figure 10 shows the process of result eviction and replacement in memory, which automatically converts the random writes to sequential writes (similar to [25]). In Figure 10, there are three types of result entries: "normal result", "victim result", and "fresh result". "normal result" denotes the normal result entries stored on SSD, which are read-only, "victim result" denotes the evicted result entries, which are marked as invalid during the eviction, and "fresh result" denotes the result entries evicted from memory and ready to flush to SSD. In Figure 10(a), a sequence of small random writes may cover several RBs. However, in Figure 10(b), we collect a sequence of result entries and assemble them as a logic result block so that several small random writes can be assembled into a large sequential write.

Before result entries are flushed to SSD, all the evicted result

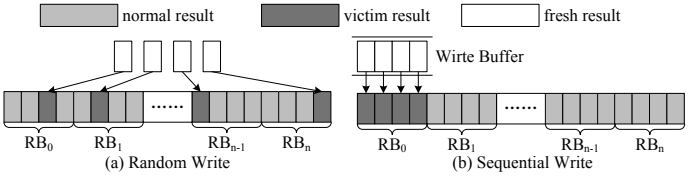

Fig. 10. The result replacement policy in memory

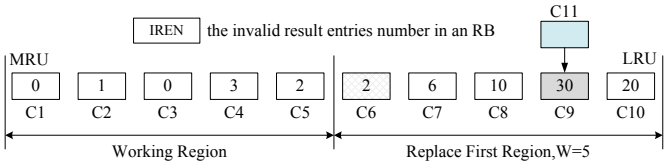

Fig. 11. The result replacement policy in SSD

entries should be transferred to write buffer first. The cache manager will check where the result entries in write buffer are in replaceable state, if so, the cache manager will remove them directly so as to reduce unnecessary write to SSD; if not, the cache manager will keep them in write buffer until they are flushed to SSD. When the number of result entries in write buffer reaches a specific number, the cache manager will flush the assembled RB to SSD. Once a RB in SSD is overwritten by a fresh RB, the cache manager will modify the "flag" value of the evicted result entry in RB mapping and set its corresponding bit into invalid.

Figure 11 shows how to choose the victim RB for replacement in SSD. In Figure 11, "IREN" denotes the invalid result entries number in an RB, which includes replaceable result entries and victim result entries. We divide the LRU list into two parts: one is "Working Region", and the other is "Replace First Region". "W" denotes the window size of the replace first region. The "Working Region" mainly maintains the most recently used data, while the "Replace First Region" keeps the least recently used data. Our replacement policy is intentionally designed to replace the RB that contains the largest "IREN". In this paper, we only implement a simple algorithm to choose the victim RB. It is worth being studied and optimized in the future work.

2) Inverted List Cache Replacement Policy: Figure 12 presents an example of the inverted list cache eviction and replacement policy in memory. The cache manager maintains the LRU list according to the value of the inverted list. In Figure 12, the cached entry "C9" will be evicted first and replaced by "C11" when the inverted list cache is full in memory. Note that the size of "C9" should be larger than "C11".

Figure 13 presents an example of the inverted list replacement policy in SSD. In Figure 13, we divide the LRU list into two parts: "Working Region" and "Replace First Region". Suppose that the oncoming write sequence happens as described in Figure 13. When the first write (i.e. "1" in Figure 13) arrives, the cache manager will first check whether there are some inverted list entries with "replaceable" state in "Replace First Region", if any it will replace the "replaceable" inverted list entries first, thus the first write "1" will overwrite "C6" firstly.

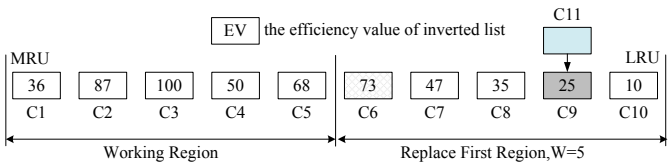

Fig. 12. CBLRU for the inverted list cache in memory

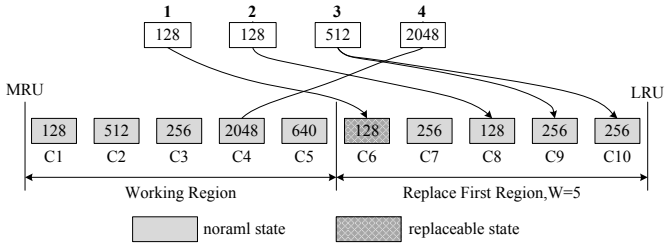

Fig. 13. CBLRU for the inverted list cache in SSD

The second write "2" will do the same check operation as the first one, but there are no available "replaceable" inverted list entries in SSD, so it will choose the inverted list entry which has the same size ("C8") as "2" to overwrite in the "Replace First Region". When the size of data to be written is larger than any data blocks in "Replace First Region", the cache manager will choose several inverted list entries to assemble a bigger data block, for example the third write "3" is inclined to overwrite "C9" and "C10". However, when the size of data to be written is too large and there is no suitable choice in "Replace First Region", which may be the worst case, the cache manager will look up in a wider region, namely in all the LRU list. According to this policy, the fourth write "4" will finally overwrite "C4" in Figure 13. However, the fourth situation has little chance of taking place.

In order to reduce write operation to SSD furthermore, we optimize the proposed cache algorithm CBLRU and propose another optimized algorithm called Cost-based Static LRU (CBSLRU). The CBSLRU algorithm divides the cache capacity into two parts: one is static, the other is dynamic. The dynamic cache is the same as CBLRU. The static cache mainly stores the most efficient data without any change, namely no eviction and no replacement. In realistic search engines, we can get the most efficient result entries and inverted list entries by analyzing the query log and then store the most valuable data on SSD first. In the next section, we will compare the proposed CBLRU with CBSLRU.

VII. PERFORMANCE EVALUATION

Our evaluation has four objectives. First, to verify that our proposed algorithms can improve the hit ratios. Second, to verify that our proposed algorithms do improve the retrieval performance of search engines. Third, to verify that our cache policy can reduce the cost of search engine servers effectively. Fourth, to verify that our proposed policies can reduce the block erasure operations inside SSD and improve the average access time considerably.

Table II summarizes the hardware and software environment settings. Our simulative search engine is on the basis of Lucene 3.0.0.

TABLE II
HARDWARE AND SOFTWARE ENVIRONMENT SETTINGS

Test-platform Environment	
IR Tool	Lucene 3.0.0
Data Set	enwiki-20090805-pages-articles.xml
Query Log	AOL-user-ct-collection
I/O Trace Analyzer	DiskMon 2.0.1
SSD Simulator	FlashSim/DiskSim 3.0 (PSU)
SSD	Intel SSD 320 Series 40GB
HDD	WDCWD3200AAJS 180GB
OS	Windows Server 2003/Ubuntu 10.04
CPU/RAM	Inter(R) Pentium(R) Dual CPU E2180/2GB

TABLE III
SIMULATION ENVIRONMENT SETTINGS

Simulated SSD	
FTL	page-mapping
Page Size	2KB
Block Size	128KB
Page Read	32.725 μs
Page Write	101.475 μs
Block Erase	1.5 ms

Table III summarizes the simulation environment setting. We use DiskMon to collect the hard disk I/O trace during the process of retrieval test and then we adopt FlashSim (PSU) [26] to do the SSD simulation. The key parameters of SSD relevant to our experiments are illustrated in Table III.

A. Hit Ratio Evaluation

The hit ratio is a critical evaluation in cache algorithm. In Figure 14, the number of total documents is 5,000,000, and the cache size ranges from about 20MB (10^6 units) to 200MB (10^7 units). Figure 14(a) presents the hit ratio comparison between result cache, inverted list cache, and the two-level cache which includes result cache and inverted list cache. Note that "RC" denotes result cache, "IC" denotes inverted list cache and "RIC" denotes result and inverted list cache. It can be seen from Figure 14(a) that the hit ratio will increase with the increase of cache capacity at a certain range. However, if we increase the cache capacity continuously, the hit ratio will have no obvious improvement furthermore.

Under the same condition, since a result entry is much smaller than an inverted list, the result cache will gain higher hit ratio than the inverted list cache. When the available capacity of cache reaches a certain extent in "RC", the hit ratio will not increase any more. However, things are quite different in "IC", because inverted lists are more significant in locality than results, the hit ratio improvement will be fairly obvious with the increase of cache capacity. Therefore, in realistic search engines, it is important to keep the capacity of "RC" within bounds. Taking into account the different characteristics between "RC" and "IC", we choose to allocate larger cache capacity for "IC". In experiments, we simply assume that "RC" takes up 20% of the cache capacity, while "IC" takes up 80%. The experimental results in Figure 14(a) further show that the "RIC" will gain higher hit ratio.

Figure 14(b) illustrates the hit ratio comparison between

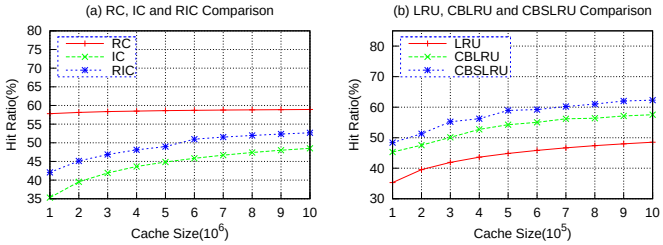

Fig. 14. The hit ratio comparison

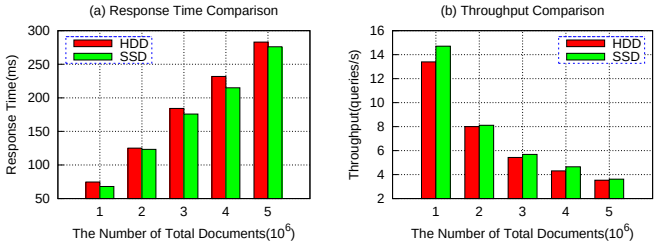

Fig. 15. The search test without cache

LRU, CBLRU and CBSLRU. It can be seen from Figure 14(b) that our proposed CBLRU and CBSLRU gain higher hit ratios than the traditional LRU. In this experiment, our proposed CBLRU and CBSLRU respectively improve the hit ratio by 9.05% and 13.31% averagely compared with LRU. The reason is that we consider the efficiency value of inverted list rather than the access frequency merely. In addition, only part of inverted lists are cached in CBLRU and CBSLRU, the limited cache can hold much more valid data.

B. Performance Evaluation

Figure 15 shows the results of retrieval test without any cache. With the increase number of documents, there is a sharp increase in the average response time and decrease in throughput. Although the performance of random read in SSD is much higher than that in HDD, the performance improvement is not obvious as expected with limited data in experiments. We believe that if the scale of data becomes much larger, the difference may be more noticeable, which is because that more seek time is needed in HDD.

Figure 16 presents the performance comparison with 1L cache and 2L cache. Note that "1LC" denotes one-level cache which only includes memory, and "2LC" denotes two-level cache which includes memory and SSD. "R" denotes result cache, "RI" denotes the result and inverted list cache. "HDD" denotes the index files are stored on HDD, while "SSD" denotes the index files are stored on SSD. In this experiment, we assume that the size of the result cache in SSD is as 10 times large as the size of the result cache in memory, and the size of the inverted list cache in SSD is as 100 times large as the size of inverted list cache in memory. As shown in Figure 16(a), there is a little performance improvement by replacing HDD with SSD to store the index files, but it is not obvious. Figure 16(b) proves that our proposed two-level cache design achieves better performance, especially in caching results and inverted lists.

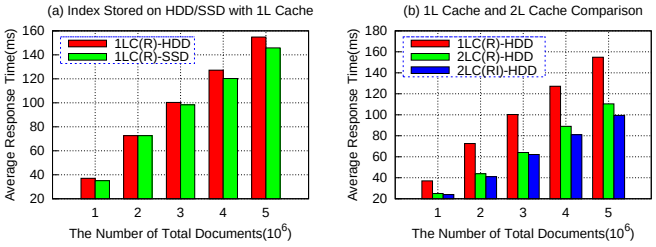

Fig. 16. Performance comparison with 1L cache and 2L cache

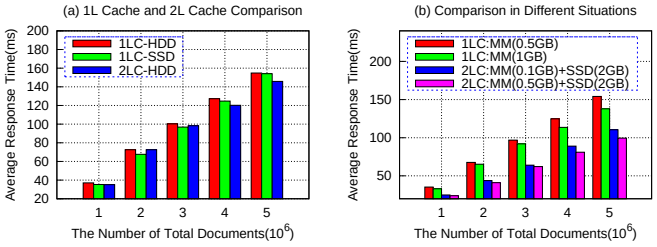

Fig. 18. The cost performance evaluation

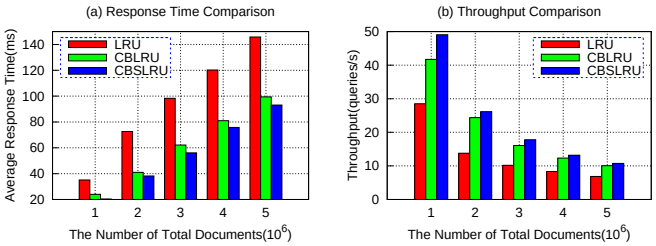

Fig. 17. Performance comparison between LRU, CBLRU and CBSLRU

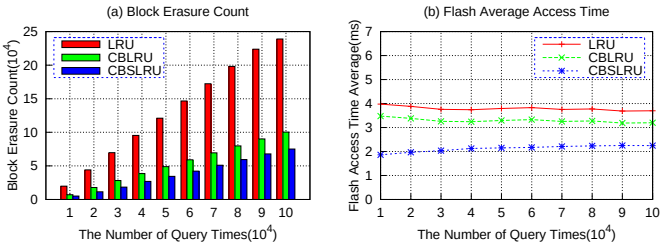

Fig. 19. The simulated performance inside SSD

Figure 17 shows that the two-level cache architecture with our proposed CBLRU and CBSLRU approaches can obviously improve the performance. In comparison with LRU, the average response time are reduced by 35.27% and 41.05% and the throughputs are increased by 55.29% and 70.47% in CBLRU and CBSLRU respectively. However, with the increase of capacity, the overhead of cache management may increase, which can affect the retrieval performance. Therefore, we strongly recommend that it is necessary to optimize the cache algorithm and choose appropriate capacity size for SSD cache.

C. Cost Performance Evaluation

In this paper, one of our main goals is to reduce the cost of servers without influencing the retrieval performance. Figure 18 presents the average response time under different conditions. In the figure, "1LC" denotes one-level cache, which uses memory merely, "2LC" denotes two-level cache, which uses memory and SSD as the cache (we use CBSLRU algorithm here). "HDD" denotes the index files are stored on HDD, and "SSD" denotes the index files are stored on SSD. "MM" denotes memory. In this paper, we assume that the capacity of "HDD" and "SSD" is large enough to hold all the index files.

Figure 18(a) indicates that our proposed SSD-based hybrid storage architecture demonstrates the best performance comparing to the one-level cache architecture with index files stored on HDD or SSD. Figure 18(b) represents the average response time between different capacities of memory and SSD. The experimental results show that the two-level cache architecture is superior to the one-level cache architecture obviously. In the two-level cache architecture, we can reduce the capacity of memory and enlarge the capacity of SSD without performance degradation. As the cost per GB of memory (14.5$) is much higher than that of SSD (1.9$), smaller memory means less cost. Compare to the strategies of increasing the capacity of memory or totally replacing HDD with SSD, our proposed storage architecture will cut down the cost largely under the current hardware condition of large-scale search engines.

D. Simulation Validation

In this experiment, we set the experimental environment as follows: the number of total indexed documents is 5 million, and the query count ranges from 10000 to 100000. We conduct experiments with three different cache algorithms, namely LRU, CBLRU and CBSLRU. During the process of retrieval, we use DiskMon to collect the I/O access pattern in SSD.

Inside SSD, the block erasure is a most expensive operation. Researchers have proved that the background operations affect foreground jobs evidently, especially on read [5]. In addition, SSD have limited block erasure count, thus the lifetime of SSD can be measured indirectly with the erasure count in SSD. In addition, the block erasure count can also reflect the I/O performance of SSD to some extent.

Figure 19(a) shows the simulated block erasure count comparison between LRU, CBLRU and CBSLRU. It can be seen from Figure 19(a) that our proposed CBLRU and CBSLRU can reduce the block erasure count. We believe the reason is that the LRU algorithm brings a large number of small random write and partition fragmentation, while our proposed CBLRU and CBSLRU are inclined to change the small random writes to a large sequential write, which can effectively reduce the block erasure count, and avoid partition fragmentation. In addition, we set a threshold for writing in the experiments. Only the result or inverted list entries exceeding a given threshold can be flushed to SSD. Otherwise, it will be discarded directly, which can reduce unnecessary writes to SSD. Furthermore, due to the static cache, the CBSLRU algorithm can also reduce write operations to SSD. Therefore, the block erasure count in CBSLRU is less than CBLRU. In this experiment, the block erasure count is reduced by 59.92% and 71.52% in CBLRU and CBSLRU compared to LRU.

Figure 19(b) presents the average access time in SSD. It can be seen from Figure 19(b) that, as the process of retrieval

goes on, the average access time gradually decreases and to a steady value. We believe the reason is that, in the beginning of the experiment, the dominant operations are writes; but with the increase of cached entries, the dominant operations are reads. The experimental results have shown that our proposed CBLRU and CBSLRU behavior better than LRU in average access time of SSD. We believe that there are two main reasons: first, our proposed CBLRU and CBSLRU can reduce unnecessary write operations; second, CBLRU and CBSLRU change several small random writes to a large sequential write, which can reduce block erasure count inside SSD and improve the performance of SSD. In this experiment, the average access time are reduced by 13.20% and 43.83% in CBLRU and CBSLRU compared to LRU.

VIII. CONCLUSION

In this paper, we propose an SSD-based hybrid storage architecture and SSD-based data management policies for large-scale search engines. We have made three contributions in this paper. First, we analyze the I/O patterns of search engines, and choose an appropriate cache scheme for search engines. Second, in order to improve the I/O performance of SSD in the two-level architecture, we propose an improved log-based cache data management policy. Third, we propose appropriate data replacement policies for SSD. The experimental results demonstrate our proposed policies.

There are several interesting problems that need further discussion. First, in our research, we assume that the index files stored on SSD are static without any change. The situation of the dynamic scenario is required to further study in the future. Second, in order to reduce write operations and improve the cache performance, we can also consider the three-level cache scheme, namely results, inverted lists and intersections [19]. We believe that a good policy to determine when to make intersections will further improve the performance.

ACKNOWLEDGMENTS

We thank Zhao Zhang at Iowa State University, and Zhichun Zhu at University of Illinois at Chicago for their valuable advices and insightful comments. This research is partially supported by National Natural Science Foundation of China under grants 61173170 and 60873225, National High Technology Research and Development Program of China under grant 2007AA01Z403, Innovation Fund of Huazhong University of Science and Technology under grants 2012TS052, 2011TS135 and 2010MS068, and CCF Opening Project of Chinese Information Processing.

REFERENCES

[1] "Google plans," http://www.informationweek.com/news/storage/systems/showArticle.jhtml?articleID=207602745.

[2] "Google platform," http://en.wikipedia.org/wiki/Google_platform.

[3] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD Lifetimes with Disk-Based Write Caches," Proc. of the 8th USENIX Conference on File and Storage Technologies (FAST 10), pp.101-114, 2010.

[4] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "SSD bufferpool extensions for database systems," Proc. of the PVLDB 2010, vol.3, no.2, pp.1435-1446, 2010.

[5] F. Chen, D. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," Proc. of ACM SIGMETRICS 2009, Seattle, pp.181-192, 2009.

[6] Intel Corporation, "Understanding the flash translation layer (FTL) specification," Technical Report AP-684, 1998.

[7] J. Kim, J.M. Kim, S.H. Noh, S. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compactflash Systems," IEEE Trans. on Consumer Electronics, vol.48, no.2, pp.366-375, 2002.

[8] J. Kang, H. Jo, J. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," Proc. of the 6th ACM IEEE International conference on Embedded software, Seoul, Korea, pp.161-170, 2006.

[9] 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 Trans. on Embedded Computing Systems, vol.6, no.3, pp.18-es, 2007.

[10] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 09), Washington, DC, USA, vol.44, no.3, 2009.

[11] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel R turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. Transactions on Storage, 4(2):1-24, 2008.

[12] Panabaker, Ruston. Hybrid Hard Disk and ReadyDrive Technology: Improving Performance and Power for Windows Vista Mobile PCs. http://www.microsoft.com/whdc/ system/sysperf/accelerator.mspx.

[13] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: a replacement algorithm for flash memory," Proc. of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Seoul, Korea, pp.234-241, 2006.

[14] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR:integration of LRU and writes sequence reordering for flash memory," IEEE Trans. on Consumer Electronics, vol.54, no.3, pp.1215-1223, 2008.

[15] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," Proc. of the 6th USENIX Conference on File and Storage Technologies (FAST 08), San Jose, California, pp.1-14, 2008.

[16] Q. Gan and T. Suel, "Improved techniques for result caching in web search engines," Proc. of the 18th International Conference on World Wide Web (WWW 09), Madrid, Spain, pp.431-440, 2009.

[17] T. Fagni, R. Perego, F. Silvestri, and S. Orlando, "Boosting the performance of Web search engines: Caching and prefetching query results by exploiting historical usage data," ACM Trans. on Information Systems, vol.24, no.1, pp.51-78, 2006.

[18] P. Saraiva, E. de Moura, N. Ziviani, W. Meira. R. Fonseca, and B. Ribeiro-Neto, "Rank preserving two-level caching for scalable search engines," Proc. of the 24th Annual SIGIR Conference on Research and Development in Information Retrieval (SIGIR 01), New Orleans, Louisiana, USA, pp.51-58, 2001.

[19] X. Long and T. Suel, "Three-level caching for efficient query processing in large web search engines," Proc. of the 14th International Conference on World Wide Web (WWW 05), Chiba, Japan, pp.369-395, 2005.

[20] B. Huang and Z. Xia, "Allocating inverted index into flash memory for search engines," Proc. of the 20th International Conference Companion on World Wide Web (WWW 11), Hyderabad, India, pp.61-82, 2011.

[21] T. Pritchett and M. Thottethodi, "SieveStore: a highly-selective, ensemble-level disk cache for cost-performance," Proc. of the 37th Annual International Symposium on Computer Architecture, Saint-Malo, France, pp.163-174, 2010.

[22] "UMass Trace Repository," http://traces.cs.umass.edu/index.php/Storage/Storage.

[23] "Lucene," http://lucene.apache.org/java/docs/index.html.

[24] "SSD Trim," http://www.ssdtrim.com.

[25] H. Kim and U. Ramachandran, "FlashLite: a userlevel library to enhance durability of SSD for P2P File Sharing," Proc. of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS 09), Montreal, QC, pp.534-541, 2009.

[26] "FlashSim," http://csl.cse.psu.edu/?q=node/322.