Hot Random Off-loading: A Hybrid Storage System With Dynamic Data Migration

Lin Lin, Yifeng Zhu, Jianhui Yue, Zhao Cai and Bruce Segee
Department of Electrical and Computer Engineering,
University of Maine, Orono, USA
{llin, zhu, jyue, zcai, Segee}@eece.maine.edu

***Abstract*—Random accesses are generally harmful to performance in hard disk drives due to more dramatic mechanical movement. This paper presents the design, implementation, and evaluation of Hot Random Off-loading (HRO), a self-optimizing hybrid storage system that uses a fast and small SSD as a by-passable cache to hard disks, with a goal to serve a majority of random I/O accesses from the fast SSD. HRO dynamically estimates the performance benefits based on history access patterns, especially the randomness and the hotness, of individual files, and then uses a 0-1 knapsack model to allocate or migrate files between the hard disks and the SSD. HRO can effectively identify files that are more frequently and randomly accessed and place these files on the SSD. We implement a prototype of HRO in Linux and our implementation is transparent to the rest of the storage stack, including applications and file systems. We evaluate its performance by directly replaying three real-world traces on our prototype. Experiments demonstrate that HRO improves the overall I/O throughput up to 39% and the latency up to 23%.**

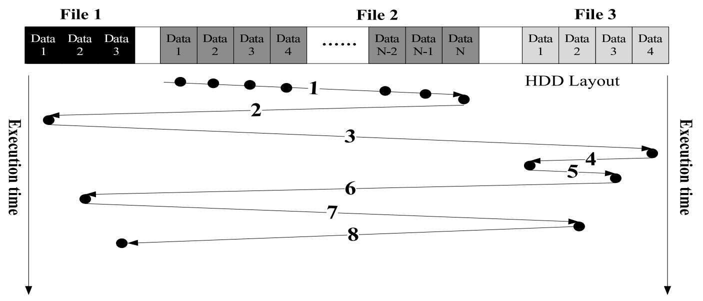
I. Introduction

The increasingly widening speed gap between hard disks and memory systems is a major performance bottleneck in computer systems. Under random accesses, disk heads frequently move to different noncontiguous physical locations and such slow mechanical movements introduce significant delay. Many researchers have made great efforts to improve the random access latency, such as group disk layout [1], [2], prefetch [3], [4] and I/O scheduler [5], [6]. This paper exploits solid state devices (SSDs) to address the performance issue. Unlike magnetic hard disks, SSDs use non-volatile memory chips and contain no moving components. The read and write performance of SSD is asymmetric but their response time is almost constant. While SSDs have the overhead of erasure-before-write, the write latency in SSD has recently been dramatically improved. Currently, the random read and write performance of SSD is one to two orders of magnitude better than hard disks, as shown in Figure 2(c).

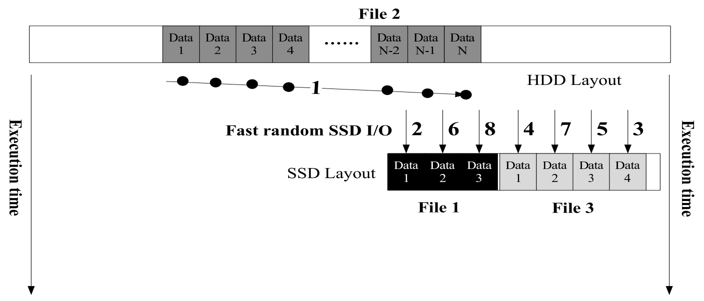
Nowadays SSDs, especially high-end ones, are still much more expensive than hard disks in terms of gigabytes per dollar [7]. Combined with the concerns of reliability due to limited numbers of erasure cycles, the cost of SSDs has impeded their applications as the major storage media, especially in enterprise systems. Thus SSDs cannot replace hard disks as the primary storage media in the industry in the near future. On the other hand, throughput-price ratio of SSD is approximately 2 orders of magnitude better than hard disks if one measures random I/Os per second per dollar [8]. SSDs provide extraordinarily fast performance for workloads with intensive random accesses. Such workloads are notoriously harmful to performance for hard disks. This key advantage gives us an exciting opportunity to build a hybrid system composed of one or more slow but large-capacity hard disk(s) and a small but fast SSD. The capacity of the SSD in such a hybrid storage system can be as small as 1% of the disk capacity. For example, it may consist of a SSD with a capacity of only ten gigabytes and disks with multiple terabytes. The hybrid storage leverages the fast random access performance in SSDs to boost the overall I/O performance without generating a large cost overhead.

The key challenging research issue in such a hybrid storage system is how to dynamically allocate or migrate data between the SSD and the disks in order to achieve the optimal performance gain. In this paper, we propose a hybrid storage architecture that treats the SSD as a by-passable cache to hard disks, and design an online algorithm that judiciously exchanges data between the SSD and the disks. Our basic principle is to place hot and randomly accessed data on the SSD, and other data, particularly cold and sequentially accessed data on hard disks. Our hybrid storage system, called Hot Random Off-loading (HRO), is implemented as a simple and general user-level layer above conventional file systems in Linux and supports standard POSTIX interfaces, thus requiring no modifications to underneath file systems or users applications. This prototype is comprehensively evaluated by using a commodity hard disk and SSD.

HRO dynamically and transparently places data on either SSD or disks based on the history I/O activities, including the randomness and the hotness. The randomness measures how far the target data of two consecutive requests typically are isolated in disk physical or virtual layout, and the hotness estimates how often data are accessed. Since the SSD used in our hybrid storage system is limited in capacity, we model the data allocation issue as a classic 0-1 knapsack problem. We exploit a simple but fast approximate solution to dynamically migrate data between SSD and hard drives. Data migration is not executed very frequently and is only triggered when the storage system is mostly idle, such as midnights, thus incurring little interference to foreground applications. After the migration, future accesses to these data are automatically and transparently redirected to the corresponding devices. As a result of data migration and traffic redirection, HRO achieves the following I/O performance benefits:

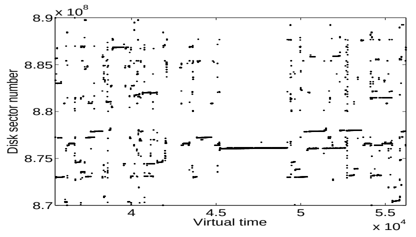


(a) **Original disk layout.** Accesses are made in the order from 1 to 8. Due to the long inter-file distance between file 1 and file 3, long seeks occur if accesses to these files are interleaved.

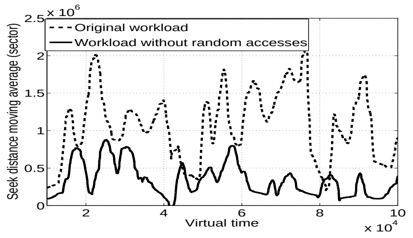


(b) **Expected hybrid layout.** By offloading random accesses to SSD, most hot random accesses are served by SSD, and most sequential accesses are served by the hard disk. Long-distance seeks are eliminated in this example.

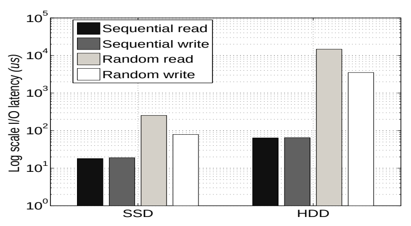
Fig. 1. **A simple example motivating our research**



(a) **I/O access traces captured at the block level.**



(b) **Average seek distance comparison.**



(c) **Log scale I/O latency of hard disk and SSD.**

Fig. 2. Plot a gives a snapshot of the access sequence of disk sectors captured by a tool called *blktrace* when replaying the mail server workload. It shows that even after the I/O scheduler in the operating systems reorders and merges I/O requests, there still exists a large variety of random accesses to the hard disk. The virtual time is defined as the number of references issued so far and is incremented for each request. In plot b, it is observed that if we filter out random accesses, the average disk seek distance is reduced significantly. Plot c clearly shows that random operation latency of the hard disk is much inferior to SSD.

- Hot random accesses are off-loaded from slow hard disks to fast SSD, i.e., a random access is served by the SSD with a very large likelihood. This is motivated from the fact that random access in SSD are two orders of magnitude faster than hard disks as shown in Figure 2(c).
- The access locality of data traffic to hard disks becomes stronger when most random accesses are filtered out and redirected to the SSD. Accordingly, seek and rotational latencies in hard disks are significantly reduced.

We evaluate our design and implementation by first reconstructing the file systems image and then replaying these traces on the constructed image. We use three I/O intensive workloads, including a mail server workload, a research workload, and an office workload. Experimental results show that HRO improves the overall I/O throughput up to 39% and the latency up to 23%.

The key contributions of this paper are summarized as follow. (1) We design and implement a hybrid storage system in Linux, called HRO, which is shown to significantly improve the storage system performance, under three representative workloads tested in this paper. (2) We develop a 0-1 knapsack optimization model that is based on the estimates of randomness and hotness to dynamically migrate files between disks and the SSD, with a goal to redirect hot and random accesses to SSD.

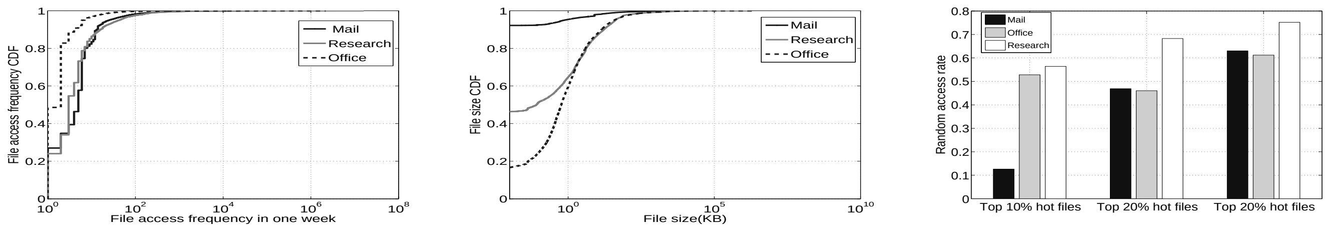
The rest of the paper is organized as follows. Motivation and characteristics of I/O workloads are discussed in Section II. Design and implementation details are given in Section III. Experimental results are presented in Section IV. Section V summarizes related work. Conclusions are made in Section VI.

II. RESEARCH MOTIVATIONS

A. Motivations

Hard drive disks often do not perform well in a multi-task environment due to the loss of access locality caused by the interleaving of multiple disk I/O streams. For example, on a typical server, such as a file server, a mail server, or a web server, many processes run independently in a time-sharing fashion. Even if data requested by each process is sequentially stored on a disk, we find that multiplexing between requests from different processes degrades the I/O performance signaficantly. Although disk scheduling algorithms, such as CFQ and anticipatory scheduling [6], has been used to reduce the impact of multiplexing, the access locality is still difficult to preserve due to the intrinsic nature of time-sharing and fairness enforcement.

We use a simple example to illustrate that disk performance suffers when multiple I/O processes concurrently access the shared storage system, no matter whether these accesses are sequential or random. In Figure 1(a), there are three hot files, including file 1, 2 and 3, which are often accessed in that order. Each black circle in the figure is an I/O access. Assume that file 1 and file 2 are sequentially accessed, and file 3 is randomly accessed. These files are assumed to be placed non-contiguously on disks, as shown in Figure 1(a).



(a) File access frequency CDF (b) File size CDF (c) Random hot access.

Fig. 3. **Cumulative distribution function (CDF) of access frequencies, file sizes, randomness.** In plot *a*, a very small percentage of files attract a majority of I/O accesses. Plot *b* show that three workloads have totally different distributions of file sizes. Plot *c* indicates that non-sequential accesses exist in three workloads.

One can easily observe that while the sequentiality on file 2 benefits disk performance, the disk suffers from three aspects. Long-distance seek operations occur between file 1 and file 3. Accesses to file 1 becomes pseudo-sequentially as they are interfered by the accesses of file 2 and 3. Disk heads have to move back and forth when serving requests to file 3.

These observations motivate us to offload random accesses to SSD to improve the sequentality of accesses to disks. As illustrated in Figure 1(b), if file 1 and file 3 are placed on SSD, the disk accesses become completely sequential. In real systems, disk access patterns are more complicated than this example. However, the simple example intuitively illustrates how seek time and rotational delays can be significantly reduced via offloading hot and random accesses.

In real systems we often observe such similar harmful access patterns. Figure 2(a) presents the access sequence of disk sectors captured by *blktrace* [9] when replaying a mail server workload. In this workload, multiplexed I/O requests from different users are often not arranged into a fully sequential I/O request by the I/O scheduler or disk firmware. We conduct simple experiments to estimate how much improvement can potentially be achieved if all random accesses are filtered out. We remove most of the random accesses when replaying the mail server workload and compare the performance results to that of the original workload. Figure 2(b) computes the moving average of hard disk head seek distance in the original workload and the filtered workload. The results show that in the filtered workload the seek distances measured in sectors are significantly reduced, and some of them are reduced to almost zero.

B. Characteristics of Workloads in Real Systems

In this section, we examine the basic characteristics of typical server I/O workloads and elaborate on those characteristics that directly motivate our work. We focus on three modern server workloads [10], including a campus mail server workload (Mail), a department research workload (Research) and a department office workload (Office). Some statistics of these three workloads are summarized in Table I.

We find that a hot file tends to remain as hot in the near future in real systems. We identify the top 10%, 20%, 30%, and all most frequently accessed files of each day, and calculate the percentage of hot files in the following days that are also hot files of the first day. The results show that all workloads exhibit large overlaps of hot files across successive days during a weeklong period. This indicats that popularity is consistent across multi-day periods. In addition, we observe that only a very small subset of files are hot. In Figure 3(a), the hot files are only less than 3 percent of the total files. This is consistent with the conventional wisdom that a very small percentage of files attract most of the I/O accesses. Figure 3(b) shows the distributions of file sizes in the three workloads. In these three workloads, no very large files exist and most files are in KBs or MBs. As a result, a SSD with a small capacity might be sufficient to cache hot files, thus reduce the price overhead of a hybrid storage system. In Figure 3(c), the randomness of three workloads are compared. We can see that randomness exists in all cases, and especially in the top accessed files. Thus, HRO can migrate random access dominated data to mitigate the hard disk seek overhead.

TABLE I
STATISTICS OF THREE WORKLOADS STUDIED

	Office	Mail	Research
Trace period	One week	One week	One week
Total ops	211,308,494	187,974,468	29,550,778
Metadata ops	66%	14%	75%
Read	24%	65%	10%
Write	10%	21%	15%
Read (MB)	833,135	845,123	32,498
Write (MB)	242,376	313,987	61,488
R/W Ratio	3.4	2.5	0.5

III. DESIGN AND IMPLEMENTATION

In this section, we present more practical and realtistic details in our implementation of HRO. Figure 4 presents the architecture of HRO and its relation to the rest of the storage stack. HRO is built above Fuse [11] in Linux at the user space level, and is similar to the structure of Umbrella [12]. We implement HRO with about 4000 lines of C++ code. It automatically organizes files stored in different file systems into a single and shared virtual namespace. HRO supports the standard POXIS interface for the unified namespace. It automatically de-multiplexes the namespace operations among underneath native file systems. Users can mount the hybrid storage systems without knowing details of the storage structure and all the HRO background migration is transparent to the end user.

Positioning HRO above the file system layer and operating at the granularity of files is important for two important reasons. First, by operating above the file system layer, HRO becomes independent of the file system, and can thereby work seamlessly with any native file systems. It can also support multiple active file systems and mount-points simultaneously. Second, working at the file level can simplify the management complexity of the virtual hybrid layer, and reduce the performance and memory overhead significantly.

HRO consists of five components: data collector, mapping table, randomness calculator, allocation module and migrator. The following discusses each component in detail.

DATA COLLECTOR captures all access requests on the system call I/O path and collects these information for the randomness calculator. HRO intercepts every I/O request and identifies the corresponding physical file location from the mapping table, then sends the I/O request to the lower VFS and file system. At the same time, all requests are sent to the data collector and the randomness calculator. The allocation algorithm studies these requests information and identifies the data to be migrated. This execution is shown in Figure 4 as step 1 to 9.

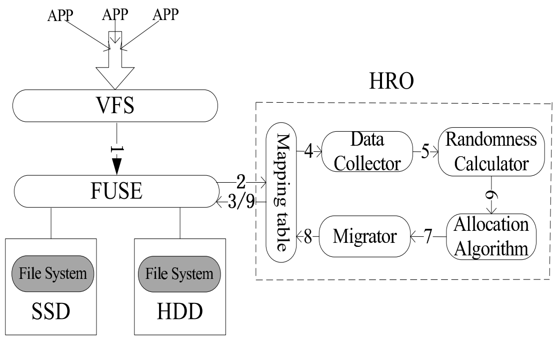


Fig. 4. **HRO architecture.** HRO is built by using FUSE and it is composed of five components: data collector, mapping table, randomness calculator, migrator, and allocation module.

The **MAPPING TABLE** is a hash table in memory to identify in which storage device, SSD or disks, a specific file is stored. The structure of the mapping table is shown in Figure 5. Specifically, it translates the logical file location to the physical file location which is indicated by $DeviceID$ and $Inode$. The hash key is the logical file location and each hash entry is a special data structure as explained in Table II. To benefit from the allocation of data, we must be able to quickly find them and send the actual I/O request to the corresponding storage device. The mapping table is composed of the hash entries for all files. The lookup, delete and add operations on the hash table are designed for high speed. In addition, the mapping table has a very small memory overhead. Only a total of 50 bytes are needed for each file entry in the table. A mapping table of 50MB can keep track of 1,000,000 frequently accessed files. As most of todays computer systems have multiple gigabytes of memory, it is usually not a problem to keep all hash entries in memory. For extremely large systems, the mapping table can be flushed to hard disks and loaded into memory on demand.

The **RANDOMNESS CALCULATOR** mainly evaluates the randomness of each individual file. Information such as last

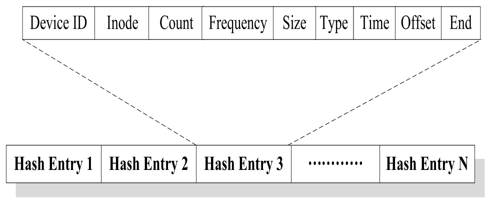


Fig. 5. **Mapping table structure.** The mapping table is composed of N hash entry, each hash entry is a data structure explained in Table II.

TABLE II
EXPLANATION OF FIELDS IN A HASH ENTRY

Field	Explanation
Device ID	File located device ID (8 bits)
Inode	File inode number in the file system (48 bits)
Count	File randomness count number (32 bits)
Size	File size (64 bits)
Frequency	File access frequency (32 bits)
Type	File last access type (8 bits)
Time	File last access time (32 bits)
Offset	File last access offset (64 bits)
End	File last access end (64 bits)

file access time, last access type, file size, file access frequency are updated upon each file access. In order to recognize the file random/sequential access pattern, Algorithm 1 is designed to logically merge sequential I/O requests that arrive within a short time window. The algorithm counts the total number of merged requests. Similar to the block-level I/O scheduler in many operating systems, we merge small requests into a larger sequential segment and keep independent random requests as they are. If the arrival interval of two requests is larger than a predefined threshold 0.5 second, then we treat these requests as non-sequential. After merging the requests, the total number of access segments remained for a given file is defined as the randomness count of that file. For example, if the total number of segments equals to the access frequency, then this file is accessed fully randomly, because no requests are merged. If the segment number equals to one, then all the requests are merged into one because of the perfect sequentiality of the requests.

In Algorithm 1, for each new request, we first locate the corresponding hash entry in the mapping table. From line 3 to line 5, the algorithm compares the sequentiality of the new request with previous I/Os. If the read/write type, access interval as well as the access offset match, then we just update the last I/O information and merge them as a sequential segment as shown in line 6 and 7. Otherwise, the new request is treated as a random one, the algorithm increases the randomness count R_i for file i and uses the new I/O request as the last I/O request as shown in line 11 and 12.

The **MIGRATOR** handles data movement between SSD and disks. It compares the new placement of data generated from the file allocation algorithm to their old placement, and identifies files that need to be migrated. It then schedules and optimizes the migration. Because migrations cause additional I/O traffic, thus care must be taken so that they do not affect the foreground I/O performance. In our design, HRO conducts migrations during system idle time such as midnight. For simplicity, in our experiments, all the allocation algorithms are executed only once each day and migrations are performed at midnight.

```
Algorithm 1: I/O requests merge and count algorithm
Input  : A new I/O request info.
Output: Updated hash entry information
1  foreach New I/O request do
2      Find the corresponding hash entry;
3      if I/O type equals to the last access type then
4          if I/O interval time smaller than 0.5s then
5              if New I/O start address equals to last I/O end address then
6                  Last I/O end ← new I/O end;
7                  Update last I/O access time;
8              end
9          end
10     else
11         Replace the last I/O info. with new I/O info.;
12         $R_i = R_i + 1$;
13     end
14 end
```

The **ALLOCATION ALGORITHM** is specially designed to maximize the utilization of SSD in our hybrid storage systems. The capacity of SSD is small in our hybrid systems since SSD are much more expensive than disks. Which data and how much data should be placed on SSD is a key issue for improving the overall performance. We find that there is a similarity between HRO data placement issue and the classic 0/1 knapsack problem. The placement problem can be summarized as following: Given a set of files, each with a size and a benefit value, determine the selection of files to be included in a container so that the total size is less than a given limit and the total value is as large as possible.

We create a 0/1 knapsack problem model for HRO to optimize the data allocation efficiency. First, we need to determine what is the benefit value of storing a file on SSD. Because hard disks are slow for random accesses and SSD is extraordinary fast for random accesses, we should take the randomness and hotness into considerations. Intuitively, the benefit of storing a file on SSD increases if the file is more randomly accessed and/or the file is more frequently accessed. As a result, the benefit value v_i of file i is empirically defined as follows in this paper

$$v_i = \frac{f_i}{s_i} \cdot R_i \tag{1}$$

where f_i is the access frequency of file i, s_i is the file size, and the R_i is the file random access count calculated in Algorithm 1. The benefit value v_i is defined as the multiplication of access frequency per unit size with the randomness count R_i, which means that if a file has a high access frequency, a small file size, and a large the randomness count R_i, then it is regarded as a high benefit value file.

Mathematically the 0-1-knapsack data allocation problem can be formulated as the following equations. In HRO, we have n different files. Each file i has a benefit value of v_i and a size of s_i. The variable x_i indicates the target device of each file. x_i is either 0 or 1; 0 means placing the file on the hard disk, and 1 means storing the file in SSD. HRO selects a combination of files to maximize the total benefit value with the subject to the SSD capacity limitation. This model is capable of finding the most valuable files from previous I/O workloads and choosing them as candidates for migration.

$$Maxmize \sum_{i=1}^{n} x_i v_i$$

$$Subject\ to : \sum_{i=1}^{n} s_i x_i \leq CAPACITY_{ssd}, \qquad x_i \in \{0, 1\}$$

However, the 0-1-knapsack is a classic NP-complete problem and no efficient algorithms have been found. In all three workloads studied in this paper, the real file system image contains millions of files, and thus the non-polynomial-time solution is unacceptable in real systems. In HRO, we deployed a polynomial-time approximation algorithm to keep the problem simple while achieving a reasonably good migration performance.

```
Algorithm 2: Data allocation algorithm
Input  : $Mapping\ table$, $FileSet_{ssd}$
Output: $SET_{ssd}$, $SET_{hdd}$
1  $MoveSet \leftarrow empty$;
2  $TotalSize \leftarrow 0$;
3  forall $Mapping\ table\ entry$ do
4      Sort all items in the decreasing order of $v_i/s_i$;
5  end
6  for $i \leftarrow 1$ to $n\ and\ TotalSize < CAPACITY_{ssd}$ do
7      if $TotalSize + size_i \leq CAPACITY_{ssd}$ then
8          $TotalSize \leftarrow TotalSize + size_i$;
9          $x_i \leftarrow 1$;
10         $Add\ i\ to\ MoveSet$;
11     end
12 end
13 $SET_{ssd} \leftarrow MoveSet - (MoveSet \bigcap FileSet_{ssd})$;
14 $SET_{hdd} \leftarrow FileSet_{ssd} - (MoveSet \bigcap FileSet_{ssd})$;
```

In Algorithm 2, the input is the mapping table and the file set $FileSet_{ssd}$ which includes all the files currently stored on SSD. The output is the file set SET_{ssd}(files need to be migrated to SSD from HDD) and file set SET_{hdd}(files need to move from SSD to HDD). It sets $MoveSet$ to empty, which is the moving set generated by HRO allocation algorithm without comparing with previous actual file location. $TotalSize$ is set to zero initially, which is the total size of files that is scheduled to move. First it calculates and sorts the file value in decreasing order, as shown in line 4. Then files to be moved to SSD are selected according to its value in a greedy way. For all the remaining best value files, if the file size plus current $TotalSize$ is smaller than the SSD capacity, then $TotalSize$ increases and the file is indicated to be stored on SSD, as shown in line 6 to line 12. After the

TABLE III
CONFIGURATION OF STORAGE SERVER

Components	Specification
Operating system	Ubuntu 10.04 with kernel 2.6.31
File system	Ext3
CPU	AMD Opteron dual core 1000 Hz
Memory	1G DDR2 667Hz
SSD	OCZ-AGILITY2
Capacity	60GB
Sequential Read/Write	20us/70us
Random Read/Write	270us/375us
Hard Disk	3* WDC WD7500AAKS
Capacity	750GB
Rotational speed	7200RPM
Read Seek	8.9ms
Track-to-track Seek	2ms
Full Stroke Read Seek	21ms

loop, we compare the moving set with the file set currently on SSD, and add files that are in the $MoveSet$ but not on SSD into SET_{ssd}. Similarily, files that need to be moved to hard drives are added to SET_{hdd}. The complexity of this algorithm is only $O\left(n \cdot logn\right)$. This allocation algorithm is only an approximation solution to our problem. However, we found that it is efficient and achieves very good performance in our experiments.

IV. EXPERIMENTAL STUDY

We use three representative workloads to evaluate our design in this section. We compare HRO against two basic systems: a conventional storage system based on disks, and a hybrid storage systems in which the SSD stores the most frequently accessed files. Note that the placement policy in HRO considers both hotness and randomness. In our experiments, the usable storage capacity of SSD is set approximately 5% of the size of file system images and less than 1% of the hard disk capacity, in order to minimize the cost overhead. The same type of disks is used in HRO and the baseline systems. The SSD in HRO is used as a by-passable cache to disks. We do not compare HRO with hybrid systems that simply organize SSDs and disks into storage RAIDs since such systems require SSDs with a very large storage capacity.

In this section, we evaluate HRO and the baseline systems by using two performance metrics: the I/O bandwidth and the I/O latency. In order to examine the impact of off-loading, we record all block-level requests issued by the disk scheduler at runtime and calculate average seek distances. Since most of randomly accessed files are allocated to the SSD and most of sequentially accessed data are kept on disks, the average seek distance on disks is expected to decrease significantly.

A. Experimental Setup

Our test bed is composed of two machines: the monitor machine and the storage server. The storage server configuration is given in Table III. The memory size is set to 1GB to minize the impact of the buffer cache and the buffer cache is cleared during each experiments. In order to remove the interference of requests that are not issued by our test benchmarks, the operating system is installed on a separate hard disk. Another two hard disks and a SSD are used in our experiments. NFS traces, collected in three large servers, are replayed on our prototype implementation via a modifed replay tool called *dbench* [13]. The description of these traces have been given in section II. At the same time, block-level requests are captured by *blktrace* [9] and these requests are sent to the monitor machine for block level analysis. We have developed a small toolkit to analyze the I/O traces and then reconstruct the original file system image. The toolkit also translates the original NFS traces into the dbench [13] NFS replay format.

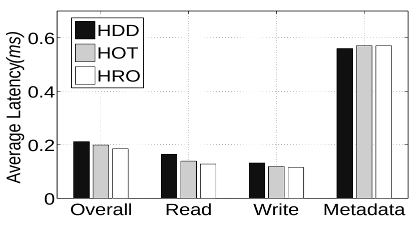
While the original I/O traces are collected over a period of several weeks, due to time limitation, we only replay a subset of traces for each workload, with each subset containing one week of I/O activities. Migration is trigged during midnight when the system is typically idle. We use an unlimited acceleration factor to reduce the experiment time and the next workload request is issued as soon as the pervious one completes. The trace replay acceleration factor is reduced to 1X when the migration is performed. While the capacity of SSD used in our experiment has 60GB, we artificially limit the usable space on SSD to 5% of the total file system images, i.e., 2.5GB for the mail workload, 12GB for the office workload and 7GB for the research workload. The capacity of SSD is less than 1% of hard disk, which is cost efficient for the HRO hybrid architecture.

B. System Bandwidth

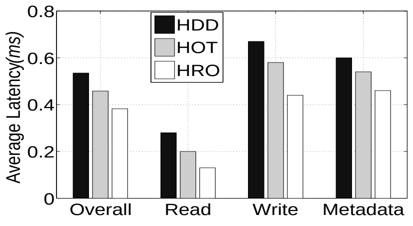
HRO successfully outperforms the two baseline systems and achieves much higher bandwidth in three workloads tested in this paper, as shown in Figure 8. The baseline of conventional disk based system is denoted as HDD, and the hybrid storage system with frequency only based migration strategy is denoted as HOT in the rest of this paper. For the mail server workload, HRO outperforms HDD and HOT by up to 15% and 6%, respectively, in bandwidth. With a very small size SSD, HRO improves the bandwidth significantly compared with HDD, and the improvement is mainly made by migration of hot random data to SSD. In the office workload, the read/write ratio is about 3.5, and it is the busiest workload in our experiment. As a result, the performance improvement is the most significant over the three workloads. For all seven days, the HRO bandwidth is consistently better than that of HDD and HOT. In average, the bandwidth of HRO in the office workload is 39% and 21% better than HDD and HOT, respectively. The research workload is dominated by writes, and the read/write traffic ratio is about 0.5, which is not friendly for SSD because of heavy background garbage collection on SSD. In addition, this workload contains a large amount of metadata operations. Therefore, compared with the other two workloads, the HRO improvement is relatively smaller in the research workload.

C. System Latency

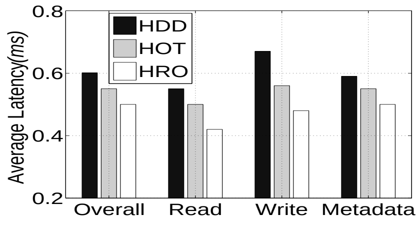
Figure 6(a) compares the latencies of these three systems under the mail server workload. HRO improves the average latency of HDD and HOT by 12% and 8% respectively. Most



(a) **Mail server workload**

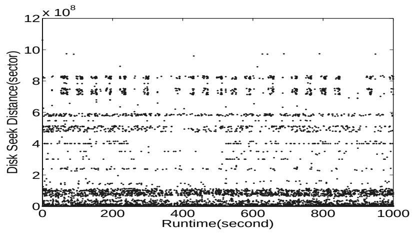


(b) **Office workload**

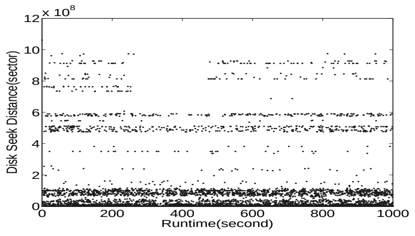


(c) **Research workload**

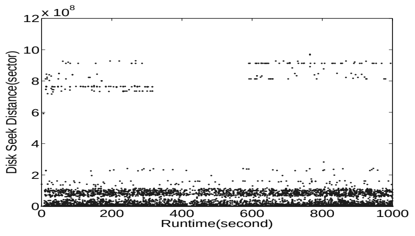
Fig. 6. **Average latency in three workloads.**



(a) HDD



(b) HOT



(c) HRO

Fig. 7. **Disk seek distance over the time under the mail server workload.** By offloading hot random data to SSD, HRO significantly mitigates the hard disk seek distance.

importantly, HRO can reduce the read latency of HDD and HOT by 22% and 10%, and reduce the write latency of HDD by 13%. The significant read/write improvement indicates that HRO can successfully capture hot and random data and place them on SSD. For metadata operations, these three schemes almost have the same performance.

For the office workload, HRO reduces the average latency of HDD and HOT by 23% and 17% respectively as shown in Figure 6(b). The average read latency is reduced by 24% and 16% while the average write latency is reduced by 29% and 14%. Metadata operation performance is also improved correspondingly, because there exists a large amount of metadata operations in this workload. Figure 6(c) shows the average system latency improvement. HRO reduces the average latency of HDD and HOT by 16% and 10%, the average read latency by 21% and 15%, and the average write latency by 26% and 14%.

D. Reduce disk seek distances analysis

When replaying the file-level I/Os specified in the traces during the experiments, we also capture the actual block-level access sequences to the disks and the requests served by the buffer cache are not included in the sequence. Due to the space limitation, we only take the mail server workload as an example to illustrate the experiment results.

Figure 7 plots the disk seek distances observed in three schemes. Apparently, HDD has the largest amount of long-distance seek operations caused by the random accesses and long-distance seeks are reduced by HOT. Impressively, most of large seeks occurred in the HDD system is now eliminated in HRO, as shown in Figure 7. Figure 9 compares the moving average of seek distances of these three schemes and the moving window size is set as 5 requests. This figure clearly shows that HRO can successfully mitigate most of the random access and reduce the disk head seek distance of HDD and HOT with an average of 69% and 53%, respectively. The reduction in disk seek distances is then directly translated into the performance gain.

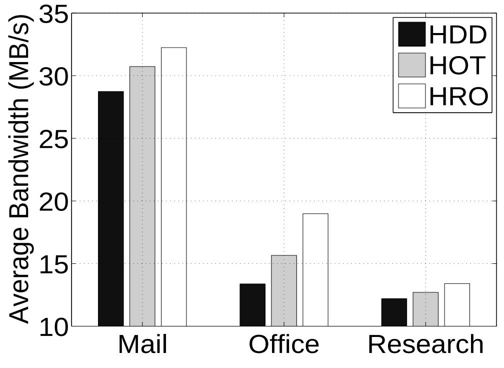


Fig. 8. **Average bandwidth of three workloads.**

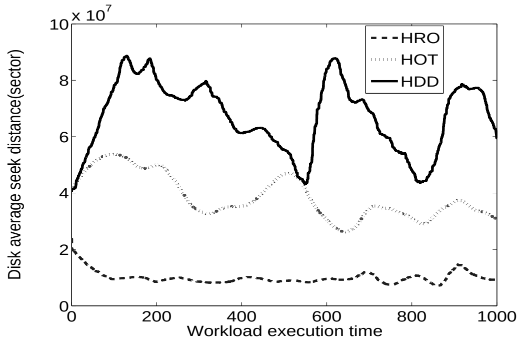


Fig. 9. **Disk seek distance runtime average.**

V. RELATED WORK

A. Adaptive Disk Layout

Many researchers have made great efforts to reduce or hide long-seek operations in disks. FFS [14] and its successors [15] improve disk bandwidth and latency by placing related data objects (e.g., data blocks and inodes) physically close to each other on a disk drive. BORG [1] is a self-optimizing storage system that performs automatic block reorganization based on the observed I/O workload. Related blocks are moved together to mitigate the disk seek distance. FS2 [2] proposes replication of frequently accessed blocks based on disk access patterns in file system free space to reduce seek distance and energy consumption. C-FFS [16] advocates co-locating inodes and file blocks for small files, which can reduce the latency by accessing inodes and file data together. HFS [17] combines the strengths of FFS and LFS while avoiding their weaknesses. This is accomplished by distributing file system data into two partitions based on their size and type.

B. Hybrid Storage and Cache Technique

In [18], the authors create a hybrid system composed of a SSD and a hard disk used in a database environment. All operations are treated as fixed size random operations. It places read dominated blocks on the SSD while allocates write dominated blocks on the hard disk. Another work [19] configures a SSD-based multi-tier system with the optimal number of devices per tier to achieve performance goals at minimum cost. Umbrella [12] is a file system level approach to combine multiple device in a unified namespace. [20] allocates data block according to the optimization equations to maximize the hybrid system performance. DCD [21] uses a small log disk, referred to as cache-disk, as a secondary disk cache to optimize disk write performance. Write off-loading [22] allows write requests on spun-down disks to be temporarily redirected to persistent storage elsewhere in the data center to save energy. Griffin [23] extends SSD life time and maintains acceptable system performance. FlashVM [24] is a system architecture and a core virtual memory subsystem built in the Linux kernel that uses dedicated flash for paging.

VI. CONCLUSION

This paper presents a cost-effective hybrid storage system that leverages the fast SSD as a by-passable cache of slow disks to offload workloads that are harmful to disk performance. Our hybrid storage system, called HRO, takes both the randomness and hotness into considerations when deciding what data should be stored on SSD. Specially, HRO places data files that are accessed most frequently and in a random fashion to the SSD with a goal to reduce the number of long-distance seek operations on disks. We design an allocation algorithm based on the classic 0-1 knapsack optimization problem to dynamically migrate files between disks and the SSD. We implement our design in Linux and our implementation is transparent to underneath file systems and user applications. We evaluate our design and implementation by replaying three representative workloads on our implementation. Our experiments show that a SSD of a very small capacity (1% of the capacity of disks) can very effectively improve the overall I/O performance of disks by up to 39% and the latency up to 23%.

ACKNOWLEDGMENT

This work was supported by National Science Foundation under grants EAR-1027809, IIS-0916663, CCF-0937988, CCF-0621493 and EPS-0904155.

REFERENCES

[1] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, “Borg: block-reorganization for self-optimizing storage systems,” in *Proccedings of the 7th conference on File and storage technologies*, 2009.

[2] H. Huang, W. Hung, and K. G. Shin, “Fs2: dynamic data replication in free disk space for improving disk performance and energy consumption,” in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[3] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, “Diskseen: exploiting disk layout and access history to enhance i/o prefetch,” in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, 2007.

[4] S. VanDeBogart, C. Frost, and E. Kohler, “Reducing seek overhead with application-directed prefetching,” in *Proceedings of the 2009 conference on USENIX Annual technical conference*, 2009.

[5] A. Riska, J. Larkby-Lahet, and E. Riedel, “Evaluating block-level optimization through the io path,” in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, 2007.

[6] S. Iyer and P. Druschel, “Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o,” in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[7] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, “Migrating server storage to ssds: analysis of tradeoffs,” in *Proceedings of the 4th ACM European conference on Computer systems*, 2009.

[8] “Storage devices iops comparison,” http://en.wikipedia.org/wiki/IOPS.

[9] “Blktrace block level utility,” http://linux.die.net/man/8/blktrace.

[10] “Harvard sos project,” http://www.eecs.harvard.edu/sos/traces.html.

[11] “Fuse: Filesystem in userspace,” http://fuse.sourceforge.net.

[12] J. A. Garrison and A. L. N. Reddy, “Umbrella file system: Storage management across heterogeneous devices,” *ACM Transactions on Storage*, March 2009.

[13] “Dbench file system benchmark,” http://dbench.samba.org/.

[14] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, “A fast file system for unix,” *ACM Transactions on Computer Systems*, vol. 2, 1984.

[15] R. Card, T. Tso, and S. Tweedle, “Design and implementation of the second extended filesystem,” *First Dutch International Symposium on Linux*, 1994.

[16] G. R. Ganger and M. F. Kaashoek, “Embedded inodes and explicit grouping: exploiting disk bandwidth for small files,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 1997.

[17] Z. Zhang and K. Ghose, “hfs: a hybrid file system prototype for improving small file and metadata performance,” *SIGOPS Oper. Syst. Rev.*, vol. 41, March 2007.

[18] I. Koltsidas and S. D. Viglas, “Flashing up the storage layer,” *Proc. VLDB Endow.*, vol. 1, August 2008.

[19] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, “Cost effective storage using extent based dynamic tiering,” in *Proceedings of the th USENIX Conference on File and Storage Technologies*, 2011.

[20] X. Wu and A. L. N. Reddy, “Exploiting concurrency to improve latency and throughput in a hybrid storage system,” in *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[21] Y. Hu and Q. Yang, “Dcd - disk caching disk: A new approach for boosting i/o performance,” in *In Proceedings of the 23rd International Symposium on Computer Architecture*. ACM Press, 1996.

[22] D. Narayanan, A. Donnelly, and A. Rowstron, “Write off-loading: practical power management for enterprise storage,” in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.

[23] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, “Extending ssd lifetimes with disk-based write caches,” in *Proceedings of the 8th USENIX conference on File and storage technologies*, 2010.

[24] M. Saxena and M. M. Swift, “Flashvm: virtual memory management on flash,” in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010.