Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems

Feng Chen David Koufaty
Circuits and Systems Research
Intel Labs
Hillsboro, OR 97124, USA
{feng.a.chen,david.a.koufaty}@intel.com

Xiaodong Zhang
Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210, USA
zhang@cse.ohio-state.edu

ABSTRACT

With the fast technical improvement, flash memory based Solid State Drives (SSDs) are becoming an important part of the computer storage hierarchy to significantly improve performance and energy efficiency. However, due to its relatively high price and low capacity, a major system research issue to address is on how to make SSDs play their most effective roles in a high-performance storage system in cost- and performance-effective ways.

In this paper, we will answer several related questions with insights based on the design and implementation of a high performance hybrid storage system, called *Hystor*. We make the best use of SSDs in storage systems by achieving a set of optimization objectives from both system deployment and algorithm design perspectives. Hystor manages both SSDs and hard disk drives (HDDs) as one single block device with minimal changes to existing OS kernels. By monitoring I/O access patterns at runtime, Hystor can effectively identify blocks that (1) can result in long latencies or (2) are semantically critical (e.g. file system metadata), and stores them in SSDs for future accesses to achieve a significant performance improvement. In order to further leverage the exceptionally high performance of writes in the state-of-the-art SSDs, Hystor also serves as a write-back buffer to speed up write requests. Our measurements on Hystor implemented in the Linux kernel 2.6.25.8 show that it can take advantage of the performance merits of SSDs with only a few lines of changes to the stock Linux kernel. Our system study shows that in a highly effective hybrid storage system, SSDs should play a major role as an independent storage where the best suitable data are adaptively and timely migrated in and retained, and it can also be effective to serve as a write-back buffer.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Secondary Storage

General Terms

Design, Experimentation, Performance

Keywords

Solid State Drive, Hard Disk Drive, Hybrid Storage System

ICS'11, May 31–June 4, 2011, Tuscon, Arizona, USA.

1. INTRODUCTION

High-performance storage systems are in an unprecedented high demand for data-intensive computing. However, most storage systems, even those specifically designed for high-speed data processing, are still built on conventional hard disk drives (HDDs) with several long-existing technical limitations, such as low random access performance and high power consumption. Unfortunately, these problems essentially stem from the mechanic nature of HDDs and thus are difficult to be addressed via technology evolutions.

Flash memory based Solid State Drive (SSD), an emerging storage technology, plays a critical role in revolutionizing the storage system design. Different from HDDs, SSDs are completely built on semiconductor chips without any moving parts. Such a fundamental difference makes SSD capable of providing one order of magnitude higher performance than rotating media, and makes it an ideal storage medium for building high performance storage systems. For example, San Diego Supercomputer Center (SDSC) has built a large flash-based cluster, called *Gordon*, for high-performance and data-intensive computing [3]. In order to improve storage performance, Gordon adopts 256TB of flash memory as its storage [24]. However, such a design, which is backed by a $20 million funding from the National Science Foundation (NSF), may not be a typical SSD-based storage solution for widespread adoption, because the high cost and relatively small capacity of SSDs will continue to be a concern for a long time [11], and HDDs are still regarded as indispensable in the storage hierarchy due to the merits of low cost, huge capacity, and fast sequential access speed. In fact, building a storage system completely based on SSDs is often above the acceptable threshold in most commercial and daily operated systems, such as data centers. For example, a 32GB Intel® X25-E SSD costs around $12 per GB, which is nearly 100 times more expensive than a typical commodity HDD. To build a server with only 1TB storage, 32 SSDs are needed and as much as $12,000 has to be invested in storage solely. Even considering the price-drop trend, the average cost per GB of SSDs is still unlikely to reach the level of rotating media in the near future [11]. Thus, we believe that in most systems, SSDs should not be simply viewed as a replacement for the existing HDD-based storage, but instead SSDs should be a means to enhance it. Only by finding the fittest position of SSDs in storage systems, we can strike a right balance between performance and cost. Unquestionably, to achieve this goal, it is much more challenging than simply replacing HDDs with fast but expensive SSDs.

1.1 Critical Issues

A straightforward consideration of integrating SSD in the existing memory hierarchy is to treat the state-of-the-art SSDs, whose cost and performance are right in between of DRAM memory and

HDDs, as a *secondary-level cache*, and apply caching policies, such as LRU or its variants, to maintain the most likely-to-be-accessed data for future reuse. However, the SSD performance potential could not be fully exploited unless the following related important issues, from both policy design and system deployment perspectives, be well addressed. In this paper, we present a unique solution that can best fit SSDs in the storage hierarchy and achieve these optimization goals.

1. Effectively identifying the most performance-critical blocks and fully exploiting the unique performance potential of SSDs – Most existing caching policies are temporal locality based and strive to identify the most likely-to-be-reused data. Our experimental studies show that the performance gains of using SSDs over HDDs is highly dependent on workload access patterns. For example, random reads (4KB) on an Intel® X25-E SSD can achieve up to 7.7 times higher bandwidth than that on an HDD, while the speedup for sequential reads (256KB) is only about 2 times. Besides identifying the most likely-to-be-reused blocks as done in most previous studies, we must further identify the blocks that can receive the most significant performance benefits from SSDs. We have systematically analyzed various workloads and identified a simple yet effective metric based on extensive experimental studies. Rather than being randomly selected, this metric considers both *temporal locality* and *data access patterns*, which well meets our goal of distinguishing the most performance-critical blocks.

2. Efficiently maintaining data access history with low overhead for accurately characterizing access patterns – A major weakness of many LRU-based policies is the lack of knowledge about deep data access history (i.e. recency only). As a result, they cannot identify critical blocks for a long-term optimization and thus suffer the well-known cache pollution problem (workloads, such as reading a streaming file, can easily evict all valuable data from the cache [14]). As a key difference from previous studies, we profile and maintain data access history as an important part of our hybrid storage. This avoids the cache pollution problem and facilitates an effective reorganization of data layout across devices. A critical challenge here is how to efficiently maintain such data access history for a large-scale storage system, which is often in Terabytes. In this paper, a special data structure, called *block table*, is used to meet this need efficiently.

3. Avoiding major kernel changes in existing systems while effectively implementing the hybrid storage management policies – Residing at the bottom of the storage hierarchy, a hybrid storage system should improve system performance without intrusively changing upper-level components (e.g. file systems) or radically modifying the common interfaces shared by other components. Some previously proposed solutions attempt to change the existing memory hierarchy design by inserting non-volatile memory as a new layer in the OS kernels (e.g. [18, 19]); some require that the entire file system be redesigned [34], which may not be viable in practice. Our design carefully isolates complex details behind a standard block interface, which minimizes changes to existing systems and guarantees compatibility and portability, which are both critical in practice.

In our solution, compared to prior studies and practices, SSD plays a different role. We treat the high-capacity SSD as a part of storage, instead of a caching place. Correspondingly, different from the conventional caching-based policies, which frequently update the cache content on each data access, we only periodically and asynchronously reorganize the layout of blocks across devices for a long-term optimization. In this paper, we show that this arrangement makes SSDs the best fit in storage hierarchy.

1.2 Hystor: A Hybrid Storage Solution

In this paper, we address the aforesaid four issues by presenting the design and implementation of a practical hybrid storage system, called *Hystor*. Hystor integrates both low-cost HDDs and high-speed SSDs as a *single* block device and isolates complicated details from other system components. This avoids undesirable significant changes to existing OS kernels (e.g. file systems and buffer cache) and applications.

Hystor achieves its optimization objectives of data management through three major components. First, by monitoring I/O traffic on the fly, Hystor automatically learns workload access patterns and identifies performance-critical blocks. Only the blocks that can bring the most performance benefits would be gradually remapped from HDDs to high-speed SSDs. Second, by effectively exploiting high-level information available in existing interfaces, Hystor identifies semantically-critical blocks (e.g. file system metadata) and timely offers them a high priority to stay in the SSD, which further improves system performance. Third, incoming writes are buffered into the low-latency SSD for improving performance of write-intensive workloads. We have prototyped Hystor in the Linux Kernel 2.6.25.8 as a stand-alone kernel module with only a few lines of codes added to the stock OS kernel. Our experimental results show that Hystor can effectively exploit SSD performance potential and improve performance for various workloads

1.3 Our Contributions

The contribution of this work is threefold. (1) We have identified an effective metric to represent the performance-critical blocks by considering both temporal locality and data access patterns. (2) We have designed an efficient mechanism to profile and maintain detailed data access history for a long-term optimization. (3) We present a comprehensive design and implementation of a high performance hybrid storage system, which improves performance for accesses to the high-cost data blocks, semantically-critical (file system metadata) blocks, and write-intensive workloads with minimal changes to existing systems. While we have prototyped Hystor as a kernel module in software, a hardware implementation (e.g. in a RAID controller card) of this scheme is possible, which can further reduce system deployment difficulty as a drop-in solution.

In the rest of this paper, we will first examine the SSD performance advantages in Section 2. We study how to identify the most valuable data blocks and efficiently maintain data access history in Section 3 and 4. Then we present the design and implementation of Hystor in Section 5 and 6. Section 7 presents our experimental results. Related work is presented in Section 8. The last section concludes this paper.

2. SSD PERFORMANCE ADVANTAGES

Understanding the relative performance strengths of SSDs over HDDs is critical to efficiently leverage limited SSD space for the most performance gains. In this section, we evaluate an Intel® X25-E 32GB SSD [13], a representative high-performance SSD, and compare its performance with a 15,000 RPM Seagate® Cheetah® 15.5k SAS hard disk drive, a typical high-end HDD. Details about the two storage devices and experiment system setup are available in Section 7.

In general, a workload can be characterized by its read/write ratio, random/sequential ratio, request size, and think time, etc. We use the Intel® Open Storage Toolkit [21] to generate four typical workloads, namely *random read*, *random write*, *sequential read*, and *sequential write*. For each workload, we set the queue depth of 32 jobs and vary the request size from 1KB to 256KB. All workloads directly access raw block devices to bypass the buffer cache

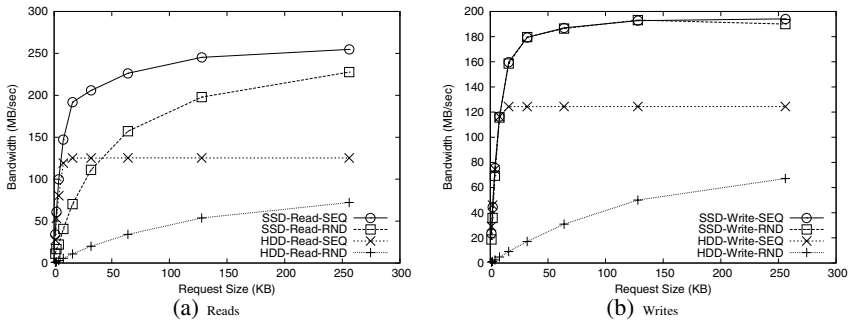


Figure 1: I/O Bandwidths for reads and writes on the Intel® X25-E SSD and the Seagate® Cheetah® HDD.

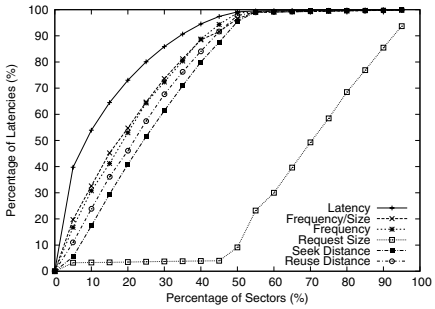


Figure 2: Accumulated HDD latency of sectors sorted using different metrics.

and file system. All reads and writes are synchronous I/O with no think time. Although real-life workloads can be a mix of various access patterns, we use the four synthetic microbenchmarks to qualitatively characterize the SSD. Figure 1 shows the experimental results. We made several findings to guide our system designs.

First, as expected, the most significant performance gain of running workloads on the SSD appears in random data accesses with small request sizes, for both reads and writes. For example, with a request size of 4KB, random reads and random writes on the SSD achieve more than 7.7 times and 28.5 times higher bandwidths than on the HDD, respectively. As request size increases to 256KB, the relative performance gains of sequential reads and writes diminish to 2 times and 1.5 times, respectively. It clearly shows that achievable performance benefits are highly dependent on workload access patterns, and we must identify the blocks that can bring the most performance benefits by migrating them into SSDs.

Second, contrary to the long-existing understanding about low write performance on SSDs, we have observed an exceptionally high write performance on the SSD (up to 194MB/sec). Similar findings have been made in recent performance studies on the state-of-the-art SSDs [5, 6]. As a high-end product, the Intel® X25-E SSD is designed for commercial environments with a sophisticated FTL design [13]. The highly optimized SSD internal designs significantly improve write performance and make it possible to use an SSD as a write-back buffer for speeding up write-intensive workloads, such as email servers.

Third, we can see that write performance on the SSD is largely independent of access patterns, and random writes can achieve almost identical performance as sequential writes. This indicates that it is unnecessary to specially treat random writes for performance purposes like in some prior work. This allows us to remove much unnecessary design complexity. In addition, we also find that writes on the SSD can quickly reach a rather high bandwidth (around 180MB/sec) with a relatively small request size (32KB) for both random and sequential workloads. This means that we can achieve the peak bandwidth on SSDs without need of intentionally organizing large requests as we usually do on HDDs.

Based on these observations, we summarize two key issues that must be considered in the design of Hystor as follows.

1. We need to recognize workload access patterns to identify the most *high-cost* data blocks, especially those blocks being randomly accessed by small requests, which cause the worst performance for HDDs.

2. We can leverage the SSD as a write-back buffer to handle writes, which often raise high latencies in HDDs. Meanwhile, we do not have to treat random writes specifically, since random writes on SSD can perform as fast as sequential writes.

3. HIGH-COST DATA BLOCKS

Many workloads have a *small* data set contributing a *large* percentage of the aggregate latency in data accesses. A critical task for Hystor is to identify the most performance-critical blocks.

3.1 Identifying High-Cost Blocks

A simple way to identify the high-cost blocks is to observe I/O latency of accessing each block and directly use the accumulated latency as an indicator to label the 'cost' of each block. In a hybrid storage, however, once we remap blocks to the SSD, we cannot observe their access latency on HDD any more. Continuing to use the previously observed latency would be misleading, if the access pattern changes after migration. Thus, directly using I/O latency to identify high-cost blocks is infeasible.

Some prior work (e.g. [12, 27]) maintains an on-line hard disk model to predict the latency for each incoming request. Such a solution heavily relies on precise hard disk modeling based on detailed specification data, which is often unavailable in practice. More importantly, as stated in prior work [12], as the HDD internals become increasingly more complicated (e.g. disk cache), it is difficult, if not impossible, to accurately model a modern hard disk and precisely predict the I/O latency for each disk access.

We propose another approach – using a pattern-related metric as an *indicator* to indirectly *infer* access cost without need of knowing the exact latencies. We associate each block with a selected metric and update the metric value by observing accesses to the block. The key issue here is that the selected metric should have a strong correlation to access latency, so that by comparing the metric values, we can effectively estimate the *relative* access latencies associated to blocks and identify the relatively high-cost ones. Since the selected metric is device independent, it also frees us from unnecessary burdens of considering specific hardware details (e.g. disk cache size), which can vary greatly across devices.

3.2 Indicator Metrics

In order to determine an effective indicator metric that is highly correlated to access latencies, we first identify four candidate metrics, namely *request size*, *frequency*, *seek distance*, *reuse distance*, and also consider their combinations. We use the *blktrace* tool [2] to collect I/O traces on an HDD for a variety of workloads. In the off-line analysis, we calculate the accumulated latency for each accessed block, as well as the associated candidate metric values. Then we rank the blocks in the order of their metric values. For example, concerning the metric *frequency*, we sort the blocks from the most frequently accessed one to the least frequently accessed one, and plot the accumulated latency in that order.

Figure 2 shows an example of TPC-H workload (other workloads are not shown due to space constraints). The X axis shows the top

percentage of blocks, sorted in a specific metric value, and the Y axis shows the percentage of aggregate latency of these blocks. Directly using latency as the metric represents the ideal case. Thus, the closer a curve is to the *latency* curve, the better the corresponding metric is. Besides the selected four metrics, we have also examined various combinations of them, among which **frequency/request size** is found to be the most effective one. For brevity, we only show the combination of *frequency/request size* in the figure. In our experiments, we found that *frequency/request size* emulates latency consistently better across a variety of workloads, the other metrics and combinations, such as seek distance, work well for some cases but unsatisfactorily for the others.

The metric **frequency/request size** is selected with a strong basis – it essentially describes both **temporal locality** and **access pattern**. In particular, *frequency* describes the temporal locality and *request size* represents the access pattern for a given workload. In contrast to the widely used recency-based policies (e.g. LRU), we use frequency to represent the temporal locality to avoid the well-recognized cache pollution problem for handling weak-locality workloads (e.g. scanning a large file would evict valuable data from the cache) [14]. It is also worth noting here that there is an *intrinsic correlation* between request size and access latency. First of all, the average access latency per block is highly correlated to request size, since a large request can effectively amortize the seek and rotational latency over many blocks. Second, the request size also reflects workload access patterns. As the storage system sits at the bottom of the storage hierarchy, the sequence of data accesses observed at the block device level is an optimized result of multiple upper-level components. For example, the I/O scheduler attempts to merge consecutive small requests into a large one. Thus, a small request observed at the block device often means that either the upper-level components cannot further optimize data accesses, or the application accesses data in such a non-sequential pattern. Finally, small requests also tend to incur high latency, since they are more likely to be intervened by other requests, which would cause high latencies from disk head seeks and rotations. Although this metric cannot perfectly emulate the ideal curve (latency), as we see in the figure, it performs consistently the best in various workloads and works well in our experiments.

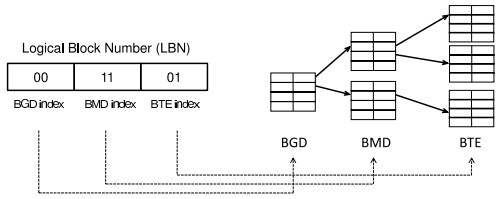
4. MAINTAINING DATA ACCESS HISTORY

To use the metric values to profile data access history, we must address two critical challenges – (1) how to represent the metric values in a compact and efficient way, and (2) how to maintain such history information for each block of a large-scale storage space (e.g. Terabytes). In short, we need an efficient mechanism to profile and maintain data access history at a low cost.

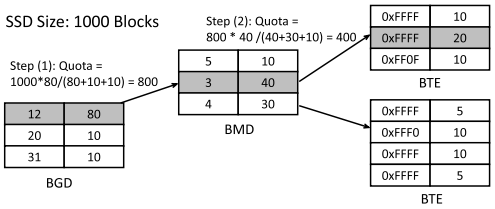
4.1 The Block Table

We use the *block table*, which was initially introduced in our previous work [15], to maintain data access history. Akin to the page table used in virtual memory management, the block table has three levels, *Block Global Directory* (BGD), *Block Middle Directory* (BMD), and *Block Table Entry* (BTE), as shown in Figure 3(a). The three levels, namely BGD, BMD, and BTE, of this structure essentially describe the storage space segmented in units of regions, sub-regions, and blocks, accordingly.

In the block table, each level is composed of multiple 4KB pages, each of which consists of multiple entries. A block's logical block number (LBN) is broken into three components, each of which is an index to an entry in the page at the corresponding level. Each BGD or BMD entry has a 32-bit *pointer* field pointing to a (BMD or BTE) page in the next level, a 16-bit *counter* field recording data



(a) *The block table structure*



(b) *Traversing the block table*

Figure 3: The Block Table. Each box represents an entry page. In BGD and BMD pages, left and right columns represent *unique* and *counter* fields. In BTE pages, two columns represent *flag* and *counter* fields. The two steps show the order of entries being traversed from BGD to BTE entries.

access information, and a 16-bit *unique* field tracking the number of BTE entries belonging to it. Each BTE entry has a 16-bit *counter* field and a 16-bit *flag* field to record other properties of a block (e.g. whether a block is a metadata block). This three-level tree structure is a very efficient vehicle to maintain storage access information. For a given block, we only need three memory accesses to traverse the block table and locate its corresponding information stored in the BTE entry.

4.2 Representing Indicator Metric

We have developed a technique, called *inverse bitmap*, to encode the *request size* and *frequency* in the block table. When a block is accessed by a request of *N* sectors, an *inverse bitmap*, *b*, is calculated using the following equation:

$$b = 2^{max(0,7-\lfloor log_2 N \rfloor)} \tag{4.1}$$

As shown above, inverse bitmap encodes request size into a single byte. The smaller a request is, the bigger the inverse bitmap is.

Each entry at each level of the block table maintains a *counter*. The values of the counters in the BGD, BMD, and BTE entries represent the 'hotness' of the regions, sub-regions, and blocks, respectively. Upon an incoming request, we use the block's LBN as an index to traverse the block table through the three levels (BGD→BMD→BTE). At each level, we increment the counter of the corresponding entry by *b*. So the more frequently a block is accessed, the more often the corresponding counter is incremented. In this way, we use the inverse bitmap to represent the size for a given request, and the counter value, which is updated upon each request, to represent the indicator metric *frequency/request size*. A block with a large counter value is regarded as a high-cost (i.e. hot) block. By comparing the counters associated with blocks, we can identify the blocks that should be relocated to the SSD.

As time elapses, a counter (16 bits) may overflow. In such a case, we right shift all the counters of the entries in the *same* entry page by one bit, so that we can still preserve the information about the relative importance that the counter values represent. Since such a right shift operation is needed for a minimum of 512 updates to a single LBN, this operation would cause little overhead. Also note

that we do not need to right shift counters in other pages, because we only need to keep track of the *relative* hotness for entries in a page, and the relative hotness among the pages is represented by the entries in the upper level.

The block table is a very efficient and flexible data structure to maintain the block-level information. For example, the full block table can be maintained in persistent storage (e.g. SSD). During the periodic update of the block table, we can load only the relevant table pages that need to be updated into memory (but at least one page at each level). Also note that the block table is a sparse data structure – we only need to maintain history for *accessed* blocks. This means that the spatial overhead in persistent storage is only proportional to the *working-set size* of workloads. In the worst case, e.g. scanning the whole storage space, the maximum spatial overhead is approximately 0.1% of the storage space (a 32-bit BTE entry per 4KB chunk). In practice, however, since most workloads only access partial storage space, the spatial overhead would be much lower. If needed, we can further release storage space by trimming the rarely updated table pages. This flexibility of the block table provides high scalability when handling a large storage space.

5. THE DESIGN OF HYSTOR

After introducing the indicator metric and the block table, we are now in a position to present the design of Hystor. Our goal is to best fit the SSD in the storage systems and effectively exploit its unique performance potential with minimal system changes.

5.1 Main Architecture

Hystor works as a pseudo block device at the block layer, as shown in Figure 4(a). The upper-level components, such as file systems, view it simply as a *single* block device, despite the complicated internals. Users can create partitions and file systems on it, similar to any directly attached drive. With minimal system changes, Hystor is easy to integrate into existing systems.

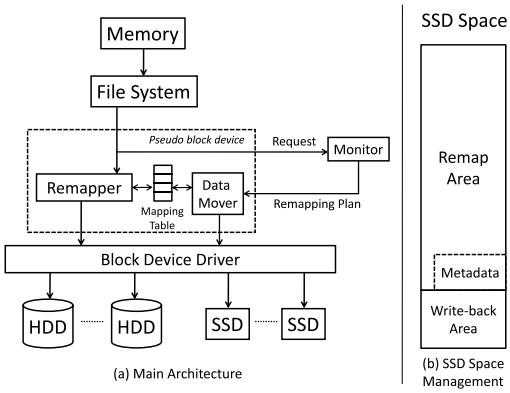


Figure 4: Architecture of Hystor.

Hystor has three major components, namely *remapper*, *monitor*, and *data mover*. The remapper maintains a mapping table to track the original location of blocks on the SSD. When an incoming request arrives at the remapper, the mapping table is first looked up. If the requested block is resident in the SSD, the request is redirected to the SSD, otherwise, it is serviced from the HDD. This remapping process is similar to the software RAID controller. The remapper also intercepts and forwards I/O requests to the monitor, which collects I/O requests and updates the block table to profile workload access patterns. The monitor periodically analyzes the data access history, identifies the blocks that should be remapped to the SSD, and requests the data mover to relocate data blocks across storage devices. The monitor can run in either kernel mode or user mode. The data mover is responsible for issuing I/O commands to the block devices and updating the mapping table accordingly to reflect the most recent changes.

5.2 Logical Block Mapping

Hystor integrates multiple HDDs and SSDs and exposes a linear array of logical blocks to the upper-level components. Each logical block is directly mapped to a physical block in the HDD and indexed using the logical block number (LBN). A logical block can be selected to remap to the SSD, and its physical location in the SSD is dynamically selected. Hystor maintains a *mapping table* to keep track of the remapped logical blocks. This table is also maintained in a statically specified location in the persistent storage (e.g. the first few MBs of SSD), and it is rebuilt in the volatile memory at startup time. Changes to the mapping table are synchronously written to the storage to survive power failures. In memory, the table is organized as a B-tree to speedup lookups, which only incur minimal overhead with several memory accesses. Since only remapped blocks need to be tracked in the mapping table, the spatial overhead of the mapping table is small and proportional to the SSD size. Techniques, similar to the dynamic mapping table [10], can also be applied to only maintain the most frequently accessed mapping entries to further reduce the in-memory mapping table size.

In essence, Hystor manages remapped blocks in an 'inclusive' manner, which means that, when a block is remapped to the SSD, its original home block in the HDD would not be recycled. We choose such an inclusive design for three reasons. First, the SSD capacity is normally at least one order of magnitude smaller than the HDDs, thus, there is no need to save a small amount of capacity for low-cost HDDs. Also, if we attempt to fully utilize the HDD space, a large mapping table has to be maintained to track every block in the storage space (often in granularity of TBs), which would incur high overhead. Second, when blocks in the SSD need to be moved back to the HDD, extra high-cost I/O operations are required. In contrast, if blocks are duplicated to the SSD, we can simply drop the replicas in the SSD, as long as they are clean. Finally, this design also significantly simplifies the implementation and avoids unnecessary complexity.

5.3 SSD Space Management

In Hystor, the SSD plays *a major role* as a storage to retain the best suitable data, and *a minor role* as a write-back buffer for writes. Accordingly, we logically segment the SSD space into two regions, *remap* area and *write-back* area, as shown in Figure 4(b). The remap area is used to maintain the identified critical blocks, such as the high-cost data blocks and file system metadata blocks. All requests, including both reads and writes, to the blocks in the remap area are directed to the SSD. The write-back area is used as a buffer to temporarily hold dirty data of incoming write requests. All other requests are directed to the HDD. Blocks in the write-back area are periodically synchronized to the HDD and recycled for serving incoming writes. We use a configurable quota to guard the sizes of the two regions, so there is no need to physically segment the two regions on the SSD.

We allocate blocks in the SSD in *chunks*, which is similar in nature to that in RAID [25]. This brings two benefits. First, when moving data into the SSD, each write is organized more efficiently in a reasonably large request. Second, it avoids splitting a request into several excessively small requests. In our prototype, we choose an initial chunk size of 8 sectors (4KB). We will further study the effect of chunk size on performance in Section 7.5. In Hystor, all data allocation and management are performed in chunks.

5.4 Managing the Remap Area

The *remap* area is used to maintain identified critical blocks for a long-term optimization. Two types of blocks can be remapped to the SSD: (1) the high-cost data blocks, which are identified by analyzing data access history using the block table, and (2) file system metadata blocks, which are identified through available semantic information in OS kernels.

5.4.1 Identifying High-Cost Data Blocks

As shown in Section 4, the block table maintains data access history in forms of the *counter* values. By comparing the counter values of entries at the BGD, BMD, or BTE levels, we can easily identify the hot regions, sub-regions, and blocks, accordingly. The rationale guiding our design is that the hottest blocks in the hottest regions should be given the highest priority to stay in the high-speed SSD.

Program 1 Pseudocode of identifying candidate blocks.

```
counter():       the counter value of an entry
total_cnt():     the aggregate value of counters
                 of a block table page
sort_unique_asc(): sort entries by unique values
sort_counter_dsc(): sort entries by counter values
quota:           the num. of available SSD blocks

sort_unique_asc(bgd_page); /*sort bgd entries*/
bgd_count = total_cnt(bgd_page);
for each bgd entry && quota > 0; do
  bmd_quota  = quota*counter(bgd)/bgd_count;
  bgd_count -= counter(bgd);
  quota     -= bmd_quota;

  bmd_page = bgd->bmd;    /*get the bmd page*/
  sort_unique_asc(bmd_page);/*sort bmd entries*/
  bmd_count = total_cnt(bmd_page);
  for each bmd entry && bmd_quota > 0; do
    bte_quota  = bmd_quota*counter(bmd)/bmd_count;
    bmd_count -= counter(bmd);
    bmd_quota -= bte_quota;

    bte_page   = bmd->bte;
    sort_counter_dsc(bte_page);
    for each bte entry && bte_quota > 0; do
      add bte to the update list;
      bte_quota --;
    done
    bmd_quota += bte_quota; /*unused quota*/
  done
  quota += bmd_quota; /*unused quota*/
done
```

Program 1 shows the pseudocode of identifying high-cost blocks. We first proportionally allocate SSD space quota to each BGD entry based on their counter values, since a hot region should be given more chance of being improved. Then we begin from the BGD entry with the least number of BTE entries (with the smallest *unique* value), and repeat this process until reaching the BTE level, where we allocate entries in the descending order of their counter values. The blocks being pointed to by the BTE entries are added into a candidate list until the quota is used up. The unused quota is accumulated to the next step. In this way, we recursively determine the hottest blocks in the region and allocate SSD space to the regions correspondingly. Figure 3(b) illustrates this process, and it is repeated until the available space is allocated.

5.4.2 Reorganizing Data Layout across Devices

Workload access pattern changes over time. In order to adapt to the most recent workload access patterns, Hystor periodically wakes up the monitor, updates the block table, and recommends a list of candidate blocks that should be put in SSD, called *updates*, to update the remap area. Directly replacing all the blocks in the SSD with the updates would be over-sensitive to workload dynamics. Thus we take a 'smooth update' approach as follows.

We manage the blocks in the remap area in a list, called the *resident list*. When a block is added to the resident list or accessed, it is put at the top of the list. Periodically the monitor wakes up and sends a list of updates as described in Section 5.4.1 to the data mover. For each update, the data mover checks whether the block is already in the resident list. If true, it informs the monitor that the block is present. Otherwise, it reclaims the block at the bottom of the resident list and reassign its space for the update. In both cases, the new block (update) is placed at the top of the resident list. The monitor repeats this process until a certain number (e.g. 5-10% of the SSD size) of blocks in the resident list are updated. So we identify the high-cost blocks based on the most recent workloads and place them at the top of the resident list, and meanwhile, we always evict unimportant blocks, which have become rarely accessed and thus reside at the list bottom. In this way, we *gradually* merge the most recently identified high-cost data set into the old one and avoid aggressively shifting the whole set from one to another.

Once the resident list is updated, the data mover is triggered to perform I/O operations to relocate blocks across devices asynchronously in the background. Since the data mover can monitor the I/O traffic online and only reorganize data layout during idle periods (e.g. during low-load hours), the possible interference to foreground jobs can be minimized.

5.4.3 User-level Monitor

As a core engine of Hystor, the monitor receives intercepted requests from the remapper, updates the block table, and generates a list of updates to relocate blocks across devices. The monitor can work in either kernel mode or user mode with the same policy. We implemented both in our prototype.

Our user-level monitor functions similarly to *blktrace* [2]. Requests are temporarily maintained in a small log buffer in the kernel memory and periodically passed over to the monitor, a user-level daemon thread. Our prototype integrates the user-level monitor into *blktrace*, which allows us to efficiently use the existing infrastructure to record I/O trace by periodically passing requests to the monitor. The kernel-level monitor directly conducts the same work in the OS kernel.

Compared to the kernel-level monitor, the user-level monitor incurs lower overhead. Memory allocation in the user-level monitor is only needed when it is woken up, and its memory can even be paged out when not in use. Since each time we only update the data structures partially, this significantly reduces the overhead.

5.4.4 Identifying Metadata Blocks

File system metadata blocks are critical to system performance. Before accessing a file, its metadata blocks must be loaded into memory. With only a small amount of SSD space, relocating file system metadata blocks into SSD can effectively improve I/O performance, especially for metadata-intensive workloads during a cold start. Hystor attempts to identify these *semantically critical* blocks and proactively remap them to the SSD to speed up file accesses at an early stage, which avoids high-cost cold misses at a later time. In order to avoid intrusive system changes, we take a conservative approach to leverage the information that is already available in the existing OS kernels.

In the Linux kernel, metadata blocks are tagged such that an I/O scheduler can improve metadata read performance. So far, this mechanism is used by some file systems (e.g. Ext2/Ext3) for metadata reads. In our prototype, we modified a single line at the block

layer to leverage this available information by tagging incoming requests for metadata blocks. No other changes to file systems or applications are needed. Similar tagging technique is also used in Differentiated Storage Services [22]. When the remapper receives a request, we check the incoming request's tags and mark the requested blocks in the block table (using the *flag* field of BTE entries). The identified metadata blocks are remapped to the SSD. Currently, our implementation is effective for Ext2/Ext3, the default file system in Linux. Extending this approach to other file systems needs additional minor changes.

Another optional method, which can identify metadata blocks without any kernel change, is to *statically* infer the property of blocks by examining their logical block numbers (LBN). For example, the Ext2/Ext3 file system segments storage space into 128MB block groups, and the first few blocks of each group are always reserved for storing metadata, such as inode bitmap, etc. Since the location of these blocks is statically determined, we can mark them as metadata blocks. As such a solution assumes certain file systems and default configurations, it has not been adopted in our prototype.

5.5 Managing the Write-back Area

The most recent generation of SSDs has shown an exceptionally good write performance, even for random writes (50-75µs for a 4KB write [5]). This makes the SSD a suitable place for buffering dirty data and reducing latency for write-intensive workloads. As a configurable option, Hystor can leverage the high-speed SSD as a buffer to speed up write-intensive workloads.

The blocks in the write-back area are managed in two lists, a *clean list* and a *dirty list*. When a write request arrives, we first allocate SSD blocks from the *clean list*. The new dirty blocks are written into the SSD and added onto the *dirty list*. We maintain a counter to track the number of dirty blocks in the write-back area. If this number reaches a *high watermark*, a *scrubber* is waken up to write dirty blocks back to the HDD until reaching a *low watermark*. Cleaned blocks are placed onto the clean list for reuse. Since writes can return immediately once the data is written to the SSD, the synchronous write latency observed by foreground jobs is very low. We will examine the scrubbing effect in Section 7.4. Another optional optimization is to only buffer small write requests in the SSD, which further improves the use of the write-back area

As mentioned previously, we do not specifically optimize random writes, since the state-of-the-art SSDs provide high random write performance [5, 6]. One might also be concerned about the potential reliability issues of using SSD as a write-back buffer, since flash memory cells can wear out after a certain number of program/erase cycles. Fortunately, unlike early generations of SSDs, the current high-end SSDs can provide a reasonably high reliability. For example, the Mean Time Before Failure (MTBF) rating of the Intel® X25-E SSDs is as high as 2 million hours [13], which is comparable to that of typical HDDs. In this paper we do not consider the low-end SSDs with poor write performance and low reliability, which are not suitable for our system design goals.

6. IMPLEMENTATION ISSUES

We have prototyped Hystor in the Linux kernel 2.6.25.8 as a stand-alone kernel module with about 2,500 lines of code. The user-level monitor is implemented as a user-level daemon thread with about 2,400 lines of code. Neither one requires any modifications in the Linux kernel. The alternative kernel implementation of the monitor module consists of about 4,800 lines of code and only about 50 lines of code are inserted in the stock Linux kernel.

In our prototype, the remapper is implemented based on the software RAID. When the kernel module is activated, we use `dmsetup` to create a new block device with appointed HDD and SSD devices. By integrating the Hystor functionality on the block layer, we can avoid dealing with some complex issues, such as splitting and merging requests to different devices, since the block layer already handles these issues. The downside of this design is that requests observed at this layer may be further merged into larger requests later, so we track the LBN of the last request to estimate the mergeable request size. Another merit of this design is that Hystor can work seamlessly with other storage, such as RAID and SAN storage. For example, a RAID device can be built upon Hystor virtual devices, similarly, Hystor can utilize RAID devices too. Such flexibility is highly desirable in commercial systems.

As a core engine of Hystor, the monitor can work in either kernel mode or user mode. In both cases, the monitor is implemented as a daemon thread. Periodically it is triggered to process the collected I/O requests, update the block table, and generate updates to drive the data mover to perform data relocation. In kernel mode, the observed requests are held in two log buffers. If one buffer is full, we swap to the other to accept incoming requests, and the requests in the full buffer are updated to the block table in parallel. In user mode, requests are directly passed to the user-level daemon. The analysis of data access history can also be done offline. In our current prototype, we maintain the block table and the mapping table full in memory, and in our future work we plan to further optimize memory usage by only loading partial tables into memory as discussed previously.

7. EVALUATION

7.1 Experimental System

Our experimental system is an Intel® D975BX system with a 2.66GHz Intel® Core™ 2 Quad CPU and 4GB main memory on board. Our prototype system consists of a Seagate® Cheetah® 15k.5 SAS hard drive and a 32GB Intel® X25-E SSD, both of which are high-end storage devices on the market. Also note that we only use *partial* SSD space in our experiments to avoid overestimating the performance. Table 1 lists the detailed specification of the two devices. The HDDs are connected through an LSI® MegaRaid® 8704 SAS card and the SSD uses a SATA 3.0Gb/s connector.

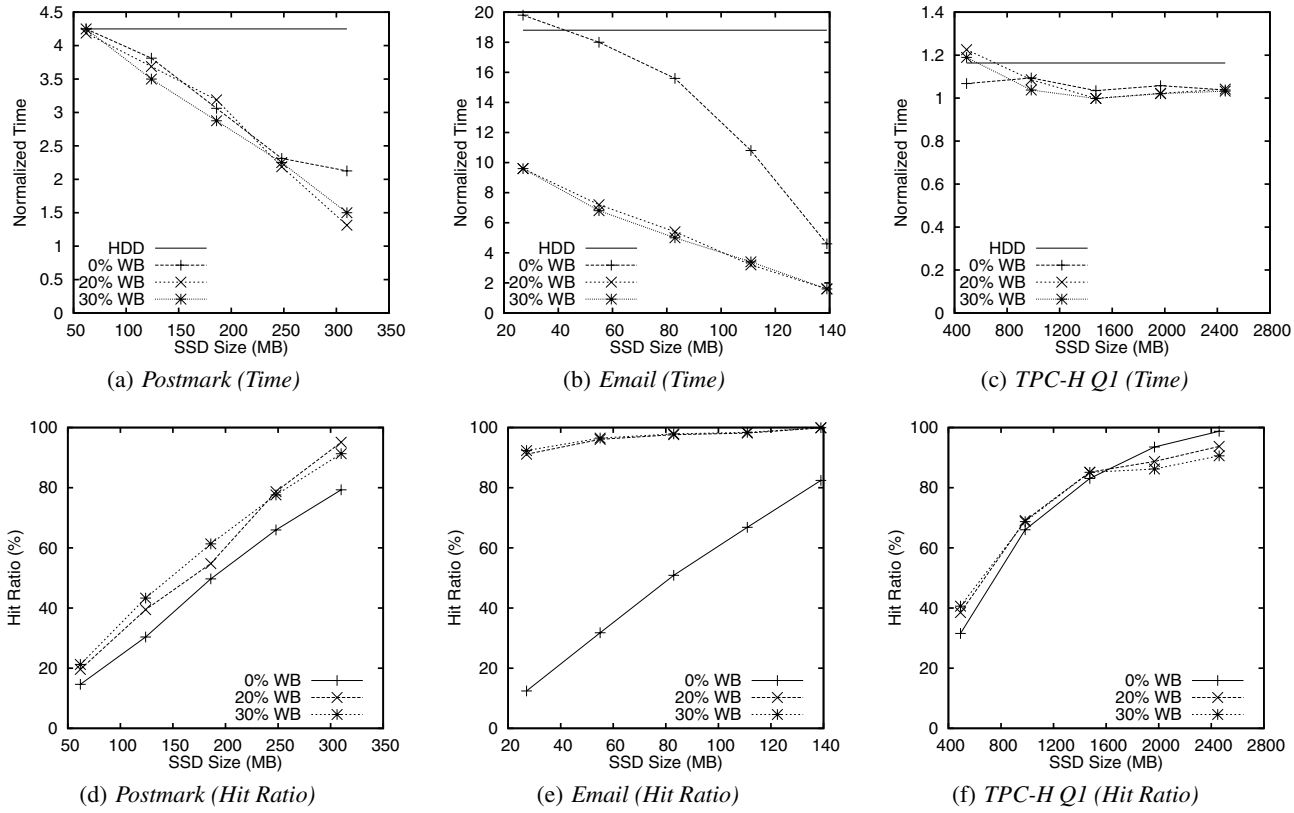
	X25-E SSD	Cheetah HDD
Capacity	32GB	73GB
Interface	SATA2	SAS
Read Bandwidth	250 MB/Sec	125 MB/Sec
Write Bandwidth	180 MB/Sec	125 MB/Sec

Table 1: Specifications of the SSD and the HDD.

We use Fedora™ Core 8 with the Linux kernel 2.6.25.8 and Ext3 file system with default configurations. In order to minimize the interference, the operating system and home directory are stored in a separate hard disk drive. We use the *noop* (No-op) I/O scheduler, which is suitable for non-HDD devices [5,6], for the SSD. The hard disk drives use the *CFQ* (Completely Fair Queuing) scheduler, the default scheduler in the Linux kernel, to optimize the HDD performance. The on-device caches of all the storage devices are enabled. The other system configurations use the default values.

7.2 Performance of Hystor

In general, the larger the SSD size is, the better the Hystor's performance is. In order to avoid overestimating the performance improvement, we first estimate the working-set size (the number of blocks being accessed during execution) of each workload by



(a) *Postmark (Time)* (b) *Email (Time)* (c) *TPC-H Q1 (Time)*

(d) *Postmark (Hit Ratio)* (e) *Email (Hit Ratio)* (f) *TPC-H Q1 (Hit Ratio)*

Figure 5: Normalized execution times and hit ratios of workloads. The horizontal line represents the time of running on HDD.

examining the collected I/O traces off line, then we conduct experiments with five configurations of the SSD size, namely 20%, 40%, 60%, 80%, and 100% of the working-set size. Since we only conservatively use *partial* SSD space in our experiments, the performance of Hystor can be even better in practice. To examine the effectiveness of the write-back area, we configure three write-back area sizes, namely 0%, 20%, and 30% of the available SSD space.

For each workload, we perform the baseline experiments on the SSD. We rerun the experiments with various configurations of the SSD space. We show the execution times normalized to that of running on the SSD-only system. Therefore, the normalized execution time '1.0' represents the ideal case. In order to compare with the worst case, running on the HDD-only system, we plot a horizontal line in the figures to denote the case of running on the HDD. Besides the normalized execution times, we also present the hit ratios of I/O requests observed at the remapper. A request to blocks resident in the SSD is considered a *hit*, otherwise, it is a *miss*. The hit ratio describes what percentage of requests are serviced from the SSD. Due to space constraints, we only present results for the user-mode monitor here, and the kernel monitor shows similar results. Figure 5 shows the normalized execution times and hit ratios.

7.2.1 Postmark

Postmark is a widely used file system benchmark [28]. It creates 100 directories and 20,000 files, then performs 100,000 transactions (reads and writes) to stress the file system, and finally deletes files. This workload features intensive small random data accesses.

Figure 5(a) shows that as the worst case, postmark on the HDD-only system runs 4.2 times slower than on the SSD-only system. Hystor effectively improves performance for this workload. With the increase of SSD space, the execution time is reduced till close to an SSD-only system, shown as a linear curve in the figure. In this case, most data blocks are accessed with similar patterns, which is challenging for Hystor to identify high-cost data blocks based on access patterns. However, Hystor still provides performance gains proportional to available SSD space.

Since this workload features many small writes, allocating a large write-back area helps improve hit ratios as well as execution times. Figure 5(d) shows that with the SSD size of 310MB, allocating 30% of the SSD space for write-back can improve hit ratio from 79% to 91%, compared to without write-back area. Accordingly, the execution time is reduced from 34 seconds to 24 seconds, which is a 29% reduction.

Also note that multiple writes to the same block would cause synchronization issues. With a smaller write-back area, dirty blocks have to be more frequently flushed back to the HDD due to capacity limit. When such an operation is in progress, incoming write requests to the same blocks have to be suspended to maintain consistency, which further artificially increases the request latency. For example, when the cache size grows to 310MB, the amount of hits to the lock-protected blocks decreases by a factor of 4, which translates into a decrease of execution time.

7.2.2 Email

Email was developed by University of Michigan based on Postmark for emulating an email server [31]. It is configured with 500 directories, 500 files, and 5,000 transactions. This workload has intensive synchronous writes with different append sizes and locations based on realistic mail distribution function, and it features a more skewed distribution of latencies. Most data accesses are small random writes, which are significantly faster on the SSD.

Figure 5(b) shows that running *email* exclusively on the SSD is 18.8 times faster than on the HDD. With no write-back area, the performance of Hystor is suboptimal, especially for a small

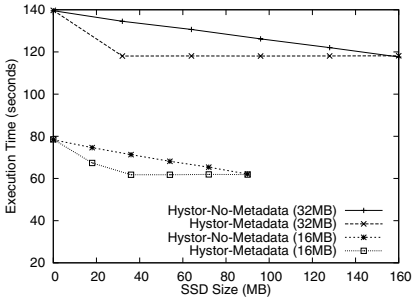


Figure 6: Optimization for metadata blocks. 32MB and 16MB refer to the two workloads. Hystor with and without optimizing metadata are referred to as *Hystor-Metadata* and *Hystor-No-Metadata*.

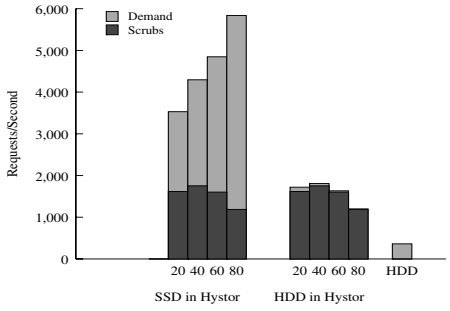


Figure 7: Request arrival rate in *email*. The numbers on X axis for each bar refer to various configurations of the SSD size (% of the working-set size). HDD refers to the HDD-only system.

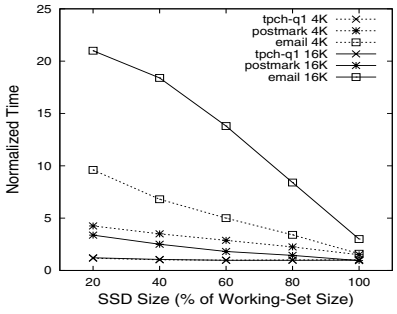


Figure 8: Effect of chunk size on performance. 4K and 16K refer to the chunk sizes, respectively. The numbers on X axis refer to various configurations of the SSD size (% of the working-set size).

SSD size. As we see in the figure, with SSD size of 27MB (20% of the working-set size), Hystor may even be slightly worse than the HDD-only system, due to additional I/O operations and increased probability of split requests. Without the write-back area, data blocks remapped in the SSD may not be necessarily to be reaccessed in the next run. This leads to a hit ratio of only 12.4% as shown in Figure 5(e). In contrast, with the write-back area, the hit ratio quickly increases to over 90%. This shows that the write-back area also behaves like a small cache to capture some short-term data reuse. As a result, the execution time is reduced by a half.

7.2.3 TPC-H Query 1

TPC-H Q1 is the query 1 from the TPC-H database benchmark suite [33]. It runs against a database (scale factor 1) managed by PostgreSQL 8.1.4 database server. Different from the other workloads, this workload does not benefit much from running on the SSD, since its data accesses are more sequential and less I/O intensive. As shown in Figure 5(c), running this workload on the HDD-only system is only 16% slower than running on the SSD. However, since the reuse of data blocks is more significant, the hit ratio in this case is higher. Figure 5(f) shows that with SSD size of 492MB, the hit ratio of incoming requests is about 30% to 40%. When the SSD size is small, the write-back area may introduce extra traffic, which leads to a 2-5% slowdown compared to running on HDD. As the write-back area size increases, the number of write-back operations is reduced dramatically, which improves the I/O performance.

7.3 Metadata Blocks

Hystor also identifies metadata blocks of file systems and remaps them to the SSD. We have designed an experiment to show how such an optimization improves performance.

In Ext2/Ext3 file systems, a large file is composed of many data chunks, and *indirect blocks* are used to locate and link these chunks together. As a type of metadata, indirect blocks do not contain file content but are crucial to accessing files. We create a 32GB file and use the Intel® Open Storage Toolkit [21] to generate two workloads, which randomly read 4KB data each time until 16MB and 32MB of data are read. This workload emulates data accesses in files with complex internal structures, such as virtual disk file used by virtual machines. In such random workloads, the accessed file data are unlikely to be reused, while indirect blocks would be reaccessed, thus holding metadata blocks in the SSD would be beneficial. We use this example to compare the performance of Hystor with and without optimization for file system metadata blocks, denoted as *Hystor-Metadata* and *Hystor-No-Metadata* respectively.

Figure 6 shows the experimental results of Hystor-No-Metadata and Hystor-Metadata. Both approaches eventually can speed up the two workloads by about 20 seconds. However, Hystor-Metadata is able to achieve that performance with a much smaller SSD space. For the workload reading 32MB data, Hystor-Metadata identifies and remaps nearly all indirect blocks to the SSD with just 32MB of SSD space. In contrast, Hystor-No-Metadata lacks the capability of identifying metadata blocks. Since only around 20% of the blocks being accessed are metadata blocks, most blocks remapped to the SSD are file content data blocks, which are unfortunately almost never reused. Therefore Hystor-No-Metadata requires about 160MB of SSD space to cover the whole working-set, while Hystor-Metadata needs only 32MB SSD space. A similar pattern can be observed in the case of reading 16MB data.

This experiment shows that optimization for metadata blocks can effectively improve system performance with only a small amount of SSD space, especially for metadata-intensive workloads. More importantly, different from identifying high-cost data blocks by observing workload access patterns, we can proactively identify these *semantically critical* blocks at an early stage, so high-cost cold misses can be avoided. It is also worth noting that the three major components in Hystor are complementary to each other. For example, although the current implementation of Hystor identifies metadata blocks only for read requests, writes to these metadata blocks still can benefit from being buffered in the write-back area.

7.4 Scrubbing

Dirty blocks buffered in the write-back area have to be written back to the HDD in the background, called *scrubbing*. Each scrub operation can cause two additional I/O operations – a read from the SSD and a write to the HDD. Here we use *email*, the worst case for scrubs, to study how scrubbing affects system performance. Figure 7 shows the request arrival rate (number of requests per second) for *email* configured with four SSD sizes (20-80% of the working-set size) and a 20% write-back fraction . The requests are broken down by the source, internal scrubbing daemon or the upper-layer components, denoted as *scrubs* and *demand* in the figure, respectively.

As shown in Figure 7, the request arrival rate in Hystor is much higher than that in the HDD-only system. This is due to two reasons. First, the average request size for HDD-only system is 2.5 times larger than that in Hystor, since a large request in Hystor may split into several small ones to different devices. Second, two additional I/O operations are needed for each scrub. We can see that, as the SSD size increases to 80% of the working-set size, the arrival rate of scrub requests drops by nearly 25% on the SSD, due to less frequent scrubbing. The arrival rate of on-demand requests in-

creases as the SSD size increases, because the execution time is reduced and the number of on-demand requests remains unchanged.

An increase of request arrival rate may not necessarily lead to an increase of latency. In the case with 80% of the working-set size, as many as 5,800 requests arrive on the SSD every second. However, we do not observe a corresponding increase of execution time (see Figure 5(b)) and the SSD I/O queue still remains very short. This is mainly because the high bandwidth of the SSD (up to 250MB/sec) can easily absorb the extra traffic. On the HDD in Hystor, the request arrival rate reaches over 1,800 requests per second. However, since these requests happen in the background, the performance impact on the foreground jobs is minimal.

This case shows that although a considerable increase of request arrival rate is resident on both storage devices, conducting background scrubbing causes minimal performance impact, even for write-intensive workloads.

7.5 Chunk Size

Chunk size is an important parameter in Hystor. A large chunk size is desirable for reducing memory overhead of the mapping table and the block table. On the other hand, a small chunk size can effectively improve utilization of the SSD space, since a large chunk may contain both hot and cold data.

Figure 8 compares performance of using a chunk size of 8 sectors (4 KB) and 32 sectors (16 KB). We only present data for the cache with a 20% write-back fraction here. We can see that with a large chunk size (16KB), the performance of *email* degrades significantly due to the underutilized SSD space. Recall that most of the requests in *email* are small, hot and cold data could co-exist in a large chunk, which causes the miss rate to increase by four-fold. With the increase of SSD size, such a performance gap is reduced, but it is still much worse than using 4KB chunks. The other workloads are less sensitive to chunk size.

This experiment shows that choosing a proper chunk size should consider the SSD size. For a small-capacity SSD, a small chunk size should be used to avoid wasting precious SSD space. A large SSD can use a large chunk size and afford the luxury of increased internal fragmentation in order to reduce overhead. In general, a small chunk size (e.g. 4KB) is normally sufficient for optimizing performance. Our prototype uses a chunk size of 4KB in default.

8. RELATED WORK

Flash memory and SSDs have been actively studied recently. There is a large body of research work on SSDs (e.g. [1, 5–8]). A survey [9] summarizes the key techniques in flash memory based SSDs. Here we present the work most related to this paper.

The first set of work is generally cache-based solutions. An early work [19] uses flash memory as a secondary-level file system buffer cache to reduce power consumption and access latencies for mobile computers. SmartSaver [4] uses a small-factor flash drive to cache and prefetch data for saving disk energy. A hybrid file system, called Conquest [34], merges the persistent RAM storage into the HDD-based storage system. Conquest caches small files, metadata, executables, shared libraries into the RAM storage and it demands a substantial change to file system designs. AutoRAID [35] migrates data inside the HDD-based RAID storage to improve performance and cost-efficiency based on patterns. Sun® Solaris™ [18] can set a high-speed device as a secondary-level buffer cache between main memory and hard disk drives. Microsoft® Windows® ReadyBoost [23] takes a similar approach to use a flash device as an extension of main memory. Intel® TurboMemory [20] uses a small amount of flash memory as a cache to buffer disk data and uses a threshold size to filter large requests. Kgil et al. [16] propose to use flash memory and DRAM as a disk cache and adopt an LRU-based wear-level aware replacement policy [16]. SieveStore [29] uses a selective caching approach by tracking the access counts and caching the most popular blocks in solid state storage. Hystor views and places SSDs in the storage hierarchy in another way – the high-speed SSD is used as a part of storage rather than an additional caching tier. As such, Hystor only reorganizes data layout across devices periodically and asynchronously, rather than make caching decision on each data access. In addition, recognizing the non-uniform performance gains on SSDs, Hystor not only adopts frequency, rather than recency that has been commonly used in LRU-based caching policies, to better describe the temporal locality, and it also further differentiates various workload access patterns and attempts to make the best use of the SSD space with minimized system changes.

Some other prior work proposes to integrate SSD and HDD together and form a hybrid storage system. Differentiated Storage Services [22] attempts to classify I/O requests and passes information to storage systems for QoS purposes. The upper-level components (e.g. file systems) classify the blocks and the storage system enforces the policy by assigning blocks to different devices. ComboDrive [26] concatenates SSD and HDD into one single address space, and certain selected data and files can be moved into the faster SSD space. As a block-level solution, Hystor hides details from the upper-level components and does not require any modification to applications. Considering the disparity of handling reads and writes in SSDs, Koltsidas and Viglas propose to organize SSD and HDD together and place read-intensive data in SSD and write-intensive data in HDD for performance optimization [17]. Soundararajan et al. propose a solution to utilize HDD as a log buffer to reduce writes and improve the longevity of SSDs [32]. Recently, I-CASH [30] has been proposed to use SSD to store seldom-changed reference data blocks and HDD to store a log of deltas, so that random write traffic to SSD can be reduced. Our experimental studies show that the state-of-the-art SSDs have exceptionally high write performance. Specifically optimizing write performance for SSDs can yield limited benefits on these advanced hardware. In fact, Hystor attempts to leverage the high write performance of SSDs and our experimental results show that such a practice can effectively speed up write-intensive workloads.

9. CONCLUSION

Compared with DRAM and HDD, the cost and performance of SSDs are nicely placed in between. We need to find the fittest position of SSDs in the existing systems to strike a right balance between performance and cost. In this study, through comprehensive experiments and analysis, we show that we can identify the data that are best suitable to be held in SSD by using a simple yet effective metric, and such information can be efficiently maintained in the block table at a low cost. We also show that SSDs should play a major role in the storage hierarchy by adaptively and timely retaining performance- and semantically-critical data, and it can also be effective as a write-back buffer for incoming write requests. By best fitting the SSD into the storage hierarchy and forming a hybrid storage system with HDDs, our hybrid storage prototype, Hystor, can effectively leverage the performance merits of SSDs with minimized system changes. We believe that Hystor lays out a system framework for high-performance storage systems.

10. ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive comments. We also thank our colleagues at Intel® Labs, espe-

cially Scott Hahn and Michael Mesnier, for their help and support through this work. We also would like to thank Xiaoning Ding at Intel® Labs Pittsburgh, Rubao Lee at the Ohio State University, and Shuang Liang at EMC® DataDomain for our interesting discussions. This work was partially supported by the National Science Foundation (NSF) under grants CCF-0620152, CCF-072380, and CCF-0913150.

11. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of USENIX'08*, Boston, MA, June 2008.

[2] Blktrace. http://linux.die.net/man/8/blktrace.

[3] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of ASPLOS'09*, Washington, D.C., March 2009.

[4] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proceedings of ISLPED'06*, Tegernsee, Germany, Oct. 2006.

[5] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of SIGMETRICS/Performance'09*, Seattle, WA, June 2009.

[6] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of HPCA'11*, San Antonio, Texas, Feb 12-16 2011.

[7] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of FAST'11*, San Jose, CA, Feb 15-17 2011.

[8] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proceedings of ISCA'09*, Austin, TX, June 2009.

[9] E. Gal and S. Toledo. Algorithms and data structures for flash memories. In *ACM Computing Survey'05*, volume 37(2), pages 138–163, 2005.

[10] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of ASPLOS'09*, Washington, D.C., March 2009.

[11] J. Handy. Flash memory vs. hard disk drives - which will win? http://www.storagesearch.com/semico-art1.html.

[12] L. Huang and T. Chieuh. Experiences in building a software-based SATF scheduler. In *Tech. Rep. ECSL-TR81*, 2001.

[13] Intel. Intel X25-E extreme SATA solid-state drive. http://www.intel.com/design/flash/nand/extreme, 2008.

[14] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of USENIX'05*, Anaheim, CA, April 2005.

[15] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proceedings of FAST'05*, San Francisco, CA, December 2005.

[16] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *Proceedings of ISCA'08*, Beijing, China, June 2008.

[17] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. In *Proceedings of VLDB'08*, Auckland, New Zealand, August 2008.

[18] A. Leventhal. Flash storage memory. In *Communications of the ACM*, volume 51(7), pages 47–51, July 2008.

[19] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Science*, Wailea, HI, Jan 1994.

[20] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. In *ACM Transactions on Storage*, volume 4, May 2008.

[21] M. P. Mesnier. Intel open storage toolkit. http://www.sourceforge.org/projects/intel-iscsi.

[22] M. P. Mesnier and J. B. Akers. Differentiated storage services. *SIGOPS Oper. Syst. Rev.*, 45:45–53, February 2011.

[23] Microsoft. Microsoft Windows Readyboost. http://www.microsoft.com/windows/windows-vista/features/readyboost.aspx, 2008.

[24] A. Patrizio. UCSD plans first flash-based supercomputer. http://www.internetnews.com/hardware/article.php/3847456, November 2009.

[25] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of SIGMOD'88*, Chicago, IL, June 1988.

[26] H. Payer, M. A. Sanvido, Z. Z. Bandic, and C. M. Kirsch. Combo Drive: Optimizing cost and performance in a heterogeneous storage device. In *Proceedings of the 1st Workshop on integrating solid-state memory into the storage hierarchy (WISH'09)*, 2009.

[27] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, portable I/O scheduling with the disk mimic. In *Proceedings of USENIX'03*, San Antonio, TX, June 2003.

[28] Postmark. A new file system benchmark. http://www.netapp.com/tech_library/3022.html, 1997.

[29] T. Pritchett and M. Thottethodi. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of ISCA'10*, Saint-Malo, France, June 2010.

[30] J. Ren and Q. Yang. I-CASH: Intelligently coupled array of ssd and hdd. In *Proceedings of HPCA'11*, San Antonio, Texas, Feb 2011.

[31] S. Shah and B. D. Noble. A study of e-mail patterns. In *Software Practice and Experience*, volume 37(14), 2007.

[32] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of FAST'10*, San Jose, CA, February 2010.

[33] Transaction Processing Performance Council. TPC Benchmark H. http://www.tpc. org/tpch, 2008.

[34] A. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a Disk/Persistent-RAM hybrid file system. In *Proceedings of the USENIX'02*, Monterey, CA, June 2002.

[35] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *ACM Tran. on Computer Systems*, volume 14, pages 108–136, Feb 1996.