A Utility-based Approach to Cost-Aware Caching in Heterogeneous Storage Systems

Liton Chakraborty[1], Ajit Singh[2]

[1]Dept. of Electrical and Computer Engineering
University of Waterloo
ON, Canada, N2L 3G1
litonc@swen.uwaterloo.ca

[2]Dept. of Electrical and Computer Engineering
University of Waterloo
ON, Canada, N2L 3G1
asingh@etude.uwaterloo.ca

Abstract

Modern single and multi-processor computer systems incorporate, either directly or through a LAN, a number of storage devices with diverse performance characteristics. These storage devices have to deal with workloads with unpredictable burstiness. Storage aware caching scheme— that partitions the cache among the disks, and aims at balancing the work across the disks — is necessary in this environment. Moreover, maintaining proper size for these partitions is crucial. The existing storage aware caching schemes assume linear relationship between cache size and hit ratio. But, in practice a (disk) partition may accumulate cache blocks (thus, choke the remaining disks) without increasing the hit ratio significantly. This disk choking phenomenon may degenerate the performance of the disk system. In this paper, we address this issue of disk choking and present a repartitioning framework based on the notion of marginal gains. Experimental results shows the effectiveness of our approach. We show that our scheme outperforms the existing storage-aware caching schemes while supplied with a workload showing the non-linear relationship between cache size and hit ratio.

1 Introduction

Modern computer systems interact with a broad and diverse set of storage devices. Accessing of local disks, remote file servers such as NFS [22], AFS [11], Sprite [18] and xFS [2], archival storage on tapes, read-only compact disks, and network attached disks [10] is a common phenomenon now a days. Disk arrays [20], where disks of different ages and performance parameters may be incorporated, are commonly used with an aim of reducing disk latencies. Moreover, nowadays there exist storage sites that a client can access across the Internet [15, 25]. Thus, there is a diversity of behavior and properties of the storage devices. And these characteristics of the devices will vary greatly as new storage components [6] are introduced.

Though this set of devices is disparate, one similarity is inherent among all: the time to access them is high, specially as compared to the CPU cache and the memory latency. Thus, there exists a wide gap between the performance of microprocessors and disks. To bridge this gap between microprocessors and disks, today's file systems use a file cache. A file cache is a portion of the main memory allocated by the operating system to be used for storing temporarily the frequently used disk blocks. Thus, the storage system can server a disk block without accessing the disk if the requested block is found within the file cache. This way, the file cache filters disk requests reducing the overall execution time of individual applications and increases overall system performance, often by an order of magnitude.

Though there has been substantial changes in the storage technology over the past decades, the caching architecture used by modern operating systems have remained unmodified. However, there have been some innovations in techniques, for example, incorporating application control [4], integrating file cache and virtual memory cache [18], integrating caching and prefetching [3]. But, the caching policy underwent relatively minor changes, with most operating systems employing LRU or LRU-like algorithms to decide which block to replace. The problem with these algorithms is that they are *cost oblivious*: the replacement cost is assumed to be uniform for all the cache blocks. On the contrary, these cache blocks might be fetched from devices with diverse performance characteristics. So, the assumption of uniform replacement cost is problematic in a system with multiple device types with a rich set of performance char-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

acteristics. As a simple example, consider a block fetched from a local disk as compared to one fetched from a remote, highly contended file server. In this case, the operating system should most likely prefer the block from the file server for replacement [9].

The storage aware caching in [9]—that is herein referred to as *Forney's Algorithm*— addresses this issue of caching in a heterogeneous storage environment and proposes a caching scheme based on aggregate partitioning that attempts to balance work across devices; it partitions the cache, assigns one partition to each device, and determines the partition sizes, at the end of an epoch, that lead to balanced work. Reference [7] proposes a modified scheme that maintains the partition size across the devices in a continuous fashion foregoing the notion of an epoch.

In aggregate repartitioning, each disk is assigned a partition of the cache. And the size of this partition is adjusted during the activity of the storage system. The goal of this adjustment is to balance the work across the disks.More formally, for each device, the number of cache misses times the average cost of each miss should be equal. We observe that existing aggregate repartitioning algorithms still have an inherent problem. These algorithms assume that the relationship between cache size and hit ratio is linear, and hence work across a slow disk can be increased by allocating more cache blocks to that disk. But, this relationship is not linear after a certain threshold. Hence, in practice a disk with a higher age or workload may consume blocks without increasing *cache hits* proportionately. This happens when a disk enters the saturation region where additional disk blocks can't impart significant increase in cache hits. This problem is inherent in all the caching methods based on aggregate partitioning. At a first glance, it appears that a lazy repartitioning approach may alleviate the problem. In lazy repartitioning, the algorithm doesn't reallocates blocks instantly to adjust the partitions to the desired size. On the contrary, this scheme reallocates blocks on demand. So, even if a slow disk logically derives blocks from the fast disks, the fast disk can still use the blocks that belong to the slow disk. Hence, the number of unutilized blocks in the slow disk could be reduced. But, in this approach, the allocation is one way: the fast disks only loose the blocks, but can never regain. In a system where the working set corresponding to the slow disk is larger than the cache size, this problem can easily be realized: in this scenario, the slow disk can consume the whole cache blocks choking the remaining disks.

In this paper, we address this problem of disk choking and propose a solution based on the notion of marginal utility. Here, marginal utility refers to the reduction in work performed by a disk (or, delay) with the addition of an extra cache block to the corresponding partition. The concept of using marginal utility in allocating buffer has been studied by the database community. In [19], the authors propose an approach for buffer allocation based on both the access pattern of queries and the availability of buffers during runtime. In relational database management systems, queries are issued by the clients and these queries wait in a queue before execution. As a query is selected for execution, the buffer manager examines the access pattern of the query and availability of the buffers in the buffer pool. Based on these observations the buffer manager allocates buffers to the queries. The issue of partitioning a cache among several competing disks is different from the buffer allocation among the queries. In database management system, a query runs for a short time, and the buffer allocation algorithm doesn't allocate buffers to a running query based on the performance of the query as it runs: buffers are allocated before execution. The main difference is that in heterogeneous storage environment, categorization or formulation of various access patterns is not possible. So, the marginal utility should be computed online based on the observation of the cache performance while supplied with a reference string. We propose a framework to capture the marginal utility values of the cache block. Based on this framework, we propose a technique to adjust the partition size during the system activity.

The rest of the paper is organized as follows: Section 2 presents the overview of the algorithmic space. Section 3 describes the framework and mechanism to repartition the cache. Section 4 outlines the simulation environment. Section 5 presents experimental results showing the effectiveness of the utility-based approach. Finally, section 7 concludes the paper and outlines future works.

2 Preliminaries

This section provides an overview of the algorithmic issues we explore. First, we outline the existing cost-aware algorithms based on aggregate repartitioning. Then we provide a taxonomy of aggregate partitioning. We use the terms *page* and *block* interchangeably in the subsequent pert of the paper.

2.1 Algorithms Based on Aggregate Partitioning

In a cost-oblivious caching approach, an incoming page (or block) replaces an existing page that may be anywhere in the cache. This can also be termed as *place-anywhere* approach. In a *place-anywhere algorithm* costs are recorded at a page level granularity, and a page can occupy any logical location in the cache. On the contrary, an *aggregate partitioning algorithm* divides the cache into logical partitions, and assigns a partition to a device. The algorithm maintains

performance or cost information at the granularity of partitions. As cost information is maintained for each partition, the amount of meta-data is reduced and cost information can be updated without scanning the whole cache. Moreover, this aggregate partitioning integrates well with the existing software, as cost oblivious policies can be employed for replacing individual pages within a partition.

Forney's algorithm is the first cost-aware algorithm that utilizes the notion of aggregate partitioning. It considers both static (due to diverse physical characteristics of storage media) and dynamic (due to variation of workload on disks, and network traffic) performance heterogeneity. In this approach, the cache is divided into logical partitions, where blocks within a partition are from the same device and thus share the same replacement cost. The size of each partition is varied dynamically to balance work across devices. Here, work is defined as the cumulative delay for each device. The main challenge of this algorithm is to determine the relative size of the partitions dynamically. This dynamic repartitioning algorithm basically works in two phases: in first phase, the cumulative delay for each device is determined; and in the second phase, cache partitions are adjusted. These two phases repeat cyclically.

The cumulative delay for each partition (or device) is measured over the last W successful device requests (distributed over all the devices), where W is the window size. Knowing the mean delay over all partitions and the per device cumulative *wait time*, the *relative wait time* for each device is determined.

During repartitioning, *page consumers* and *page suppliers* are identified based on relative wait times of the partitions. Page consumers are partitions that have relative wait time above a threshold T; and page suppliers are partitions having below-average wait times. Here, the threshold value is used to infer a variation in delay due to the variations in workload or device characteristics only. Moreover, the algorithm classifies each partition into one of four states: cool, warming, cooling, warm. Of these, the first one corresponds to page supplier and the rest correspond to page consumers. A page consumer increases its partition size by I pages, where I is the *base correction amount*. If a partition remains as a page consumer during subsequent epochs, the increase in partition size grows exponentially. On the other hand, the number of pages a page supplier must yield is given as:

$$\frac{IRWT_j}{\sum_{i \in suppliers} IRWT_i} \times \text{No. of consumed pages}$$

where,

$$IRWT_j = 1 - \text{relative wait time of partition } j$$

Reference [7] attempts to repartition the cache among the disks without using the notion of an epoch. This paper provides three approaches to identify the page suppliers and page consumers, and thus adjust the partition sizes during the activity of the cache system.

2.2 Taxonomy

Two basic approaches are possible for aggregate partitioning: *static* and *dynamic*. In static scheme, size of each partition is predetermined, and remain fixed during the operation of the system. However, without estimation of workload, and knowledge of miss rate as a function of cache size, it is not possible to come up with partition sizes that balance the work across devices. Thus, dynamic partitioning is necessary, which adjusts the partition sizes during the operation of the system.

Dynamic partitioning can be classified into *eager partitioning* and *lazy partitioning*. In eager partitioning, partition sizes are changed immediately whenever new partition sizes are desired. Lazy partitioning gradually changes the partition sizes on demand. In this scheme, a partition doesn't incorporate newly assigned blocks instantaneously, rather the partition claims new blocks only when it needs cache blocks to store incoming disk blocks.

3 Solution Approach

As outlined in Section 1, the existing approaches based on aggregate partitioning suffer from a significant limitation related to linear relationship assumption between cache size and hit ratio. However, this assumption is not valid: a disk can consume blocks without increasing cache hits proportionately. In this scenario, the proposed solution must track the utilization of a block within a partition. A partition can only consume a block if that partition can render better utilization of the block. We maintain that marginal utilities of the blocks within various partitions might be a suitable indicator of the utilization of a block within a partition. We outline a framework to calculate the marginal utility of the blocks within a partition, and use this marginal utility to make repartitioning decision. As illustrated in the subsequent subsection, the we maintain the marginal utility at the granularity of *cache segment* that consists of a few cache blocks. So, in this approach, the unit of repartitioning is a segment.

3.1 Marginal Utility

For a cache of size n, marginal utility of the nth cache block can be expressed as:

$$MU(s) = D(n-1) - D(n),$$

where $D(n)$ refers to delay experienced by the cache misses when the cache size is n.

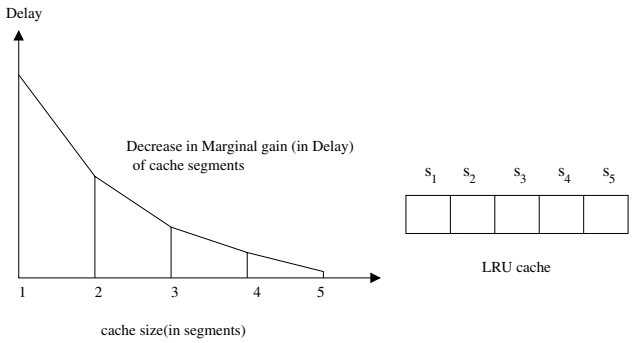

Figure 1. The LRU cache is broken into segments. Marginal utility for each segment represents the number of cache hits within the cache segment.

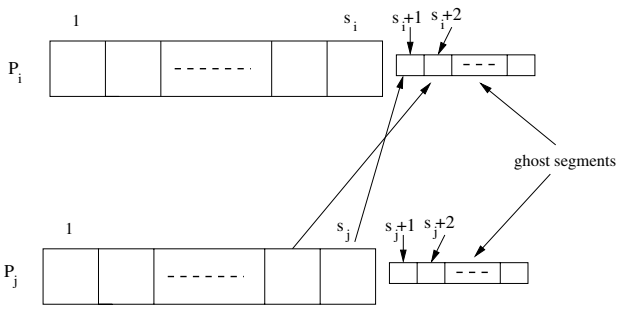

Figure 2. Partition P_i consumes the S_jth segment (and subsequent segments) of the partition P_j if those segments would render better utility in partition P_i. Once the s_jth segment is transferred to the partition P_i, this segment becomes the $s_i + 1$th segment of partition P_i. Now, the $s_i + 1$-th segment of the partition P_i is loaded with the disk blocks contained within ghost segment $s_i + 1$. Partition P_j maintains a ghost segment for the s_jth segment.

Maintaining marginal utility at block level leads to overhead in terms of processing and storage space. So, we divide a cache into segments of length n_s and maintain marginal utility for each segment. This leads to the piecewise estimation of the delay as a function of cache size. Here we assume that the cache uses the LRU replacement algorithm. Figure 1 shows the marginal utilities of the cache segments.

While accessing a block, the cache is searched for the block starting from the first location in the cache. If the block hits within a segment, the marginal utility of that segment is incremented by the access time of that block.

3.2 Repartitioning

The repartitioning scheme uses the marginal utility values while making the repartitioning decision. A partition(consumer) takes a cache segment from another partition (supplier) if the cache segment can better be utilized within the former partition. For this, a consumer should maintain the utility of the segments that are not within the current partition. So, a consumer should keep a ghost cache for the segments that lie outside the current partition size. The ghost cache doesn't keep the cache blocks; it only stores the block identifiers. So, the space overhead for the ghost cache space is very low.

3.2.1 Basic Idea

Let P_i denote partition i of size s_i in segments (s_ith segment is the last segment of partition P_i), and let P_i^s refer to the sth segment of partition P_i. At this point, partition P_i may either consume a segment at location $s_i + 1$ or supply its s_ith segment. We call the s_ith segment (that the partition will supply first) the primary *S-segment*, and the $s_i + 1$th segment (where a segment consumed from another partition will be placed) the primary *C-segment* of partition P_i. Now, a partition (P_i) can consume a segment from a partition j if

$$MU(P_i^{s_i+1}) \geq MU(P_j^{s_j}) + \delta.$$

Here, δ is a threshold that should be chosen carefully. This parameter is introduced to suppress transferring segments from one partition to another due to instantaneous variation in the workload. Here it should be noted that a partition can consume not only one segment but multiple segments (from one or more other partitions) at a time. So, this approach can quickly adapt to the variation in workload. Moreover, as the ghost segments (for a consumer) stores the identifiers of the cache blocks, those blocks can be prefetched instantaneously from the disk. This will increase the throughput of the disk system. Figure 2 shows this concept of transferring a segment to the consumer. We address issue of maintaining ghost segments at the end of this subsection.

3.2.2 Identifying Supplier and Consumers

So far we have laid out a part of the framework for repartitioning. One major issue that remains to be resolved is to decide when to make this repartitioning decision. In utility based approach the relation between a supplier and a consumer stays for a very short interval. This relation may disappear immediately after a repartitioning decision is made

As described in subsequent part this section, a partition can consume or supply multiple segments at a time. Hence, the *primary* is introduced to refer to the first segment to supply or consume

(i.e., one or more segments are transferred to a consumer). So, maintaining the supplier-consumer relationship is not conducive is this respect.

One simple approach is to take the decision at each cache miss. But, this is prohibitive as we have to scan all the partitions to decide whether there exists any consumer or supplier. Moreover, a partition may consume or supply segments after a large interval of disk activity. Another approach is to make the repartitioning decision after a certain time interval. But, this approach is also inefficient: it may perform unnecessary repartitioning task or may repartition at inopportune moment (e.g., partitions should have been adjusted long before).

Based on the above observations, we propose a repartitioning approach that minimizes the overhead and adjusts two partitions whenever it is necessary. In this approach, we try to maintain two variables max and min that refers to the partition with the maximum and minimum MU-values of the primary C-segment and S-segment, respectively . We find that maintaining the value min is not feasible. So, instead of the value min, we keep $minV$ which is the minimum MU-value of the primary S-segment among all the partitions.

Maintaining the variable max is simple. Upon a cache miss on a partition, the partition:

1 adjusts the MU-values,

2 if there is any change in the MU-value of its primary C-segment, it checks whether MU-value of its C-segment is greater than that of the partition denoted by max,and adjusts max accordingly.

So, the max value is set properly whenever there is miss. And this max value always refers to the partition with the maximum MU-value of the primary C-segment. But, the scenario is different in case of the variable min. The variable might refer to a partition that has not been accessed for a long time interval. So, the task of setting the variable min should be attributed to other active partition. For this, we maintain the variable $minV$. Now, we try to adjust the variable $minV$ only when

$$MU(P_{max}^{s_{max}+1}) - minV \geq \delta.$$

Hence, we attempt to adjust the variable $minV$ when the MU-value of the C-segment of the partition P_{max} exceeds the $minV$ by the amount δ. Note that, the above is the condition for repartitioning, but repartitioning might not be feasible at this moment as minimum MU-value of the S-segments among all the partitions might have been changed since the variable $minV$ is initialized. So, we first adjust the $minV$ value and check whether any adjustment in the partition sizes is feasible. Thus, when a miss occurs in a partition, we have to perform the following steps in addition to the two steps given earlier.

3 if there is any change in $MU(P_{max}^{s_{max}+1})$ (i.e., either max refers to the current partition or max is changed) and if $MU(P_{max}^{s_{max}+1}) - minV \geq \delta$, then initialize $minV$. Otherwise stop.

4 if $MU(P_{max}^{s_{max}+1}) - minV \geq \delta$ adjust the partition sizes.

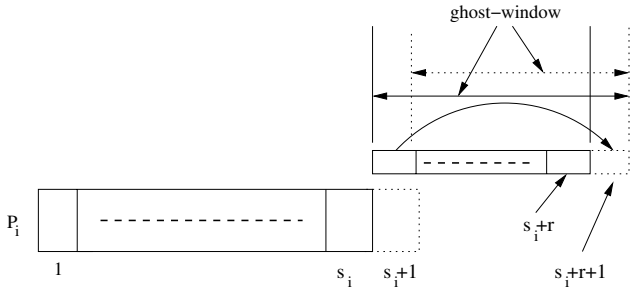

Figure 3. The rightmost ghost segment corresponds to the cache segment $s_i + r$. When the partition P_i consumes a segment $s_i + 1$, the leftmost ghost segment is allocated for the cache segment $s_i + r + 1$. The MU-value of the segment $s_i + r + 1$ is estimated (using a forward interpolation equation) based on the current values in the window. Thus, the space for the ghost window remains constant.

3.2.3 Ghost Segments

Here, it should be noted that maintaining the ghost segments for all the cache segments is not necessary. It might be sufficient to maintain a few ghost segments for each partition. We observe that increasing the number of ghost segments doesn't increase the performance, and maintaining only three ghost segments per partition renders good performance. As a partition grows, there arise the need for estimating the MU-values of a high order segment. This estimation can be done based on the MU-values of the ghost segments maintained by the algorithm, using a forward interpolation method. When a partition grows, low order segments of the ghost segments can be allocated to maintain the high order ghost segments that will newly enter into the ghost. Figure 3 shows the concept of using ghost window.

It should be noted that in this scenario partitions can't be categorized as suppliers or consumers beforehand. The intention of maintaining max and min is to track whether there develops any supplier-consumer relationship within the system.

Age (years)	Bandwidth (MB/S)	Seek time (ms)	Rotation (ms)
0	20.0	5.30	3.00
1	14.3	5.89	3.33
2	10.2	6.54	3.69
3	7.29	7.27	4.11
4	5.21	8.08	4.56
5	3.72	8.98	5.07
6	2.66	9.97	5.63
7	1.90	11.1	6.26
8	1.36	12.3	6.96
9	0.97	13.7	7.73
10	0.69	15.2	8.59

Table 1. Aging a base disk device(IBM 9LZX): The table shows the performance parameters of the same base device in different generations as the disk technology improves.

4 Evaluation Environment

This section describes our methodology for evaluating the performance of storage-aware caching. We describe our simulator and the storage environment assumed in the simulator. In Section 5, we present the results obtained using this simulator.

To measure the performance of storage-aware caching, we have implemented a trace-driven simulator. This simulator assumes a storage environment where a number of disks (of varying ages) are accessed by a single client. The client has a local cache that is partitioned across the disks. Each partition is maintained using the LRU replacement strategy. The focus of our investigation is to maintain the proper partition size dynamically. The client issues the workload for the disks.

The client workload that drives the simulator is captured using a trace file. The trace file specifies the data blocks accessed at various time points. We derive the synthetic disk traces using the PQRS algorithm proposed in [26]. This algorithm is shown to generate traces that capture the spatio-temporal burstiness and correlation in real traces [21]. We use several traces (trace 1, trace 2 and trace 3) in evaluating the performance of the caching schemes. Number of disk blocks for trace 1 and trace 2 are 1,20,000 and 1,00,000, respectively. Whereas, the number of requests for trace 1 and trace 2 are 2,00,000 and 1,80,000, respectively. The size of a disk block is taken as $8KB$. Trace 3 emulates the non-linear behavior between the cache size and hit rate. This simple trace file contains a series of sequential scans of the disk blocks.

Using these three trace files, we perform three sets of experimentations. In the first setting, we use only trace 1, and apply this trace file among each of the disks. This is similar to RAID-0 environment where a disk block is splitted across a set of disks, and each of the disks should be accessed to retrieve a block. In the second setting, we use only trace 2, and feed the reference string of the trace file on the disks in a shifted fashion. We identify equally spaced positions within the reference string, and start to feed the reference among the disks starting from these positions. In the second setting, we use only trace 1, and apply this trace file among each of the disks. This is similar to RAID-0 environment where a disk block is splitted across a set of disks, and each the disks should be accessed to retrieve a block. In the third setting, we use trace 3 and trace 2. We apply trace 3 on the slow disk, and apply trace 2 among the rest of the disks starting from the different position as described earlier.

We model the disk access time using only disk bandwidth, average seek time, average rotational latency. Hence, our disk model consider the worst case scenario. Device heterogeneity is achieved by *device aging*. As in [9], we consider a base device (IBM 9LZX) and age its performance over a range of years. A collection of disks from this set is used as the disk system in the simulator. Characteristics of the disks of various ages is shown in Table 1.

5 Experimental Results

In this section, we present a series of experimental results demonstrating the effectiveness of the proposed caching scheme. We measure the throughput obtained at the client side, and use it as the performance metric. This throughput is measured by observing the delay in retrieving the disk blocks. While measuring the delay, we consider only the delay in cache misses. Delay experienced in cache hit is comparatively negligible. We measure this throughput by varying the age of the slow disk and cache size. We perform the experimentations using three settings of the trace files as stated in section 4. Using each of the settings, we observe throughput varying the disk age and cache size. As the disk system, we consider a set of four disks. To compare our results, we use two existing storage-aware caching schemes (Forney's scheme and continuous repartitioning scheme). The δ-value is set to 500, and is observed to capture the changes in the stable behavior of a disk. For the utility based approach we maintain only three ghost segments per partition. The control values of the cache size and age of the slow disks is set to be 250 MB and 4 years, respectively. We set the segment size as the 2 percent of the total cache size.

Here it should be noted that, contrary to the reference [9], we don't model the disk request size and don't use the request locality to calculate the seek time and rotational la-

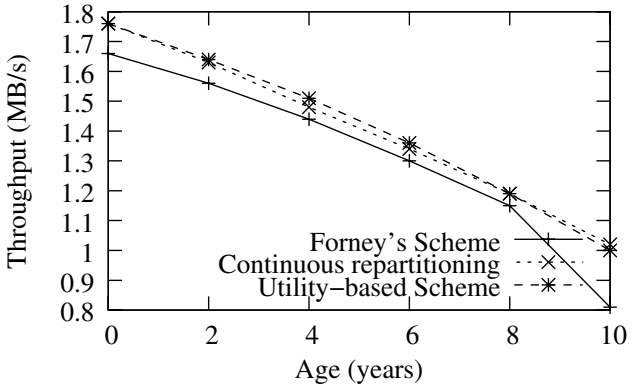

Figure 4. Overall throughput of the disk system with varying ages of the slow disk (Trace setting 1)

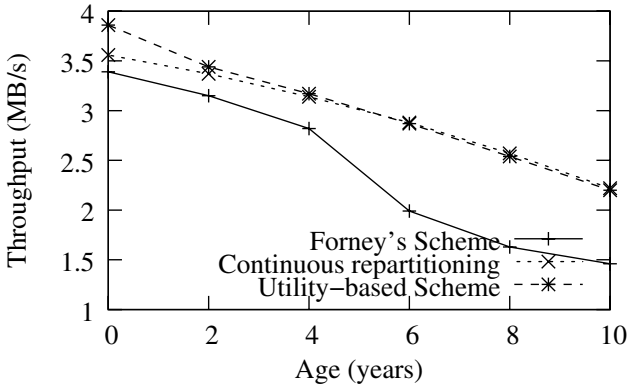

Figure 5. Overall throughput of the disk system with varying ages of the slow disk (Trace setting 2)

tency. In our simulation the request size is equal to the block size. Hence, we take a pessimistic approach and assume that one disk miss results in a delay equal to the disk access time. Whereas in the reference [9], the delay is calculated at the granularity of the request size by exploiting the locality of the requests. Here, the request size is far greater than a block size. So, these two simulation results might not be similar.

Figure 4, Figure 5 and Figure 6 show the effect of varying the age of a disk. Here, we select a particular disk and get the simulation data by aging the disk. As shown in the figure, throughput decreases with the increase in the slow disk's age. In the first two settings the performance of the continuous repartitioning and utility based scheme is almost identical, the throughput of the latter slightly dominating that of the former. However, performance of these two schemes is notably higher than the Forney's scheme.

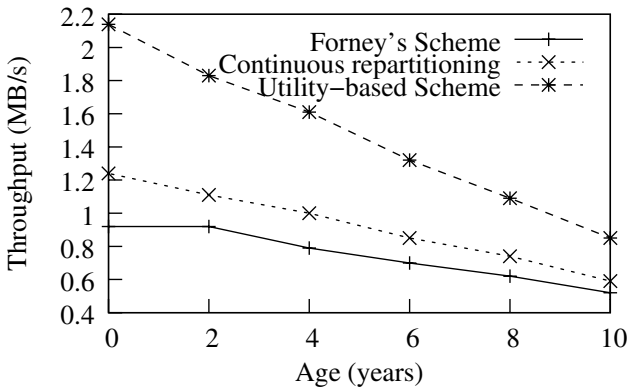

Figure 6. Overall throughput of the disk system with varying ages of the slow disk (Trace setting 3)

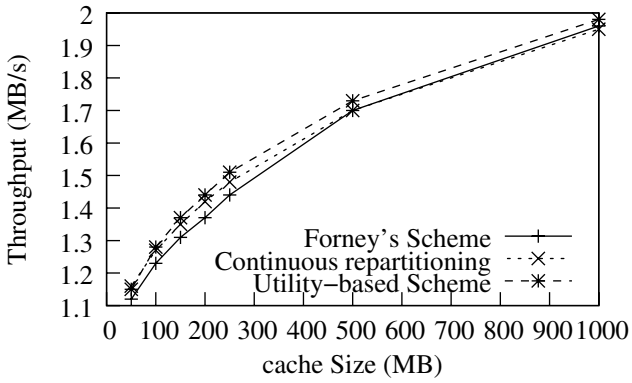

Figure 7. Overall throughput of the disk system with varying cache sizes (Trace setting 1)

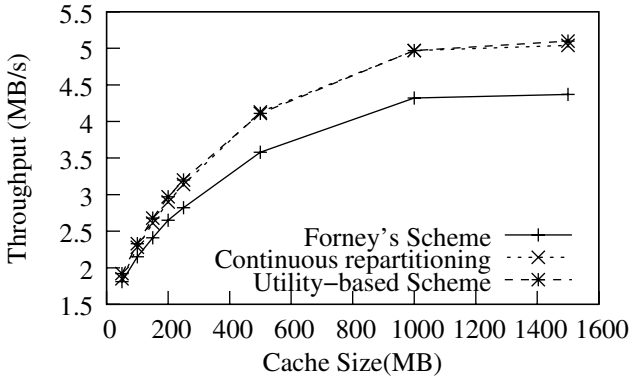

Figure 8. Overall throughput of the disk system with varying cache sizes (Trace setting 2)

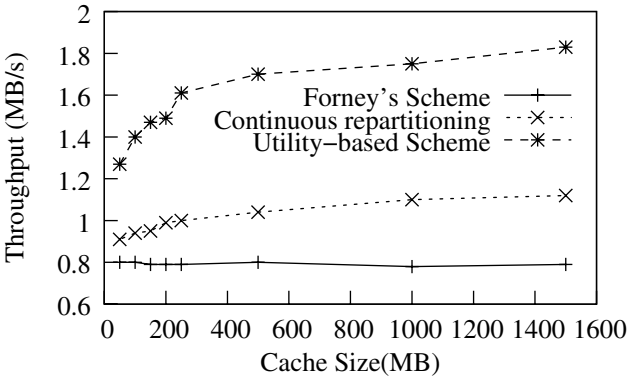

Figure 9. Overall throughput of the disk system with varying cache sizes (Trace setting 3)

Here, throughput with trace 2 is higher than that with trace 1 because of higher spatial and temporal locality. The effectiveness of the utility based scheme is evident in Figure 6. Here, the trace captures the non-linear relationship between cache size and hit ratio, and in this scenario the utility-based scheme attains significantly higher throughput than other two schemes.

The effect of varying cache size on the caching schemes is shown in Figure 7, Figure 8 and Figure 9. Here, throughput for each of the settings increases with increase in cache size. In the first two settings (Figure 7 and Figure 8), the utility-based scheme performs as good as the continuous repartitioning scheme. However, this scheme outperforms the rest of the schemes in experimental setting 3 where a trace with non-linear relationship between cache size and hit ratio is feed onto the slow disk. Here, as the cache size is increased, the throughput for the utility based scheme increases rapidly. In case of continuous repartitioning scheme, there is only a little increase in the throughput with the increase in cache size. On the other hand, in case of Forney's scheme the throughput remains almost stable with an increase in cache size.

6 Related Work

There has been works on cost-aware caching in area of web caching, main memory caching and database communities. We revisit the significant works in each of the areas.

Page replacement algorithms developed in the context of CPU or file caches don't necessarily apply to web caches. The main reason behind this is that a CPU or file cache stores fixed size blocks, and doesn't take into account the size of a document. But, a web cache uses whole document caching, and the size of the web documents vary depending on the type of the information they contain (video, audio, text, etc). Moreover, there is large variation in performance in the wide area Internet compared to the performance variation in main memory or disk storage.

In web caching, the pages can be of different sizes and costs. Hence, the scenario is different from the uniform caching where all pages have a uniform size and uniform fault cost. The *general caching problem* is more intricate than the uniform version.

In [12], Irani studies the special case of this general problem considering only the pages with varying sizes. Here, it is pointed out that Belady's rule is no longer optimal if pages and costs differ. Page replacement policies for the general caching problem is studied by Albers et al. [1]. Here, the authors classified general caching problems into four models and proposed several approximate solutions to the offline case of the problems. The theoretical computer science community has studied cost-aware algorithms as *k-server problems* [17]. Cost-aware caching falls within a restricted class of k-server problems—i.e., weighted caching. The Greedy-Dual (GD) algorithm [28] introduces variable fetch costs for pages of uniform size. The Greedy-Dual-Size(GDS) algorithm [5, 12] extends the GD to the environment with variable object size and fetch cost. *LANDLORD* [29], which is closely related to the GDS web caching algorithm in [5], is a significant algorithm in the literature. Page replacement algorithms developed in the context of web caches don't necessarily apply to storage-aware cache. The main reason behind this is that a file cache stores fixed size blocks, and doesn't take into account the size of a document. But, a web cache uses whole document caching, and the size of the web documents vary depending on the type of the information they contain (video, audio, text, etc). Moreover, there is a large variation in performance in the wide area Internet compared to the performance variation in main memory or disk storage.

In the reference [13], the authors propose a Cost-Sensitive OPTimal replacement algorithm (CSOPT) that minimizes a miss cost function in a system which has two types of miss costs: local and remote memory misses. This work is set in the context of CC-NUMA multiprocessors where local and remote misses have different costs due to the large remote-to-local memory latency ratio [16, 27]. Moreover, a remote miss always consumes interconnect bandwidth whereas a local miss can be satisfied locally. This algorithm doesn't always replace a block selected by the OPT algorithm if the block has high miss cost. Instead, CSOPT considers keeping a high cost block in the cache until it is referenced again. So, this algorithm tries to save a miss on an expensive block by trading off several misses on some cheap blocks. Hence, instead of minimizing the miss count, CSOPT minimizes the overall cost in cache misses. The size of the search tree used by the algorithm is huge which makes the algorithm unrealizable in

any practical system.

In the reference [14], the authors consider the non-uniform miss costs among the cache blocks and propose several extensions of LRU. The idea behind these extensions is to keep (if feasible) a high cost block victimized by LRU in the cache until its next reference and replace a block with low replacement cost. In such a case, the victimized block with high replacement cost is called to be in *reservation*. This idea of reservation is borrowed from the CSOPT algorithm mentioned earlier. The cost of the reserved block is deprecated over time according to various algorithms and ultimately the reservation is released.

In the reference [8], Chu and Opderbeck propose a method for varying the amount of physical memory available to a process. This method is based on the observation of page fault frequency. Partitioning the cache among multiple processes has been proposed in [24]. In this approach, an associative cache is partitioned into disjoint blocks among several processes, and the size of each partition is determined by the locality of the corresponding processes. The cache management algorithm determines the size of each partition, and dynamically adjusts the partition size. The partitioning technique is based on the method proposed by Stone, Wolf and Turek [23]. However, none of the approaches consider the storage device heterogeneity.

7 Conclusion

In this paper, we identified a problem with the caching algorithm in heterogeneous storage systems. A storage-aware caching scheme based on aggregate partitioning partitions the cache among disks. In such a scenario, supplied with a uniform workload having non-linear relationship between cache size and hit ratio, a slow disk may consume cache space choking the rest of the disks. This phenomenon of disk choking degenerates the performance of the disk system as a whole. We proposed a framework to partition the cache based on the utility of the cache blocks within a partition. Experimentations using a simple trace capturing the non-linear behavior between the cache size and hit ratio demonstrate that the utility based scheme notably outperforms other schemes. A strategy for caching disk blocks is implemented by the operating system and thus affects the performance of the computer system at a very fundamental level. Thus even a small improvement on this score assumes large significance. In our work we assume that a disk block is brought into the cache only when a miss occurs, i.e., we don't consider prefetching. As future works we like to investigate the issue of prefetching in the heterogeneous storage environment, and carry out the thorough experimentations.

References

[1] S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problem. In *In Proceedings of the Tenth Annual ACM-SIAM symposium on Discrete Algorithms*, January 1999.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Pat-terson, D. Roselli, and R. Wang. Serverless network file systems. In *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Colorado, USA, December 1995.

[3] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 188–197, May 1995.

[4] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First USENIX Conference on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[5] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 193–206, December 1997.

[6] L. R. Carley, G. R. Ganger, and D. F. Nagle. Mems-based integrated-circuit mass-storage systems. *Communications of the ACM*, 43(11):72–80, November 2000.

[7] L. Chakraborty and A. Singh. A new approach to cost-aware caching in heterogeneous storage systems. In *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High Performance Computing on Clusters (COSET-2) (Held in Conjunction with ACM international Conference on Supercomputing 2005) Cambridge Massachusetts*, pages 1–6, June 2005.

[8] W. W. Chu and H. Opderbeck. The page fault frequency replacement algorithm. In *Proceedings of the AFIS Conefference*, pages 597–608, 1972.

[9] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proceedings of the 2002 USENIX Conference on File and Storage Technology*, pages 61–74, January 2002.

[10] G. A. Gibson and R. V. Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, November 2000.

[11] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transation on Computer Systems*, 6(1):51–81, February 1988.

[12] S. Irani. Page replacement with multi-size pages and applications to web-caching. In *In Proceedings of the ACM Symposium on the Theory of Computing*, 1997.

[13] J. Jeong and M. Dubois. Optimal replacements in caches with two miss costs. In *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 155–164, June 1999.

[14] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *Proceedings of the IEEE Conference on High Performance Computing*, 2003.

[15] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[16] T. Lovett and R. Clapp. Sting: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd international Symposeum on Computer Architecte*, pages 308–317, May 1996.

[17] M. Manasse, L. McGeoch, and D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 322–333, May 1988.

[18] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Transations on Computer Systems*, 6(1), February 1988.

[19] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proceedings of the 1991 ACM Conference on Management of Data (SIGMOD)*, pages 387–396, 1991.

[20] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for reducdant arrays of inexpensive disks. In *Proceedings of the ACM SIGMOD Conference*, June 1998.

[21] C. Ruemmler and J. Wilkies. Unix disk access patterns. In *Proceedings of the Winter 1993 USENIX*, pages 405–420, January 1993.

[22] R. Sandberg. The design and implementation of the sun network file system. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, June 1985.

[23] H. S. Stone, J. L. Wolf, and J. Turek. Optimal partitioning of cache memory. *IBM Research Report RC14444*, pages 1–25, March 1989.

[24] D. Thiebaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transation on Computers*, 41(6):665–676, June 1992.

[25] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. Webos: Operating system services and widee area applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, July 1998.

[26] M. Wang, A. Ailamaki, and C. Faloutsos. Capturing the spatio-temporal behavior of real traffic data. In *Proceedings of the Symposium on Computer Performance Modeling, Measurement and Evaluation*, September 2002.

[27] W. Weber, S. Gold, P. Helland, T. Shimizu, T. Wichi, and W. Wilcke. The memory interconnect architecture: A cost-effective infrastructure for high performance servers. In *Proceedings of the 24th International Cymposeum on Computer Architecte*, pages 98–107, June 1997.

[28] N. E. Young. On-line caching as cache size varies. In *Proceedings of the Symposium on Discrete Algorithms*, 1991.

[29] N. E. Young. On-line file caching. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1999.