Noname manuscript No.
(will be inserted by the editor)

Operating System Level Data Tiering Using Online Workload Characterization

Reza Salkhordeh · Hossein Asadi · Shahriar Ebrahimi

Received: date / Accepted: date

Abstract Over the past decade, storage has been the performance bottleneck in I/O intensive programs such as online transaction processing applications. To alleviate this bottleneck with minimal cost penalty, cost-effective design of a high-performance disk subsystem is of decisive importance in enterprise applications. Data tiering is an efficient way to optimize cost, performance, and reliability in storage servers. With the promising advantages of *Solid State Drives* (SSDs) over *Hard Disk Drives* (HDDs) such as lower power consumption and higher performance, traditional data tiering techniques should be revisited in order to use SSDs in a more efficient way. Previously proposed tiering solutions have attempted to enhance performance based on different parameters such as request size or randomness. These solutions, however, are mostly optimized towards one type of I/O workloads and are not applicable to workloads with different characteristics. This paper presents an online data tiering technique at the Operating System level with a linear weighted formulation to enhance I/O performance with minimal cost overhead. The proposed technique characterizes the workload access pattern with respect to metadata versus user data, frequency of accesses, random versus sequential accesses, and read versus write accesses. To evaluate the proposed technique, it is implemented on a Linux 3.1.4 equipped with ext2 filesystem. The experimental results over I/O intensive workloads show that the proposed technique improves performance up to 30% as compared to the previous techniques while imposing negligible memory overhead to the system.

Keywords Operating system · Solid state drive · Filesystem · Performance · I/O workload characterization

Reza Salkhordeh, Hossein Asadi, and Shahriar Ebrahimi
Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.
E-mail: {salkhordeh, shebrahimi}@ce.sharif.edu, asadi@sharif.edu

1 Introduction

Storage devices are orders of magnitude slower than main memory and are considered as a performance bottleneck in I/O intensive workloads in enterprise applications such as web serving and *On-Line Transaction Processing* (OLTP). The performance gap between main memory and storage devices has increased over time since mechanical components in *Hard Disk Drives* (HDDs) have very slight performance improvement as opposed to electronical components. Based on Amdahl's Law [2], improving the performance of storage subsystem in I/O intensive applications can significantly improve the overall system performance.

To alleviate the performance gap, system designers have recently started to replace slow running HDDs with *Solid-State Drives* (SSDs). SSDs, typically based on NAND flash technology, do not have any mechanical components. NAND flash cells cannot be overwritten and need to be erased before any update. An erase operation is orders of magnitude more time-consuming than a read operation. In addition, it is also at least two times more time-consuming than a write operation. Moreover, it directly affects flash lifetime [7, 33]. In contrast to HDDs, writing in a NAND flash cell takes more time than a read operation. Therefore, write operations are more expensive than read operations in terms of response time and reliability [15]. Another interesting attribute of NAND flash cells is that they have almost constant performance in random and sequential read intensive workloads. Early SSDs were very expensive with low performance in write-intensive workloads. Hence, many studies have tried to address this limitation by using buffers at the *Operating System* (OS) level [12, 29]. In recent years, with improvements in flash technology and SSD controllers, the performance of SSDs has significantly improved in a variety of workloads. SSDs are still more expensive in terms of $/GB. Hence, it is not economically justified to replace all HDDs with SSDs in storage subsystems.

A commonly used technique to enhance performance in storage subsystems is *Redundant Array of Independent Disks* (RAID). Replacing a subset of HDDs with SSDs in a RAID array will not, however, enhance the performance since workload is distributed among all disks and HDDs still will be the performance bottleneck. Another approach to utilize SSDs in storage subsystems is to use them as a caching layer between the main memory and HDDs. Caching techniques are effective when the workload exhibits either temporal or spatial locality. Additionally, the SSD used as a caching layer cannot be used as a permanent storage device and will be a cost overhead to the system.

An alternate technique to improve the performance of disk subsystems is using data tiering. This technique relies on the fact that not all data blocks have equal performance or reliability requirements. Hence, one can create many tiers, each with various performance or reliability levels and place data blocks on tiers based on application requirements. Providing different tiers from small to large capacity and from inexpensive to costly tiers will ensure to find an optimal tier for each data block. Due to cost-effective performance advantage of tiering techniques, enterprise storage vendors such as IBM^{TM} and EMC^{TM}

offer data tiering in their storage solutions to provide improved performance with minimal cost increase [24, 10].

Due to the emergence of SSDs, data tiering requires revisiting data selection algorithms and adapting them according to the characteristics of NAND flashes. Traditional data tiering solutions were designed for homogeneous storage devices that only differ in cost and performance. These solutions, however, are not effective for a heterogeneous storage subsystem which uses both SSDs and HDDs. There are few tiering techniques proposed to enhance performance in heterogeneous storage subsystems [20, 11, 13, 21, 17, 16, 26]. The tiering technique presented in [20] uses file granularity which is only efficient for small size files. There are other tiering techniques with block granularity that place more frequently and random accessed blocks on the SSD tier [11, 17]. Some other techniques place filesystem metadata on the SSD tier [13, 21] to improve the overall performance. Although previous studies have tried to consider many parameters, none of them have used both OS-level semantic information and workload characterization simultaneously. In addition, previous techniques are mainly suitable for a limited number of workload types since they consider either a few aspects of workloads or a predefined part of SSD for one access type. For example, [8] dedicates a section of SSD to metadata requests and another section to write-back requests. In data intensive and read-dominated workloads, these sections will be underutilized with almost no performance improvement. To the best of our knowledge, there has been no comprehensive online tiering technique taking into account the characteristics of both storage devices and workload patterns. In addition, none of previous studies have proposed a weighted formulation based on workload, types of storage devices, and OS semantic information.

In this paper, we propose an efficient OS-level data tiering technique using online workload characterization. The proposed data tiering technique, called *Online OS-level Data Tiering* (OODT), places data blocks on the most suitable tier with respect to frequency of accesses, data types (metadata vs. user data), access pattern (random vs. sequential access), and the frequency of read or write operations. In the proposed technique, address space is split into fixed-size data blocks, unlike studies such as [4] that use dynamic data blocks size. This is due to the overhead of managing and migrating data blocks with dynamic size is significantly higher than the overhead of fixed-size data blocks. Next, the access pattern to each data block is characterized with respect to target parameters. OODT uses a weighted priority function to characterize and identify the most effective chunks for each tier. The priority function is a linear weighted formulation based on randomness, read ratio, and request type (data or metadata). Data blocks are placed on tiers based on disk characteristics and the priority of each data block. In the proposed tiering technique, hot and cold data blocks are dynamically tracked in each tier to identify chunks that need to be migrated to another tier when workload access pattern changes over time. This data migration occurs online with minimal impact on performance.

The proposed data tiering technique has been implemented as a block device in Linux 3.1.4 equipped with ext2 filesystem and encapsulated in a

loadable module that can be easily added to a running operating system. To evaluate the efficiency of OODT, several synthetic and commercial I/O intensive workloads have been used to stress the proposed tiering technique. Experimental results demonstrate that the proposed technique can improve performance up to 72% in metadata intensive workloads, and up to 43% in database random workloads as compared to pure HDD-based disk subsystems while placing less than 4% of the total data blocks on the high-performance tier. Additionally, OODT improves performance up to 30% over the conventional tiering techniques.

The rest of this paper is organized as follows. Section 2 provides a review or related work. Section 3 presents the proposed technique, with experimental results documented in Section 4. Section 5 discusses limitations to and possible extensions of OODT. Section 6 offers final thoughts and conclusions.

2 Related Work

Previous work can broadly be classified into two groups: a) SSDs used as a caching layer between main memory and storage devices and b) SSDs employed as a tier in a data tiering storage subsystem.

2.1 Caching

The caching layer endeavours to decrease response time by temporarily storing user data in the cache device in the following manner. In the case of write requests, user data is temporarily written into the cache device and the write acknowledge is sent out to the upper layer. The user data is written to the permanent storage device later on. In case of read requests, the requested data is transferred to the cache device for future references. If a cached user data is never referenced again, performance will not be improved. Due to high $/GB, designers use SSDs to store only performance critical and hot data blocks. Early SSDs had low random write performance. To address this issue, [34] suggested incorporating log-based write cache in a HDD device to decrease the number of writes issued to SSDs. Recent SSDs demonstrate an improved performance on random writes and the state-of-the art cache designs consider using SSDs as a faster device for all workloads and request types [18, 8].

To improve storage performance, [27] utilized a SSD cache between the main memory and HDDs. This technique selects data blocks for caching on the SSD based on their access pattern. For each data block, the number of random reads, random writes, sequential reads, and sequential writes will be stored and based on these parameters, blocks will be chosen to enter into the SSD cache.

The caching technique proposed in [18], called Azor, uses a hash function to allocate a cache line for each page. A page is placed in the cache if it has higher priority than the existing page in the cache line. Filesystem metadata

pages have the highest priority. Pages with the highest access frequency receive the next highest priority. In this technique, cold metadata pages have higher priority than hot data pages. Thus, the cache will be filled with cold metadata pages after a while and the cache hit ratio will be decreased significantly.

Another caching technique, called Hystor, has been proposed in [8]. This technique splits a SSD into three regions. The first region stores filesystem metadata pages similar to Azor. The second region is a write-back cache to speed up write requests. Lastly, the third region acts similar to a typical cache memory and it is used to cache read requests. Hystor maintains a three-level table for recording more frequent accesses to data pages. In predefined intervals, a data mover function is activated to move pages with the highest access frequency to SSDs.

Database management systems can also benefit from SSD cache since database requests mostly exhibit random pattern. Many research efforts attempted to classify database request types and cache important pages, based on these classifications [25, 23]. However, further discussion of such techniques is beyond the scope of this work.

2.2 Tiering

Data tiering techniques use a tier of SSDs for performance critical blocks and a tier of HDDs to store cold blocks. As opposed to caching techniques, there is no temporary device to store user data. Instead, to speed up more frequent accessed data blocks, a high-performance tier is utilized. Similar to early caching solutions, initial tiering techniques have used SSDs as a storage device with write performance worse than HDDs [19, 37]. In [11], an extent-based tiering technique has been proposed. In this technique, workloads are first characterized and then the performance requirement for each extent is calculated. Lastly, considering allocation of an extent to any arbitrary tier, the fraction of available throughput of the tier which will be occupied by this extent will be computed. If an extent does not have high performance requirements, this technique considers the fraction of tiers space that is occupied by the extent, rather than considering the fraction of tiers bandwidth occupied. This will prevent cold extents to consume high-performance tiers. Based on the calculated fractions for extents, each extent is assigned to a tier which has the minimum fraction. This approach endeavours to decrease the power consumption and maintain almost the same level of performance. In contrast, the technique proposed in this research effort aims at improving performance. In addition, this work does not consider the intrinsic characteristics of SSDs and treats them just as faster HDDs.

A tiering technique that splits logical block addresses into 1MB extents has been proposed in [17]. The proposed tiering technique, called Hybrid-Store, records access frequency and request size for extents. Based on such information, extents are classified into one of 32 predefined classes. Based on this classification and based on tiers characteristics such as *I/O Per Second*

(IOPS) and capacity, HybridStore solves an *Integer Linear Program* (ILP) to find an effective tier for each extent. This technique is an offline approach which vastly differs from online approaches, including our proposed technique.

There are also few tiering techniques that place files on tiers based on file granularity, file access frequency, and the randomness of requests [20, 3]. File granularity is not suitable if large-size files with random access exists in the filesystem, since placing these files on high capacity tiers will degrade performance. This is because HDDs do not provide high performance on random requests. On the other hand, these files are too big to be completely placed on high performance tiers, since such tiers have low capacity. Additionally, in large-size files such as database applications, only a small portion of files will be accessed and majority of data blocks are cold blocks that will occupy the valuable space of high performance tiers without any performance gain.

Filesystem metadata requests comprise between 50% and 70% of the total requests in many I/O intensive workloads [30]. Additionally, the number of files in filesystems increases over time. This will further increase the number of metadata requests for storing and retrieving files [1]. Many tiering techniques only split metadata and data blocks and place metadata pages on the high-performance storage devices [21, 16, 26]. These tiering techniques are not efficient in all workloads and many workloads with low metadata requests will not benefit from these tiering techniques [13].

Lastly, a recent study presented in [3] compared the performance of caching and tiering techniques. This study revealed that caching is more efficient if a workload consists of requests with high locality (e.g., workloads with Zipf distribution). This study concludes that data tiering techniques can outperform caching techniques in workloads with less data locality.

3 Proposed Tiering Architecture: OODT

The main aim of the proposed architecture is to find the optimal tier for each data block using an online workload characterization. The proposed architecture tries to migrate data blocks between high-performance tiers and inexpensive tiers when workload pattern changes. This technique quantifies workload characterizations into a priority score for each data block and uses these scores to identify the most effective blocks for high-performance as well as inexpensive tiers. To this aim, block-level tiering is used as it is not dependent on a particular filesystem and does not suffer from shortcomings of file-based tiering techniques in large-size files. OODT uses *Fine-Grained* (FG) granularity, e.g., 4KB data blocks, as allocation and migration units. By using FG granularity, data blocks that are identified to be moved to high-performance tiers will compose of less cold data as compared to *Coarse-Grained* (CG) granularity migration scheme. The FG migration scheme, however, suffers from memory overhead. This limitation will be discussed in Sec. 5.1.

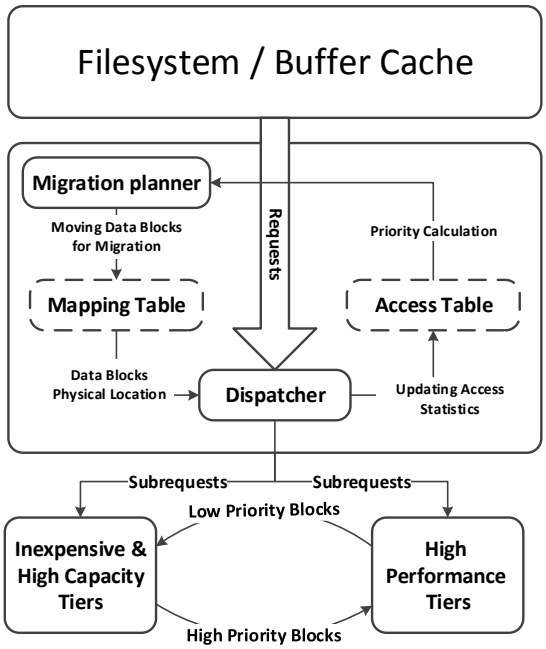


Fig. 1: OODT architecture

3.1 Components and Algorithms Used in Proposed Architecture

The main components of the proposed tiering architecture are shown in Fig. 1 which demonstrates that the proposed architecture consists of five main components: dispatcher, mapping table, access table, migration planner, and storage tiers. In this figure, boxes with dashed lines are data structures and boxes with solid lines are code components. In the proposed architecture, the dispatcher receives requests from either the filesystem or buffer cache layer. If a request is larger than the size of one data block, it will be split into several subrequests, since the corresponding data blocks might exist in different tiers. Each subrequest is intended to read or write one data block. Upon completion of servicing all subrequests, the service to the original request will be accomplished. The dispatcher uses a mapping table to find the physical location of data blocks and redirects subrequests to the corresponding tier. The mapping table stores the physical location of each extent consisting of tier number and the physical location of the extent in that tier. Another table, called access table, is used to maintain all statistics required to determine the priority of extents. This table is updated by the dispatcher based on the characterization of the request.

Algorithm 1 depicts the overall flow from the request arrival to the kernel module to dispatching subrequests to the disks. Line 1 through line 5 examines the subrequests and creates the required bio structures. *endio* function

ALGORITHM 1: Data Tiering Algorithm

```
1  Split request into subrequests;
2  foreach subrequest do Allocate a bio for subrequest;
3  Copy flags from the request into bio;
4  Set the endio function of the bio;
5  Fetch corresponding block for subrequest in lookup table;
6  if block is locked then
7      Wait until unlocked;
8  end
9  Send bio to the corresponding tier;
10 if subrequest can be merged with an item of the access queue then
11     Update the corresponding access queue member;
12 else
13     Remove last request from the access queue;
14     Calculate random priority for last request;
15     Add random priority to priority of the last request;
16     if last request is read then
17         Add read priority to the priority of the last request;
18     end
19     if last request is metadata then
20         Add metadata priority to the priority of the last request;
21     end
22     Add subrequest to the access queue;
23 end
24 requests after migration + + ;
25 if requests after migration > migration-time threshold then
26     Initiate migration function ;
27 end
28 ;
```

will be called upon completing the request for notifying upper layers. Line 6 determines if this block is locked. The migration planner locks the blocks when it attempts to migrate. In line 9, subrequests are dispatched to the corresponding tier. The request will be dispatched before updating the access table and the access queue. Note that processing the subrequest and updating the access table and access queue can be done concurrently. This can remove the overhead of updating access table in multi-core systems. In lines 10 through line 23, the access queue and the access table are updated. In line 24 through line 27, the number of requests after the last migration will be increased. If this variable reaches the migration-time threshold (which will be set by the user), the migration function will be called. An alternate approach is to set a predefined time for migrations. Both of these approaches can be implemented in OODT.

The migration planner is another component of the proposed architecture, which generates a sorted list of data blocks based on their priorities. The migration planner finds the candidate block to be migrated to another tier and issues the required requests to disks to migrate data blocks. Upon completion of these requests, the mapping table will be updated. The migration planner also tries to place data blocks based on the corresponding logical addresses in

ALGORITHM 2: Migration Planner Algorithm

```
1  Sort the list of blocks based on priority;
2  number of tiers ← number of tiers;
3  migration matrix ← a two-dimensional matrix[number of tiers][number of tiers];
4  foreach tiers do
5      thistier ← current tier;
6      foreach blocks selected for thistier do
7          if block is not in thistier then
8              tierofblock ← current tier of block;
9              Add block to migration matrix [tierofblock][thistier];
10         end
11     end
12 end
13 foreach cell in migration matrix do
14     Issue the required requests for migrating;
15     Update mapping table;
16 end
```

HDD tiers since it will prevent the fragmentation of requests and the conversion of a large sequential request into several smaller random subrequests. In the proposed technique, the migration planner is called after a speficied number of requests. However, another approach is calling the migration planner when the system is in the idle state. This approach can also be applied to the proposed architecture if needed.

The migration planner uses Algorithm 2 for data block migration. In line 3, a $n \times n$ matrix is created where n is the number of tiers. This matrix will be filled with data blocks which should be migrated to the other tiers as shown in line 4 through line 12. Each matrix cell, i.e., $matrix[i][j]$, consists of a linked list from blocks which should be moved from tier i to tier j. In line 13 through line 16, the migration planner moves data blocks between tiers based on lists of data blocks in the matrix. This might need to temporary store a few data blocks in the main memory. The memory overhead is negligible and only occurs in short time period during migrations.

3.2 Priority Computation

Priority computation is arguably the most important aspect of a tiering or caching technique. Since priority computation determines which data blocks can benefit from placing in SSDs and therefore performance gain depends on the priority computation for choosing the best data blocks. To develop an efficient priority computation algorithm, we have examined SSDs and HDDs characteristics and four parameters have been selected for priority calculation accordingly. The selected parameters extracted by workload characterization are access frequency, random access frequency, read access frequency, and filesystem metadata access frequency. We propose a linear weighted formulation with the mentioned parameters for calculating the priority of each

request, according to Equation 1.

$$Priority = w_{access} * P_{access} + w_{random} * P_{random} + w_{read} * P_{read} + w_{metadata} * P_{metadata} \quad (1)$$

In this equation, w_{access}, w_{random}, w_{read}, and $w_{metadata}$ are weights for access, randomness, read, and metadata, respectively. P_{access}, P_{random}, P_{read}, and $P_{metadata}$ are priorities for access, randomness, read, and metadata, respectively. Priority calculations and the corresponding weights are discussed in detail in Section 3.2.1 through Section 3.2.3.

3.2.1 Access frequency and randomness

Access frequency is the most important feature for priority computation. Most of existing blocks in disks are typically cold blocks that will not be accessed in the near future. Access frequency is an efficient parameter to identify and place cold data blocks into inexpensive and high capacity tiers. As mentioned in Section 1, SSDs exhibit almost similar performance on random and sequential access patterns in contrast to HDDs that have very poor performance on random workloads. Hence, placing randomly accessed data blocks on SSDs will result in higher performance gain. Additionally, 15K HDDs provide higher performance in random workloads than 10K or 7K HDDs. Therefore, randomly accessed data blocks should be moved toward high performance tiers. We combine access frequency priority with random priority into a single parameter since they are closely related to each other. The priority of each request will be the inverse of the size of the request and the weight of this priority has been set to one. The intuition behind choosing the inverse of the request size is that HDDs have a setup time for rotating the disk head. The time overhead is not dependent on the request size. Therefore, a larger request size will have less average response time per data block. As such, choosing the inverse of the request size helps to identify requests with high average response time per block.

Since the read and metadata performance is affected by the request size, the priorities of read and metadata have been set to the priority calculated in this section and their weights will be chosen based on their impact on the overall system performance. The read and metadata priorities are calculated based on the access priority. Since the total priority is the sum of access, read, and metadata priorities, choosing any value for access priority will not affect the ranking of the priorities for various data blocks. Hence, we chose the value of 1 to simplify the equation and remove unnecessary calculations during runtime. Please note that since these calculations are performed during system runtime, more complex calculations can degrade the system performance. Hence, we keep the formulation simple to have minimal impact on computation power.

The I/O scheduler in operating systems attempts to merge requests close to each other to create a more sequential request in order to enhance performance. Hence, we store eight past requests addresses in a queue called access queue

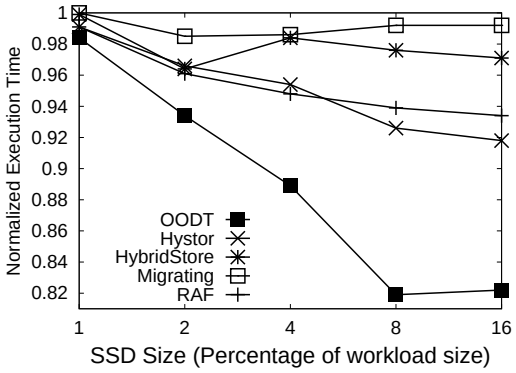


Fig. 2: Normalized execution time of various techniques

to find requests that will probably be merged in I/O scheduler, and consider them as a single request. This will enable us to more accurately identify random requests. The optimal size of the access queue depends on the number of data streams from applications. In addition, the number of active application data streams that try to simultaneously read from the disk is typically small. Since I/O requests depends on the previous requests, the number of pending requests will be limited. In the examined workloads in this paper, the number of data streams is not more than eight. Therefore, we used access queue size equal to eight, however, the size of the access queue can be customized by the system users based on the characteristics of the workloads.

To better understand the novelty of the proposed cost function, we have compared it to techniques used in the previous work. Fig. 2 depicts the normalized execution time of Postmark workload[1] with respect to the cost functions employed in the previous work. All execution times are normalized to a cost function that does not consider randomness and uses only access frequency to determine the priority of data blocks. As shown in Fig. 2, OODT improves performance up to 10% more than previous techniques. The execution of this experiment on other benchmarks yields almost the same result. Hence, we do not include such results here for the sake of brevity.

3.2.2 Read vs. write

SSDs exhibit higher performance in read requests because of the characteristics of flash memories. In addition, sending less write requests to a SSD will reduce the overhead of garbage collection and wear leveling and improves its lifetime. To develop a more efficient tiering technique, read-dominant data blocks should be moved to the high performance tier. In order to move read-dominant requests to the high performance tier, we assign a higher weight to the read requests. This means that if a request is read, we will add a specified

[1] This workload will be detailed in Section 4.2.1

percentage of the calculated access priority to the corresponding data blocks priority. Although SSDs use buffers to cache write requests and log structured writes to improve write performance, write requests will still affect SSD endurance. Additionally, the log structure will cause performance degradation due to mapping modification and garbage collection overheads when writing again to data blocks. Hence, placing write-dominated extents into SSDs will impose higher cost. Unlike SSDs, the response time of read and write requests are equal in HDDs. As a result, assigning higher priority to read requests in the HDD will not gain any performance improvement. Hence, once SSD tiers get filled, we will recalculate the priority of the remaining data blocks by removing the read priority. Afterwards, data blocks will be placed in HDD tiers based on the updated priorities. The optimal weight for read requests depends on workload and SSDs characteristics. In OODT, the read weight has been set to 10% which is semi-optimal case for majority of workloads such as Postmark, TPCH, and Webserver, based on the experiments presented in Section 5.5.1

3.2.3 Filesystem metadata

Many previous techniques give higher priority to filesystem metadata pages than data pages [13, 21, 18] since each metadata block will be accessed when a user tries to access a data block associated with the corresponding metadata block. Many metadata blocks such as *inodes* are associated to 1024 data blocks. Thus, metadata blocks have higher probability of access than data blocks associated with them. To the best of our knowledge, none of previous studies that give higher priority to the filesystem metadata have suggested assigning higher priority to hot data blocks as compared to cold metadata. To address this issue, we integrate the filesystem metadata priority into the priority of data blocks. Small weight is assigned to metadata requests, since most metadata blocks are accessed randomly and will have high priority due to their randomness. Sequential metadata requests which are usually used for logging purpose can exhibit higher performance on HDDs. Assigning aggressive high weight to metadata blocks will cause cold metadata to replace hot data blocks. The weight of metadata should be selected such that it is small enough to keep cold metadata blocks out of high performance tiers and big enough to give hot metadata blocks higher priority than hot data blocks. The effect of using different weights will be discussed in Section 5.5.2.

3.3 Filesystem Metadata Detection Technique

The filesystem stores additional data to manage user data, called *metadata*. The filesystem needs to access many metadata blocks for each user data request. In addition, metadata accesses might be required after completing user requests, e.g., to update the last access time of an accessed file. This flow has been illustrated in Fig. 3 where metadata and user data requests are shown by dark and white arrows, respectively.

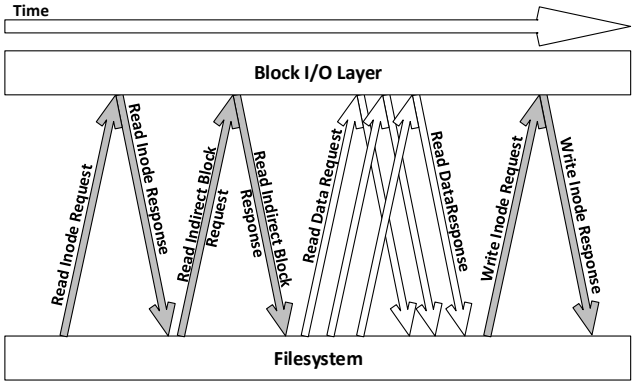


Fig. 3: Flow of subrequests to complete an arbitrary user request

The Linux I/O stack does not have a mechanism to pass semantic information from the filesystem or the buffer layer to the block I/O device layer. All previous techniques that use semantic information have tried to modify the operating system kernel. In OODT, *BIO_META* flag in *bio* structures which is not used in the current Linux versions, has been exploited for flagging metadata requests. In addition, this flag has been added to the *buffer_head* structure since many filesystem requests will be completed by the buffer layer and the buffer layer will send them to the block I/O layer.

Modifications in the buffer layer consists of adding flag to the *buffer_head* structure and sending flagged requests to the block I/O layer for metadata requests. In the filesystem, all requests dispatching to the buffer or block I/O layer has been flagged with *BIO_META* except requests for user data. Thus, indirect blocks and all other metadata pages without a fixed place in the address space are also considered as metadata, as opposed to previous efforts [13]. These modifications enable us to tag filesystem metadata requests.

3.4 Comparison of OODT with Previous Work

Table 1 compares the proposed technique with previous caching and tiering techniques. OODT mainly differs from previous work in parameter and the formulations used for priority calculation. In the techniques proposed in [11], [27], and [22], each request is characterized as either sequential or random. Additionally, there is no difference between an 8KB request and a 128KB request, since both of them will be characterized as sequential. *Hot-Random* [20] uses the same technique along with a randomness calculator which is similar to the proposed access queue but with higher overhead. Another technique to distinguish random requests from sequential requests is *HybridStore* [17] which classifies requests as highly-random (<16KB), partially sequential (32KB), and highly-sequential (>64KB). While this classification is useful, it will not be accurate when the size of all requests are close to each other. In

Table 1: Comparison of OODT with the previous work

	Tiering	Caching	Memory Overhead	Randomness	SSD Characteristics	OS Semantic	Online	Workload Adaptability
Azor [18]	×	✓	<1%	×	×	✓	✓	×
Hystor [8]	×	✓	<1%	✓	×	✓	✓	×
Migrating to SSDs [27]	×	✓	<1%	✓	✓(sec. parameter)	×	✓	✓
RAF [22]	×	✓	<0.01%	✓	✓	×	✓	×
Cost-Effective [11]	✓	×	<0.0001%	✓	×	×	✓	N/A
Macss [21]	✓	✓	<0.0001%	×	×	✓	✓	×
HybridStore [17]	✓	×	<0.001%	✓	✓	×	×	✓
Hot-Random [20]	✓	×	<0.0001%	✓	×	×	✓	N/A
Flashing-Up [19]	✓	×	<1%	×	✓(out of date)	×	✓	×
OODT	✓	×	<1%	✓	✓	✓	✓	✓

addition, this technique does not consider the effect of I/O scheduler. *Hystor* [8] uses an inverse bitmap for storing access frequency and request size in one variable which is similar to the random access frequency in the proposed technique. Inverse technique, however, is less accurate since it uses the logarithm of the request size instead of the actual request size.

The technique presented in [19] (called *Flashing Up)* places data blocks on disks with respect to read/write cost for storage devices without considering other parameters such as request size while OODT considers such parameters. In addition, this technique considers the characteristics of traditional SSDs which are not applicable to state-of-the art SSDs. RAF [22] uses almost the same technique and considers the characteristics of state-of-the art SSDs. Similar to *Flashing Up*, *Hybrid Store* needs information about read/write performance of each storage device from data sheets or performance tests for optimizations which might be difficult to obtain. The technique presented in [27] uses SSDs characteristics only as a secondary priority which will not have significant effect on performance since the main priority is based on random access frequency which has a low probability of two blocks being the same.

Most of the previous work that give filesystem metadata higher priority do not consider lower priority for cold metadata over hot data [21, 18]. Hystor [8] is another technique that places metadata in a special region of SSD cache. The optimal size of this section depends on workload access pattern. In contrast, OODT is flexible and can adapt itself to various workloads. Choosing an optimal size for the metadata section is vital for gaining high performance improvement.

The techniques employing workload adaptability such as [27, 17] do not depend on one request type or one characteristic towards performance optimizations. Adaptability ensures that the technique will provide at least a

moderate performance in most workloads. To reach adaptability, none of the parameters should have absolute priority over the other parameters. In addition, using a predefined section for a request type (i.e., write or metadata) is against adaptability. If the workload lacks that request type, the allocated section will only waste high performance tier space without any performance improvement. In Table 1, workload adaptability entries denoted by "N/A" indicates that the target technique only uses one parameter and, hence, workload adaptability is not applicable.

The memory overhead in Table 1 indicates an approximate upper limit of memory overhead instead of the actual memory overhead, since there is no reported information in previous work to calculate the exact memory overhead per gigabyte. OODT has relatively high memory overhead, since it tries to maintain an accurate data tiering technique. The memory overhead of the OODT will be further discussed in Section 5.1.

4 Experimental Results

4.1 Experimental System

In the experimental setup, we have used a workstation with a Pentium 4 2.8 GHz processor equipped with 2GB main memory and a 250 GB Maxtor 7K HDD for benchmarking. To decrease the effect of the buffer layer, its contents are synchronized with disks and cleared every second. Additionally, the operating system is installed on a separate hard disk to remove possible effects of operating system or other applications on benchmarks.

The operating system employed in the experimental setup is Ubuntu 12.04 Precise Pangolin with kernel 3.1.4 and ext3 filesystem. The tiering device is, however, formatted with ext2 [6] filesystem. We have selected ext2 filesystem since it is widely used by Linux users. The modifications applied to this filesystem mainly include tagging all requests for metadata blocks with a metadata flag. These modifications are also applicable to other filesystems, too. Table 2 shows the modified *Lines of Code* (LOC) for the proposed metadata detection technique. The source code of the proposed tiering technique is publicly available at [31]. To stress the tiering system, the filesystem has been mounted with the sync option. We set the best I/O scheduler for each tier (*noop* for the SSD tier and *CFQ* for the HDD tier). In the experiments, OODT has been implemented using only two tiers. A high performance tier is mapped to a SSD OCZ 40GB and a high capacity tier is implemented using a Maxtor 500GB 7K HDD.

It is notable to mention that the number of tiers can further be extended. However, extension of the experiments to more than two tiers needs much more hardware, and requires very large scale workloads to be able to use the benefits of all tiers.

To create a completely reproducible workload, *blktrace* [5] is used to capture I/O requests of workloads and *btreplay* is employed to rerun these workloads.

Table 2: Modified Lines of Code in the Filesystem

File	Modified LOC
fs/buffer.c	9
fs/ext2/balloc.c	6
fs/ext2/ialloc.c	4
fs/ext2/inode.c	13
fs/ext2/super.c	10
fs/ext2/xattr.c	9
include/linux/buffer_head.h	8

To evaluate the performance efficiency of OODT, the execution time of the workloads has been compared with four architectures a) a pure HDD disk subsystem, b) a pure SSD disk subsystem, c) three-level table algorithm proposed in [8] (HyStor), and e) sequence detection and randomness threshold used in [22] (RAF).

Although the last two architectures originally have been proposed for caching solutions, as shown in Section 3.2.1, the randomness priority employed in these architectures have the best performance in tiering solutions among all previous work. In addition to having a semi-optimal randomness priority, HyStor assigns higher priority to the metadata requests (with a technique which differs vastly from OODT) and RAF tries to place read intensive requests in the SSD, similar to the read priority explained in Section 3.2. Thus, we have these two architectures for the evaluation of OODT to ensure that all aspects of the proposed technique will be compared to the previous techniques with the highest performance.

4.2 Benchmarks

To fully evaluate this tiering system, we have used various benchmarks with different characteristics. These benchmarks are detailed in the following sections.

4.2.1 Postmark

Postmark is a mail server benchmark that creates many file and folders and runs transactions on these files [14]. Transactions are read, write, delete, or create files. This benchmark generates many filesystem metadata requests and many small random read or writes. Write ratio, requests size, file deletion ratio, transactions count, and many other parameters can be configured in this benchmark.

4.2.2 HammerDB

HammerDB is a database benchmarking tool that can simulate TPC-C and TPC-H benchmarks [32]. TPC-C queries are mostly small random similar to

Postmark. However, unlike Postmark, TPC-C does not generate many filesystem metadata. In addition, all requests are for database file which is quite large-size file unlike Postmark files which are small in size. TPC-H benchmark is a *Decision Support System* (DSS) simulator and consists of 22 ad-hoc queries. Similar to TPC-C, TPC-H dispatches mostly small random requests to the disk subsystem.

4.2.3 IOzone

IOzone is a benchmarking tool for filesystems that is able to run various tests in order to measure different aspects of filesystems [28]. This tool consists of 13 types of tests which can be run separately or in conjunction with each other.

4.2.4 FileBench

Filebench is a benchmarking platform that can run many different workloads using *Workload Model Language* (WML) [36]. We have used FileBench to simulate a web server. The web server workload creates a pool of files with the mean size of 16 KB. A user access to the web server will be simulated by reading ten files from file pool and then appending 16 KB data to the log file.

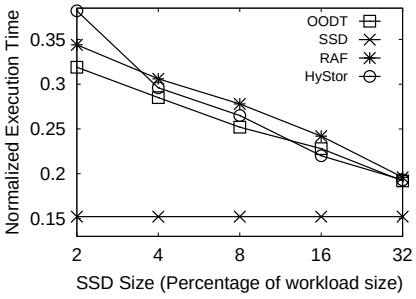
4.3 Experimental Results

In this section, experimental results for target benchmarks are presented.

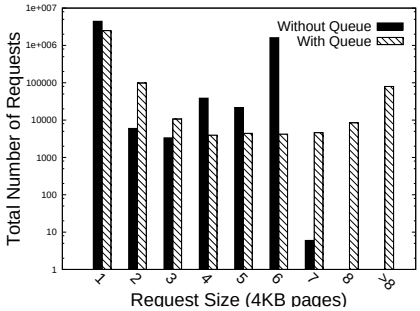
4.3.1 Postmark

We have configured Postmark to create 40,000 files with varying size from 32KB to 40KB and executed 80,000 read, write, create, and delete transactions on them with 4KB granularity. Postmark has three stages: creating files, running transactions, and deleting files. Our analysis shows that most requests in the first stage are metadata writes. The second stage is a mixed workload of metadata and random data accesses and the third stage has only metadata requests.

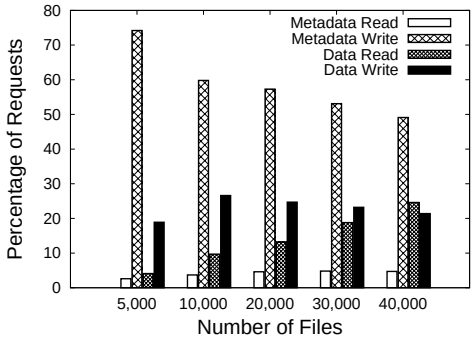
Fig. 4a shows the execution time of OODT and the previous work, which has been normalized to the execution time of a pure HDD disk subsystem (a Seagate 7K HDD). As illustrated in this figure, the proposed characterization outperforms previous techniques. In addition, OODT reduces the execution time by 72% compared to pure HDD while only 4% of data blocks are migrated to the SSD tier. OODT has up to 16.4% and 7% on average less execution time when compared to the previous techniques. The average performance gain depends on the range of SSD sizes used in the tiering techniques. This is due to when SSD size is large enough, all tiering techniques will have almost the same performance. Therefore, we excluded large SSD sizes for calculating



(a) Normalized execution time



(b) Request size distribution with and without the access queue



(c) Percentage of data read, data write, metadata read, and metadata write

Fig. 4: Normalized execution time and distribution of request size and types for Postmark workload

the average performance gain. The improved performance over the conventional tiering technique is due to the fact that OODT considers randomness, read/write ratio of data blocks, and metadata requests. In addition, each part of the priority calculation technique outperforms other techniques in that area.

The HyStor technique, however, outperforms the proposed technique when 16% of data blocks are in the SSD. Since this technique tries to place data blocks physically near to the highly accessed data blocks in the SSD tier, it provides higher performance when there is enough space in the SSD. On the other hand, when high performance tier does not have enough space for these data blocks, they will only occupy valuable space in the SSD. Therefore, the performance will decrease significantly (as can be seen in Fig. 4a) when only 2% of data blocks are in the SSD.

Fig. 4b depicts the request size distribution of this workload. Maximum request size and the average request size are 28KB and 9.5KB, respectively. Hence, this workload is characterized as a random workload. The proposed

access queue for eight past requests introduced in Sec. 3.2.1 identifies many large requests that will be created in I/O scheduler by merging smaller requests. Since most of requests in this workload are small size, random priority will not have any advantages against access frequency tiering. Hence, metadata and read priority are responsible for the performance improvements.

Fig. 4c shows the percentage of metadata requests in Postmark for various file counts. In all benchmarks, the number of transactions is double the number of files. Decreasing the number of transactions makes the creating phase of the benchmark dominant. Our experiments show that increasing the number of transactions greater than twice the number of files will not change the behaviour of the benchmark and will only increase the execution time. With growing number of files, this workload changes from a metadata intensive to data intensive workload. Thus, metadata-only tiering techniques such as [21, 13, 16, 26] will degrade the performance gain with the increasing number of files or transactions. OODT uses metadata priority along with other parameters. Thus, it can adapt itself to changes in the workload characterizations.

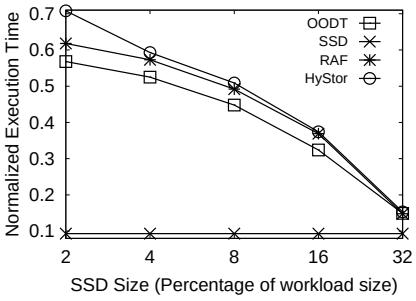
One of the interesting points in the results is that if the high performance tier is large enough to hold most of the data blocks, the difference between various tiering techniques becomes negligible. This is due to even simple techniques such as access frequency tiering will move all active data blocks to the high performance tiers.

4.3.2 TPC-C

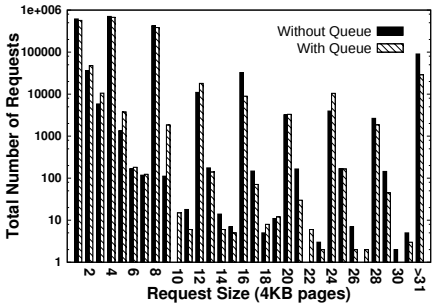
In our configuration, HammerDB uses a *MySQL* database configured with default parameters and executes 1,000,000 transactions. This workload has larger requests than Postmark with the average request size of 22KB. Thus, requests are more sequential in this workload.

Fig. 5a shows the normalized execution time with various configurations. In this workload, OODT improves performance up to 43% as compared to a pure HDD disk subsystem by placing only 2% of data blocks in the SSD tier. Additionally, our technique improves the execution time up to 20% as compared to the previous techniques. In this workload, as with Postmark, HyStor exhibits low performance improvement when the size of the high performance tiers is small. Although OODT maintains its performance gap with previous techniques in various SSD sizes, if the high performance tier is large enough, the conventional and the proposed tiering techniques will have similar performance.

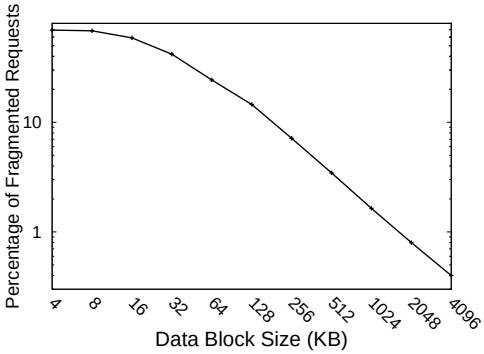
Fig. 5b shows the distribution of request sizes. This distribution is unique and completely differs from the other workloads, since requests are dispatched from a *DataBase Management System* (DBMS), as opposed to Postmark, webserver, and IOzone workloads which are file-based workloads. Database workload patterns depend on DBMS, database schema, and query types. Further investigation is required for characterization of database workloads. Fig. 5c shows the percentage of fragmented requests for various data block sizes. We



(a) Normalized execution time



(b) Request size distribution with and without the access queue



(c) Percentage of fragmented requests

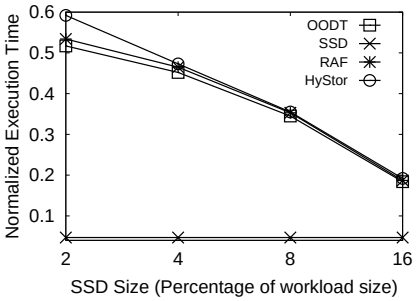
Fig. 5: Normalized execution time, request size distribution, and percentage of fragmented requests for TPC-C workload

chose the smallest data block size, since using larger data blocks will not decrease the number of fragmented requests significantly. OODT tries to place data blocks with sequential logical addresses close to each other to decrease the effect of fragmentation while having more accuracy. The effect of choosing larger data block sizes will be discussed in Section 5.2.

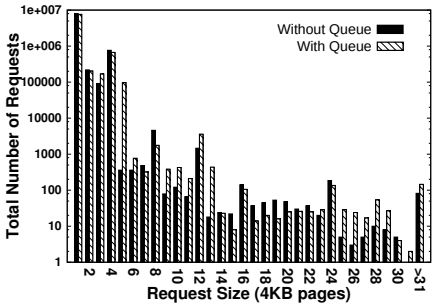
This workload has a few metadata requests and our metadata priority will not have any performance gain in this workload. On the other hand, since there exists many sequential requests, OODT will have an advantage over less accurate tiering techniques.

4.3.3 TPC-H

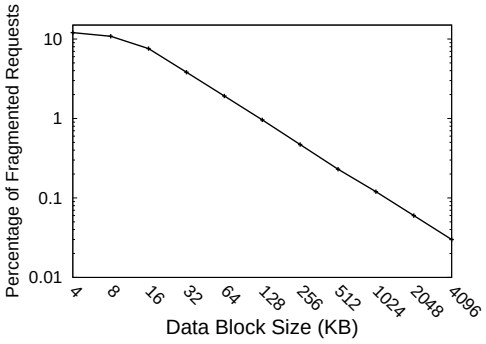
In this benchmark, similar to TPC-C, HammerDB uses MySQL database with one difference: the default engine has been replaced with *MyISAM*, since the default engine is not compatible with this workload. The scale factor of this workload has been set to 10.



(a) Normalized execution time



(b) Request size distribution, with and without the access queue



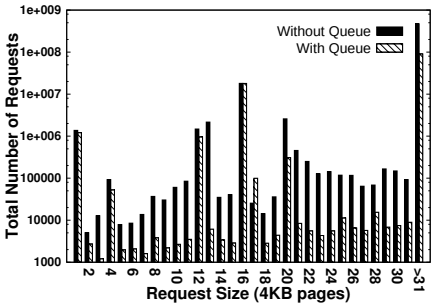
(c) Percentage of fragmented requests

Fig. 6: Normalized execution time, request size distribution, and percentage of fragmented requests for TPC-H workload

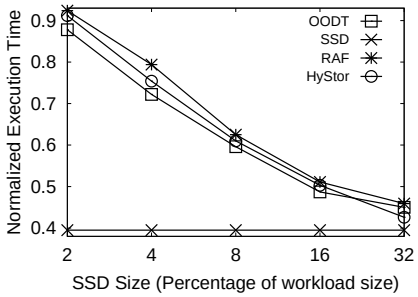
Fig. 6a shows the normalized execution time for this workload. As shown in this figure, the proposed technique improves the execution time by 50% compared to running on HDDs while only 2% of data blocks are in the SSD. The proposed technique improves the performance on average by 3% and up to 13% as compared to the previous work.

This workload is heavily random- and read-dominant. As Fig. 6b depicts, more than 85% of requests are 4KB and the average request size is 5.3KB. In addition, Fig. 6c shows that the percentage of fragmented requests is much lower than TPC-C workload, which emphasizes that using smaller data block sizes is more efficient in these workloads. TPC-H queries mostly retrieve data and do not insert any data in the database. In particular, read requests constitute 99% of the total requests. Most of the write requests are for updating metadata of database files.

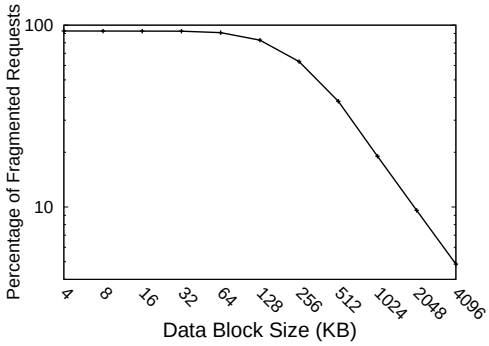
Considering the read ratio and the average request size of this workload, the only means to improve performance in this workload is considering access frequency and metadata priority. Since all previous works consider access fre-



(a) Request size distribution with and without the access queue



(b) Normalized execution time



(c) Percentage of fragmented requests

Fig. 7: Normalized execution time, request size distribution, and percentage of fragmented requests of IOzone workload

quency and all the highest accessed blocks are metadata, the proposed technique cannot improve performance when compared to the previous work as much as in the other workloads.

4.3.4 IOzone

In the experiments, IOzone has been configured to run all 13 types of tests for a specified time. Many of the tests in this benchmark have large number of sequential requests. Hence, this workload can be categorized as sequential-dominated workload. As Fig. 7a shows, most of the requests in this workload requires more than 32 data pages. This fact will reduce the gained improvement from moving data pages to SSDs. This is mainly due to these data blocks hlhaving relatively good performance in HDDs.

IOzone and TPC-H are quite opposite workloads since TPC-H is random-dominated and this workload is sequential-dominated. In contrast to this difference in characteristics, modern tiering techniques cannot significantly im-

prove performance compared to simple tiering techniques in either of these workloads. Fig. 7b depicts the normalized execution time of this workload for various techniques. In this workload, HyStor outperforms the RAF technique because this workload is sequential-dominated and the spatial locality of this workload is greater than previous workloads. Since most requests in this workload are very large, they will cross boundary of many data blocks. Increasing the data blocks' size will not decrease the number of fragmented requests, as shown in Fig. 7c. Although the percentage of fragmented requests does not decrease, requests will be broken into less subrequests when using larger data block sizes. Hence, the overhead of creating subrequests decreases.

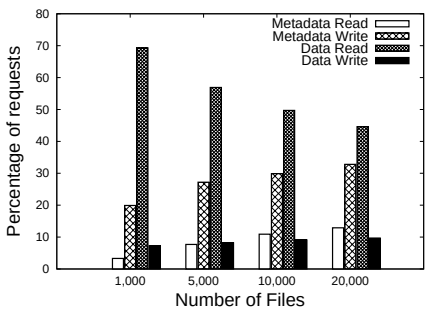
OODT has up to 9% and 5% on average less execution time compared to the previous work. This improvement is negligible as compared to the other workloads since most requests are either sequential- or read-dominated. In addition, most of the metadata blocks have high access frequency which will have high priority even in techniques that does not consider metadata priority. When 32% of data blocks reside on the high performance tier, HyStor exhibits higher performance than the proposed technique due to moving pages near the high frequently accesses pages to the SSD (similar to the Postmark workload when 16% of data blocks are in the high performance tier).

4.3.5 FileBench

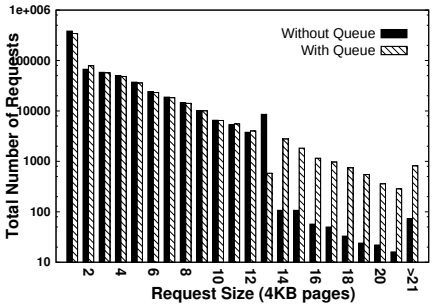
Filebench web server workload creates 10,000 files with mean directory size of 20 files. Although this workload has large sequential requests (e.g., more than 50 data blocks), the average requests size is still 10KB. This demonstrates that the web server workload is a random workload with a few large requests that will be easily detected by OODT and will be placed on the HDD tier. Additionally, since this workload uses a large pool of files, many metadata accesses are required for accessing files. Fig. 8a shows the percentage of metadata requests for this workload. As shown in this figure, creating more files will result in more metadata requests in contrast to Postmark, since there exists more requests per file in Postmark as compared to webserver workload.

We have configured the web server workload with 10,000 files. According to the results reported in Fig. 8a, this workload has 60% and 40% read and metadata requests, respectively. Therefore, this workload can be a fair representative of how well OODT can improve the performance on a workload without a dominant parameter as apposed to the other workloads. The distribution of request sizes, with and without eight-request access queue is presented in Fig. 8b, illustrating that the proposed technique can detect sequential requests that will be created by I/O scheduler and help randomness priority to be more accurate in assigning priorities.

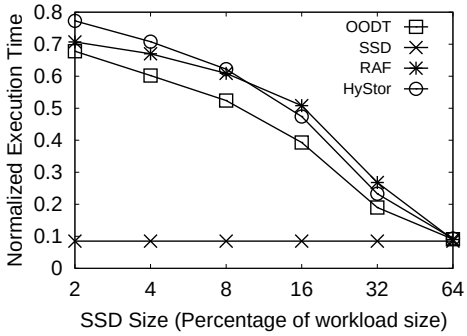
This workload has many random write requests with larger read requests. This combination is suitable for evaluating read and randomness priorities to examine if their weights are chosen correctly. If weights are chosen incorrectly, less performance improvement will be observed when compared to the other



(a) Percentage of data read, data write, metadata read, and metadata write



(b) Request size distribution with and without the access queue



(c) Normalized execution time

Fig. 8: Normalized execution time and distribution of request size and types for Webserver workload

workloads. Performance evaluations show that our technique employing a SDD which contains 16% of data blocks can improve performance by 41% as compared to the pure HDD-based disk subsystem. OODT has also up to 30% and 16% on average less execution time as compared to the previous techniques as shown in Fig. 8c. This improvement over the previous techniques shows that our technique has chosen parameters more efficiently and if a workload has many metadata requests, various request sizes, and read/write combination, this technique can improve performance significantly.

As mentioned earlier in this section, this workload has many metadata requests. Hence, metadata blocks will be accessed more frequently than the other workloads and any technique which assigns higher priority to metadata requests will further improve performance. This will justify HyStor outperforming RAF in this workload when it has enough high performance tier space for storing data blocks. The proposed technique, on the other hand, uses many priority techniques simultaneously which enables it to achieve the highest performance among all workload types compared to the previous work.

5 Discussion

5.1 Memory Overhead

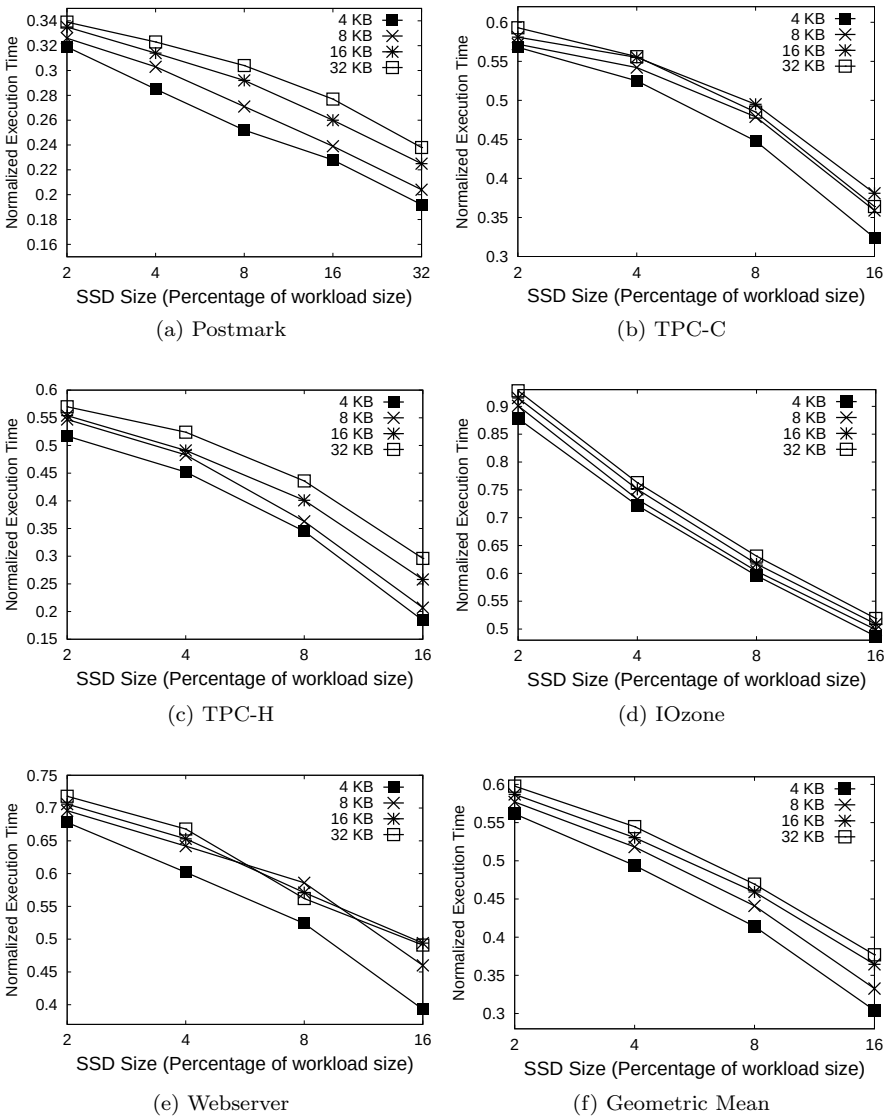
The tiering technique requires memory to store a mapping table, statistics, and other required data structures. The memory overhead can be expressed as overhead memory bytes per data block. Using larger data blocks will reduce memory overhead but may cause performance degradation. This is due to a data block with high priority possibly having many cold pages that will occupy valuable space in the high-performance tier. Our proposed technique requires six bytes for mapping and twelve bytes for statistics per data block. Memory overhead for 4KB and 4MB data blocks is 0.43% and 0.0004%, respectively. In large storage servers, allocating memory for 4KB pages is not affordable. As a result, to prevent performance degradation using 4MB data blocks, one can use a fixed memory size for statistics. In this approach, statistics are organized in a *Least Recently Used* (LRU) queue. In the case in which the statistics table reaches the maximum size, the statistics of the last data blocks will be removed from the table.

5.2 Data Block Size

As previously discussed, choosing larger data blocks will result in less accurate data tiering and lead to performance degradation. Larger data blocks, however, will reduce the memory overhead and the overhead of splitting requests. Fig. 9a shows the effect of data block size on performance in the Postmark workload. With increasing SSD size, the performance gap between various configurations increases as well since more cold data pages will be present in the SSD. Fig. 9b depicts the normalized execution time with various data block sizes for TPC-C workload. In this workload, 32KB data block configuration outperforms 16KB data block configuration when the size of the high performance tier is greater than 4% of the total workload size. Since in this workload, as opposed to Postmark, many hot data pages with sequential logical address will reside on the SSD when using large SSD sizes. In addition, the percentage of cold data pages decreases in the SSD. Additionally, configurations with larger data block sizes have less overhead for splitting requests. In even larger SSD sizes, all configurations have almost the same performance for this workload (not shown in Fig. 9b).

TPC-H workload is random-dominated and choosing smaller data block sizes can prevent cold data blocks from entering the high performance tier. Fig. 9c depicts the normalized execution time for various data block sizes. The 4KB data block size has up to 38% less execution time compared to 32KB data block size. This is mainly due to this workload being random-dominated, as mentioned in Section 4.3.3.

IOzone is a sequential-dominated workload. Hence, using tiering with small data blocks sizes will not have significant performance improvement. Fig. 9d



(a) Postmark (b) TPC-C

(c) TPC-H (d) IOzone

(e) Webserver (f) Geometric Mean

Fig. 9: Normalized execution time with various data block sizes

depicts the normalized execution time for different data block sizes. 4KB data block sizes only improves the performance by 5% compared to the 32KB configuration. We should note that 4KB configuration has almost 4x memory overhead and since this configuration cannot improve performance significantly, in sequential-dominated workloads using larger data block sizes will impose less overhead and almost the same performance.

Webserver workload has random and sequential requests simultaneously. Thus, many data blocks in the SSD will contain hot and cold data pages and the others will be filled by hot data pages. As can be seen in Fig. 9e, the 8KB configuration is outperformed by 16KB and 32KB when 8% of data blocks are in the SSD. This is mainly due to the higher overhead of the 8KB configuration and the fact that many sequential data blocks have been moved to the SSD tier. In other SSD sizes, however, the number of cold data pages in the SSD prevents configurations with larger data block sizes from outperforming 8KB configurations. As shown in Fig. 9e, the 4KB configuration always outperforms the other configurations. This reveals that preventing cold data pages from entering high performance tiers is much more important than the overhead of splitting requests.

The geometric mean of all workloads is depicted in Fig. 9f. This figure shows that choosing 4KB data block size is the best configuration in the average case if the required memory overhead can be provided. We can conclude that in sequential workloads, larger data block sizes exhibit higher performance, since they have less overhead while in random workloads, smaller data blocks sizes perform better since they have less cold data pages in the high performance tier.

5.3 Reliability

The mapping table is the most critical data on any data tiering system since if the mapping table gets corrupted or deleted, the physical location of logical data blocks will be unknown. Therefore, losing the mapping table will result in losing all data blocks. Hence, any data tiering technique should guarantee the protection of the mapping table in case of system crash or power outage. In OODT, a copy of the mapping table is stored on the high capacity tier. This may impose only a negligible performance overhead since the mapping table only changes when data migration is activated. If the system crashes while data blocks are being migrated, there is a possibility of a data loss event. To prevent such a data loss event, one can use a logging scheme on a persistent storage device to store the migrated data blocks and the corresponding status flags. With such a logging scheme, the tiering system can recover from possible crashes and the possibility of data loss events is significantly decreased.

Many modern filesystems, such as ext3/4, that are extensions of ext2 use journaling to ensure filesystem remains consistent after an operating system crash or power failure [35]. Early journaling techniques used barrier requests to properly implement journaling. All requests arriving before a barrier request should be completed before this request issues, and all requests arriving after a barrier request will be issued after its completion [9]. Barrier requests can be easily implemented in the data tiering kernel module. These requests, however, affect the system performance and as a result, are removed from the kernel since version 2.6.37. Modern journaling techniques use *REQ_FLUSH/FUA* flag in requests. The proposed tiering technique copies all flags from the bio

structures of incoming requests to the structures of all subrequests that will be sent to the storage devices. Hence, these journaling techniques are compatible with the proposed tiering technique.

5.4 Portability

Many of the previous works require heavy modifications in the kernel [25, 23], analyzing and benchmarking of used storage devices [19, 17], or designed for a specific configuration or filesystem [13, 21]. This makes such techniques less portable, deployable, and maintainable. In OODT, all implementations can be encapsulated in a kernel module, except modifications applied to the filesystem which require slight modifications in the filesystem. Hence, the proposed technique is portable without metadata priority, and, with a few changes in the filesystem, metadata priority can be enabled.

5.5 Parameter Sensitivity

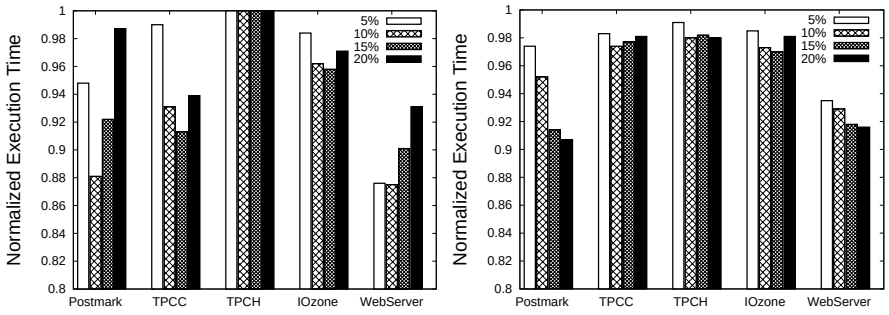
Read and metadata priority in the proposed technique are calculated by a weight of random access priority. The value of these weights have significant impact on the performance. The nearly-optimal values for parameters depend on the running workload. These values can be extracted by running the workloads with various values and selecting the best fitted values. Another approach is using analytical methods for predicting the effect of each parameter on the overall system performance. Analytical modeling of the performance in multi-tiered systems is part of our future work. In this section, the results on choosing different values for these weights will be presented.

5.5.1 Read weight

Fig. 10a shows the normalized execution time of the proposed technique with different weights for read priority (all values are normalized to the execution time with value '0' for the weight of the selected parameter). As this figure shows, semi-optimal values for different workloads are similar to each other except for TPC-H workload in which 99% of requests are read. Hence, using a constant value for the weight of read requests can be practical.

5.5.2 Metadata Weight

The optimal weight for metadata depends on the workload and the filesystem on which the workload is running. Analyzing the impact of the filesystem and its configuration on the SSD performance in the previous work shows that different filesystems and even configurations of a filesystem can result in significant difference as large as 30% in performance [38]. Thus, we attempted to select a small value for the weight of metadata in order to reduce the chance



(a) Read Priority (b) Metadata Priority

Fig. 10: Normalized execution time with various weights

of performance degradation because of assigning high priority to metadata requests.

Fig 10b depicts the normalized execution time for various metadata weights. The optimal value for ext2 filesystem is close to 15% which is decreased to 10% in this work for more generality of the proposed technique for other filesystems.

5.6 Workload Generality

The running workload has a high impact on the performance gain we can expect from OODT. In the experimental results section, we tried to cover many workloads from different applications with different characteristics. In order to show that the workloads are general, we analyzed them and reported their characteristics in Section 4. The workloads used in this study span random-dominant (TPC-H, Postmark), sequential-dominant (IOZone), read-dominant (TPC-H), write-dominant (Postmark), metadata-intensive (Postmark), data-intensive (TPC-C), and workloads with no dominant factor (Web server) types of applications which cover most of the real workloads.

We have provided samples for several workloads which can have different characteristics when configured with various values for their parameters (e.g., the number of files), namely for web server and Postmark workloads. The changes in the parameter values of the web server and Postmark workloads will transform them into a workload with other characteristics. These characteristics, however, will be similar to the workloads, which have already been analyzed in this paper.

6 Conclusion

In this paper, a data tiering technique based on online workload characterization was presented. OODT endeavours to place data blocks based on their

access pattern and storage devices characteristics on a proper tier. Access frequency, random access frequency, metadata access frequency, and read access frequency are used to calculate the priority of data blocks. To evaluate the proposed technique, a Linux kernel module was developed which runs under an ext2 filesystem. Experimental results showed that this technique can improve performance up to 72% compared to a pure HDD system by placing less than 4% of data blocks on the high performance tier. In addition, the experimental results demonstrated that the proposed technique reduces the execution time up to 30% in the commonly used workloads such as database, mail server, and webserver.

References

1. Agrawal N, Bolosky WJ, Douceur JR, Lorch JR (2007) A five-year study of file-system metadata. ACM Transactions on Storage (TOS) 3(3), DOI 10.1145/1288783.1288788
2. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference, AFIPS '67 (Spring), pp 483–485
3. Appuswamy R, van Moolenbroek D, Tanenbaum A (2012) Integrating flash-based SSDs into the storage stack. In: 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST), pp 1 –12, DOI 10.1109/MSST.2012.6232365
4. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. Journal of Parallel and Distributed Computing 10(2):188 – 192, DOI http://dx.doi.org/10.1016/0743-7315(90)90028-N, URL http://www.sciencedirect.com/science/article/pii/074373159090028N
5. Brunelle AD (2006) Block I/O layer tracing: blktrace. In: Gelato-Itanium Conference and Expo (gelato-ICE)
6. Card R, Ts'o T, Tweedie S (1994) Design and implementation of the second extended filesystem. In: Proceedings of the 1st Dutch International Symposium on Linux, pp 1–6
7. Chen F, Koufaty DA, Zhang X (2009) Understanding intrinsic characteristics and system implications of flash memory based solid state drives. SIGMETRICS Performance Evaluation Review 37(1):181–192, DOI 10.1145/2492101.1555371, URL http://doi.acm.org/10.1145/2492101.1555371
8. Chen F, Koufaty DA, Zhang X (2011) Hystor: making the best use of solid state drives in high performance storage systems. In: Proceedings of the 13th International Conference on Supercomputing (ICS), pp 22–32, DOI 10.1145/1995896.1995902
9. Corbet J, Rubini A, Kroah-Hartman G (2009) Linux Device Drivers. O'Reilly Media

10. EMC Corporation (2011) EMC FASTVP for unified storage systems. Tech. Rep. h8058.3, EMC
11. Guerra J, Pucha H, Glider J, Belluomini W, Rangaswami R (2011) Cost effective storage using extent based dynamic tiering. In: Proceedings of the 9th USENIX Conference on File And Stroage Technologies (FAST), pp 20–20
12. Jin X, Jung S, Song YH (2010) Write-aware buffer management policy for performance and durability enhancement in NAND flash memory. IEEE Transactions on Consumer Electronics 56(4):2393 –2399, DOI 10.1109/TCE.2010.5681118
13. Kaiser J, Meister D, Hartung T, Brinkmann A (2012) Esb: Ext2 split block device. In: 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS), pp 181 – 188
14. Katcher J (1997) Postmark: A new filesystem benchmark. Tech. Rep. TR3022, Network Appliance
15. Kim J, Seo S, Jung D, Kim JS, Huh J (2012) Parameter-aware I/O management for solid state disks (SSDs). IEEE Transactions on Computers (TC) 61(5):636 –649, DOI 10.1109/TC.2011.76
16. Kim JK, Lee HG, Choi S, Bahng KI (2008) A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. In: Proceedings of the 8th ACM International Conference on Embedded software (EMSOFT), pp 31–40, DOI 10.1145/1450058.1450064
17. Kim Y, Gupta A, Urgaonkar B, Berman P, Sivasubramaniam A (2011) HybridStore: A cost-efficient, high-performance storage system combining SSDs and HDDs. In: 19th IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), pp 227 – 236
18. Klonatos Y, Makatos T, Marazakis M, Flouris M, Bilas A (2011) Azor: Using two-level block selection to improve SSD-based I/O caches. In: 6th IEEE International Conference on Networking, Architecture and Storage (NAS), pp 309 –318, DOI 10.1109/NAS.2011.50
19. Koltsidas I, Viglas SD (2008) Flashing up the storage layer. Proceedings of the VLDB Endowment 1(1):514–525, DOI 10.1145/1453856.1453913
20. Lin L, Zhu Y, Yue J, Cai Z, Segee B (2011) Hot random off-loading: A hybrid storage system with dynamic data migration. In: Proceedings of the 19th IEEE Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp 318–325, DOI 10.1109/MASCOTS.2011.41
21. Liu S, Jiang J, Yang G (2012) Macss: A metadata-aware combo storage system. In: International Conference on Systems and Informatics (ICSAI), pp 919 –923, DOI 10.1109/ICSAI.2012.6223157
22. Liu Y, Huang J, Xie C, Cao Q (2010) RAF: A random access first cache management to improve SSD-based disk cache. In: 5th IEEE International Conference on Networking, Architecture and Storage (NAS), pp 492 –500, DOI 10.1109/NAS.2010.9

23. Luo T, Lee R, Mesnier M, Chen F, Zhang X (2012) hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. Proceedings of the VLDB Endowment 5(10):1076–1087
24. Martin J, Clayton N, Frese LL, Hossain K, McNutt B, Xu Y (2011) IBM system storage DS8800 and DS8700 performance with easy tier 3rd generation. Tech. Rep. WP102024, International Business Machines Corporation
25. Mesnier MP, Akers JB (2011) Differentiated storage services. SIGOPS Operating Systems Review 45(1):45–53, DOI 10.1145/1945023.1945030, URL http://doi.acm.org/10.1145/1945023.1945030
26. Miller EL, Brand SA, Long DDE (2001) HeRMES: high-performance reliable MRAM-enabled storage. In: Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS), pp 95–99
27. Narayanan D, Thereska E, Donnelly A, Elnikety S, Rowstron A (2009) Migrating server storage to SSDs: analysis of tradeoffs. In: Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys), pp 145–158, DOI 10.1145/1519065.1519081
28. Norcott W, Capps D (2002) Iozone filesystem benchmark program
29. Ou Y, Härder T (2010) Clean first or dirty first?: a cost-aware self-adaptive buffer replacement policy. In: Proceedings of the 14th International Database Engineering & Applications Symposium (IDEAS), pp 7–14, DOI 10.1145/1866480.1866482
30. Roselli D, Lorch JR, Anderson TE (2000) A comparison of file system workloads. In: Proceedings of the 11th USENIX Conference on Annual Technical Conference (ATC), pp 4–4
31. Salkhordeh R (2014) Data Tiering Kernel Module. http://dsn.ce.sharif.edu/tiering/tiering-kernel-module.tar.gz, accessed: 2014-08-01
32. Shaw S (2012) HammerDB: the open source oracle load test tool
33. Shi L, Li J, Jason Xue C, Zhou X (2013) Hybrid nonvolatile disk cache for energy-efficient and high-performance systems. ACM Transactions Design Automation Electronic Systems 18(1):8:1–8:23, DOI 10.1145/2390191.2390199
34. Soundararajan G, Prabhakaran V, Balakrishnan M, Wobber T (2010) Extending SSD lifetimes with disk-based write caches. In: Proceedings of the 8th USENIX conference on File and Storage Technologies (FAST), pp 8–8
35. Tweedie S (2000) Ext3, journaling filesystem. In: Ottawa Linux Symposium, pp 24–29
36. Wilson A (2008) The new and improved filebench. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)
37. Yang P, Jin P, Yue L (2011) Hybrid storage with disk based write cache. In: Xu J, Yu G, Zhou S, Unland R (eds) Database Systems for Adanced Applications, Lecture Notes in Computer Science, vol 6637, Springer Berlin Heidelberg, pp 264–275
38. Zhou K, Huang P, Li C, Wang H (2012) An empirical study on the interplay between filesystems and SSD. In: 7th IEEE International Confer-

ence on Networking, Architecture and Storage (NAS), pp 124–133, DOI 10.1109/NAS.2012.21