2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems

Continuous Retrieval of Replicated Data from Heterogeneous Storage Arrays

Nihat Altiparmak
Dept. of Computer Eng. and Computer Science
University of Louisville
Louisville, KY 40292
nihat.altiparmak@louisville.edu

Ali Şaman Tosun
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
tosun@cs.utsa.edu

***Abstract*—Replicated declustering techniques reduce response times of disk requests by distributing data among multiple disks and retrieving in parallel. Efficient retrieval of replicated data from multiple disks is a challenging problem, especially for heterogeneous storage architectures receiving continuous disk requests. Existing techniques either do not consider the heterogeneity of the disks or handle the requests in a discrete manner assuming the storage system is idle. In this paper, we focus on continuous retrieval techniques in heterogeneous storage architectures to minimize the response time of disk requests considering waiting time and service time of the requests as well as the execution time of the retrieval algorithm. We investigate multiple trade-offs between these three factors affecting the response time of a disk request and propose a maximum flow based adaptive retrieval strategy. Performance of the proposed and existing continuous retrieval techniques are evaluated using simulations driven by real world traces and various homogeneous and heterogeneous multi-disk storage configurations.**

***Keywords*—continuous retrieval, maximum flow, replicated declustering, storage arrays**

I. Introduction

Storage arrays have emerged to address the challenges of scalable storage and efficient retrieval of large datasets. Besides having hundreds of disk drives, a typical storage array includes controllers with processing units and caching memories. A storage array controller manages the data mappings to the drives, simultaneously handling fault recovery and data retrieval functionalities. Revenue for the enterprise storage array market clearly indicates the usage trend of these devices as big data challenges emerge. The total revenue for the first three quarters of 2010 was \$3.72 billion, an increase of 13% over the same period in 2009 [1]. As of the first quarter of 2013, this amount reached to \$5.5 billion [2].

There are many high-end enterprise storage arrays existing in the market [3], [4], [5], [6]. Besides the storage functionalities, these devices also include powerful processing units enabling advanced storage and retrieval strategies. For example, a single EMC Symmetrix Vmax Array has 240 disks, four Quad-core 2.33 GHz Intel Xeon processors, and supports up to 128 GB of memory [3]. It is possible to connect multiple Vmax arrays together to support up to 2400 drives and 1 TB of memory. Price of such a system starts from \$250,000 and goes up to millions of dollars. Therefore, efficient storage and retrieval strategies are necessary for the performance of such an expensive equipment.

Depending on the disk drives they include, storage arrays can be homogeneous or heterogeneous (hybrid). A homogeneous storage array is composed of identical disk drives. On the other hand, heterogeneous storage arrays include drives with different characteristics. Recent improvements in flash density led academia and industry to consider storage arrays partially or entirely based on flash technology. Several homogeneous flash arrays [7], [8], [9], [10], [11] and heterogeneous storage arrays [12], [13], [14], [15] combining magnetic and flash disks have been launched recently.

In a storage array, efficient data allocation and retrieval plays an important role for high performance parallel I/O. Declustering or striping is a common technique for efficient data allocation. Data space is partitioned into disjoint regions (buckets) and these regions are distributed among multiple disks so that requests can be retrieved in parallel. Besides efficient data distribution [16], [17], replication is also frequently applied to improve the performance and provide reliability. Many replicated declustering techniques are proposed in the literature [18], [19], [20], [21], [22], [23]. Readers are directed to [24] for an in-depth comparison and analysis of replicated declustering schemes.

A disk request is generally composed of multiple data buckets and in a storage system using replication, a replica selection mechanism is necessary for retrieval of these buckets. Retrieval algorithms and heuristics are used for this purpose. Existing retrieval techniques either do not consider the heterogeneity of the disks or do not provide a continuous retrieval strategy. For instance, *shortest-queue* is a commonly used retrieval heuristic in multimedia applications, where the disk with the shortest queue size in terms of number of buckets is selected for retrieval of a bucket among the disks including a copy of that bucket [25], [26], [27]. Such a strategy may yield competitive results in homogeneous systems; however, it obviously neglects the individual disk performances making it unsuitable for heterogeneous storage architectures. In a heterogeneous storage array, sometimes a flash based disk with a larger queue size might be a better choice than a magnetic disk with a smaller queue size.

Given a single request (set of buckets to be retrieved), maximum flow based retrieval algorithms guaranteeing the optimal *Service Time*, the time that the storage device spends while retrieving the request, are provided for homogeneous [28], [29], [30], [31] and heterogeneous [32], [33] storage arrays. Although optimal service time is an important property for a single request, realistic storage systems receive requests continuously and there are many other factors to be considered in continuous retrieval. For instance, *Execution Time* of the retrieval technique is an important factor delaying the retrieval duration. In some cases, using a heuristic based approach not guaranteeing the minimum service time might lead to a better performance than the maximum flow technique considering the low execution time of heuristic approaches. Besides the execution time and service time, an efficient continuous retrieval strategy should also consider the *Waiting Time*, the time that the request spends in storage array controller and individual disk queues. If a certain disk has a large queue wait time, sometimes retrieving a bucket from a slower disk with a shorter queue wait time might yield a better performance in heterogeneous storage architectures.

In this paper, we investigate the trade-offs mentioned above and propose an adaptive continuous retrieval strategy minimizing the *Response Time* of the disk requests by considering *Execution Time*, *Service Time*, and *Waiting Time*. We first analyse the trade-off between batching multiple requests for better load balancing and shorter service time versus retrieving them immediately for smaller waiting time. Then, we focus on the performance of different retrieval algorithms and heuristics to investigate the trade-off between the execution time of the retrieval technique and the service time of the request. Algorithms guaranteeing the optimal service time generally require larger execution time; however, heuristic based solutions do not guarantee the minimum service time. Finally, we introduce a maximum flow based adaptive continuous retrieval strategy and emphasize the advantage of adaptively rescheduling previously scheduled but unretrieved data buckets. Since these buckets are not retrieved yet, they can be combined with a newly arrived request for better load balancing and shorter service time.

The main contributions of this work are as follows:

- It theoretically proves the advantage of batching multiple disk requests on the total service time when maximum flow retrieval algorithm is in use.
- It introduces the idea of adaptively rescheduling previously scheduled but unretrieved data buckets based on disk queue sizes and disk performances.
- It provides a detailed performance analysis of the proposed and existing methods using simulations driven by real world storage traces on various homogeneous and heterogeneous storage array configurations.

II. Preliminaries

In this section, we define the terms used through the paper, provide background information on replicated declustering, and retrieval techniques.

A. Definitions

We define the terms used in this paper as follows:

- ***Execution Time:*** The time spent while the retrieval technique decides the retrieval schedule.
- ***Service Time:*** The time spent while the storage device retrieves the request.
- ***Waiting Time:*** The time a request spends in storage array controller and individual disk queues.
- ***Response Time:*** The time elapsed between the arrival and completion of a request, which is calculated as sum of *Execution Time*, *Service Time*, and *Waiting Time*.

B. Replicated Declustering

A replicated declustering of 7×7 grid using 7 disks is given in Figure 1. The grid on the left represents the first copy and the grid on the right represents the second copy. Each square denotes a data bucket and the number on the square denotes the disk that bucket is stored at. Request R_1 in Figure 1 has 6 buckets. For retrieval of 6 buckets from homogeneous disks, the best we can expect is $\lceil \frac{numOfBuckets}{numOfDisks} \rceil = \lceil \frac{6}{7} \rceil = 1$ disk access and this happens if the buckets of the request are spread to the disks in a balanced way. In most cases, this is not possible without replication [34]. When replication is used, each bucket is stored on multiple disks and a single disk should be chosen for the retrieval of each bucket. For instance, theoretically, request R_1 has an optimal retrieval cost of 1. However, since in the first copy the buckets $[0, 0]$ and $[2, 1]$ are both stored on *disk 0*, retrieval using the first copy requires 2 disk accesses. When we consider both copies, we can retrieve R_1 in 1 disk access by retrieving the bucket $[2, 1]$ from disk 5 using the second copy.

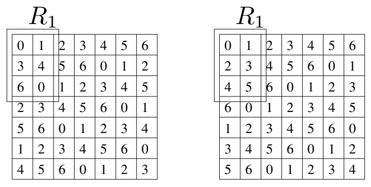


Fig. 1. Replicated Declustering

C. Retrieval Algorithms and Heuristics

Retrieval decision specifies the disk where each bucket of a request should be retrieved from. This decision is trivial if there is no replication in the system. In that case, there is only one candidate disk that a bucket can be retrieved from. However, in case of replication, retrieval algorithms/heuristics are necessary to determine the copy to be used. As plotted in Figure 2, retrieval algorithm is executed in the storage array controller and it is responsible from dispatching of the requests to the individual disk queues. Since retrieval algorithms do not perform any scheduling decision on individual disk queues, they are also referred as routing algorithms.

One way to come up with retrieval decision is the usage of a maximum flow algorithm [28]. For a given request, *maxflow* retrieval algorithm guarantees the optimal retrieval decision, i.e., the request is retrieved in the minimum *Service Time* possible. Maximum flow representation of request R_1 given in Figure 1 is provided in Figure 3. For each bucket and for each disk we create a vertex. In addition, two more vertices called source and sink are created. The source vertex s is connected to all the vertices denoting the buckets and all the vertices denoting the disks are connected to the sink vertex t. An edge is created between vertex v_i denoting bucket i and

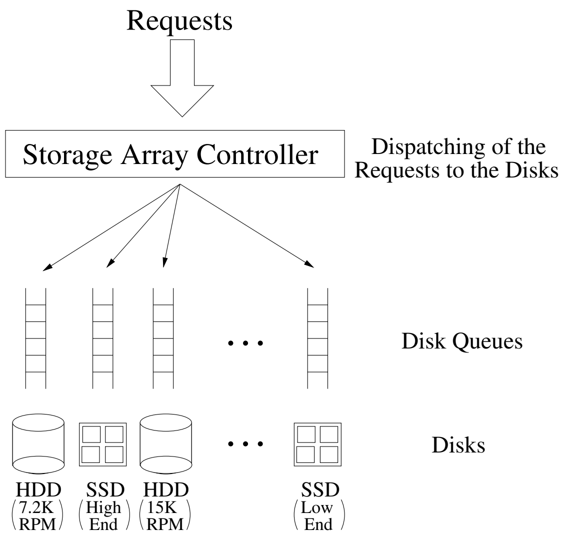


Fig. 2. Retrieval Process

vertex v_j denoting disk j if bucket i is stored on disk j. After setting the capacities of the edges appropriately, running the maximum flow algorithm will yield the optimal service time retrieval schedule. Maximum flow is shown using thick lines in Figure 3. Flow information indicates the copy to be chosen in retrieval for the optimal service time retrieval of request R_1.

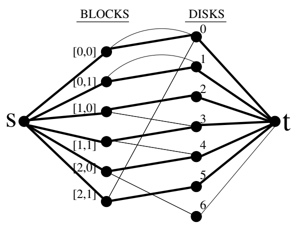


Fig. 3. Max-flow representation of request R_1

Besides the maximum flow based retrieval algorithm guaranteeing the optimal service time, different heuristic based retrieval techniques are also proposed [27], [35], [33]. We call them heuristics because they do not guarantee the minimum service time; on the other hand, they generally yield a smaller execution time. When a single request is considered, service time might be considered as a sufficient metric by itself for the performance; however, in a storage system receiving disk requests continuously, execution time and waiting time also affect the performance dramatically. Considering all of these factors, various optimizations techniques can be applied to improve the performance of multi disk storage systems.

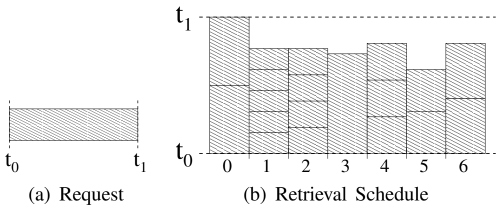
III. Continuous Retrieval Schemes

In this section, we discuss various continuous retrieval techniques that can be used together with a retrieval algorithm to reduce the response time of disk requests.

A. *Batching*

Batching is a common technique used in request/job scheduling [36], [37], [38]. In this technique, when a new request arrives, we first check the state of the storage system. If the storage system is idle, we determine the retrieval schedule of the request immediately. However, if the storage system is busy with retrieving the buckets of previous requests, then we batch the incoming requests until the storage system becomes idle again. When the storage system becomes idle, we determine the retrieval schedule of the batched request at once.

The idea of batching as a continuous retrieval technique is explained using Figures 4 and 5. Figure 4(a) represents a single request arriving at time t_0 with the optimal service time of $(t_1 - t_0)$. Figure 4(b) illustrates the retrieval schedule of this request. Numbers on the x-axis represent the disks, rectangles on a disk represent the buckets to be retrieved from that disk, and the y-axis represents the time. Assuming that the optimal retrieval schedule is as in Figure 4(b), then the optimal service time is t_1 and it is determined by the disk having the largest retrieval time; *disk 0* in this case.



(a) Request (b) Retrieval Schedule

Fig. 4. Request Representation for Batching

Based on the representation provided in Figure 4, an example of the batching idea is presented in Figure 5. Assume that the storage system is idle at time t_0. Since the storage system is idle, retrieval schedule of $R1$ is calculated and it is dispatched to the individual disk queues immediately. Assume also that the retrieval of $R1$ will be completed at time t_4. Since the storage system is busy until time t_4, all the requests arriving until t_4 are batched. At time t_4, retrieval schedule of the batched request ($R2+R3+R4$) is calculated at once, in a single run of a retrieval algorithm. The idea of batching is especially useful for the maximum flow based retrieval algorithm since it guarantees the optimal service time for a given request. Now, let us theoretically investigate the advantage of batching multiple requests on the total service time for the maximum flow retrieval algorithm.

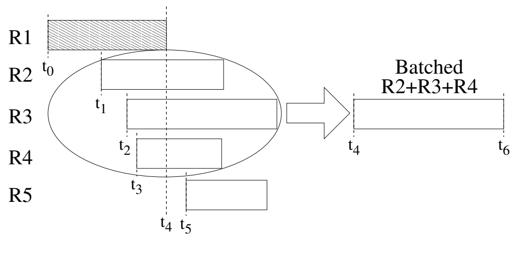


Fig. 5. *Batching* Example

Assume that the storage system receives two requests R_1 and R_2 at the same time, each representing a set of buckets to be retrieved. Let S_1 and S_2 be individually calculated optimal service times of R_1 and R_2 respectively, and S_{ind} be the total service time of retrieving R_1 and R_2 using their individually calculated optimal service time schedules. Then, we can state the following proposition:

Proposition 1: $Max(S_1, S_2) \leq S_{ind} \leq (S_1 + S_2)$.

The minimum value of $S_{ind} = Max(S_1, S_2)$ is achieved if the disk that causes the largest service time in R_1 is not used in R_2 or vice versa. On the other hand, the maximum value of $S_{ind} = (S_1 + S_2)$ is achieved if and only if the largest service time is caused by the same disk in both of the requests. These can be better visualized by considering the representation in Figure 4(b) for both S_1 and S_2 individually, and combination of those two figures for S_{ind}. In other words, we are lucky if S_2 uses the disks that S_1 does not use or vice versa. Therefore, when we calculate the optimal service time schedules individually, we do not have any control over where S_{ind} falls within the range given in Proposition 1. On the other hand, when we batch R_1 and R_2, the batched request can be represented as a single request $R_{bat} = R_1 \cup R_2$.

Lemma 1: Assume that $R_{bat1} = R_1 \cup R_2$ represents the batched request and $R_1 \cap R_2 = \emptyset$ and S_{bat1} represents the optimal service time of R_{bat1}. Then $S_{bat1} \leq S_{ind}$.

Proof: Here, R_{bat1} represents the batched request of R_1 and R_2 and these requests do not include any common bucket, i.e. $|R_{bat1}| = |R_1| + |R_2|$. Lemma 1 follows since the Max-flow retrieval guarantees the optimal service time for a set of buckets to be retrieved. In other words, for the same set of buckets, a service time less than S_{bat1} cannot be achieved. ■

Lemma 1 proves that even if R_1 and R_2 are disjoint sets, batching will result in a service time at least as good as retrieving these requests individually. Now consider another request R_2' which is created by substituting one or more buckets of R_2 with R_1's buckets so that $R_1 \cap R_2' \neq \emptyset$. In this case, batching has more chance to yield a smaller service time as stated in the following lemma:

Lemma 2: Assume that $R_{bat2} = R_1 \cup R_2'$ represents the batched request and $R_1 \cap R_2' \neq \emptyset$ and S_{bat2} represents the optimal service time of R_{bat2}. Then $S_{bat2} \leq S_{bat1}$.

Proof: In this case, R_{bat2} represents the batched request of R_1 and R_2', and these requests include at least one common bucket i.e. $|R_{bat2}| < |R_1| + |R_2'|$. The proof is based on the fact that the maximum value of a set of numbers cannot be increased by taking some numbers out of the set. By the set theory, we know that $|R_1 \cup R_2'| = |R_1| + |R_2'| - |R_1 \cap R_2'|$. Since $R_1 \cap R_2' \neq \emptyset$, then we can state that $|R_{bat2}| < |R_{bat1}|$ and $R_{bat2} \subset R_{bat1}$. In other words, S_{bat2} is achieved by subtracting some buckets from S_{bat1}. ■

Lemma 2 can be visualized by considering Figure 4(b) and removing some buckets from the top of some disks. Clearly, by removing a bucket, we cannot achieve an optimal service time greater than t_1; however, removing a bucket from *disk 0* guarantees achieving an optimal service time less than t_1. Lemmas 1 and 2 allow us to state the following theorem:

Theorem 1: Optimal service time of a batched request is always less than or equal to the total service time achieved by retrieving these requests using their individual optimal service time schedules.

Although batching multiple requests has an advantage in terms of total service time, an extra waiting time will be introduced to the requests if some disks to be used in retrieval are idle during this batching process. For instance, request $R2$ in Figure 5 waits $(t_4 - t_1)$ time, $R3$ waits $(t_4 - t_2)$ time, and $R4$ waits $(t_4 - t_3)$ time during the batching process. Note that if all the disks are busy while batching, then this extra waiting time might not have any negative effect on the response time.

In addition to the possible waiting time disadvantage, batch size may continuously increase if no upper limit is set especially when the system is overloaded. Such an increase in the batch size will cause an increase in the execution time of the retrieval algorithm if the algorithm does not have a sub-linear time complexity. In order to eliminate excessive batch sizes for polynomial time algorithms like the maximum flow technique, an appropriate upper limit should be set for the maximum batch size depending on the system parameters. When this limit is reached, currently batched request should automatically be processed.

B. *Immediate-conservative*

The second continuous retrieval technique we consider is retrieving the requests as soon as they arrive in order to eliminate the extra waiting time introduced by the batching technique. Although immediately scheduling eliminates this additional waiting time, it will introduce a larger service time if the requests are scheduled in a discrete way assuming zero disk queue waiting times. One way to improve the service time in this case is considering the disk slack times integrated into the retrieval algorithm. Since the queue length and performance (access/transfer times) of the individual disks are known to the system, it is possible to determine the load of each disk at any given time by checking its queue size. By incorporating this initial load information into the retrieval algorithm, we can improve the service time through better load balancing.

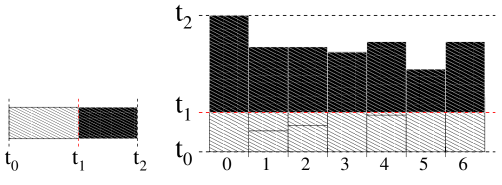


Fig. 6. Request representation for *immediate-conservative*

Immediate-conservative continuous retrieval technique is explained using Figures 6 and 7. Similar to Figure 4, Figure 6(a) represents a single request with the optimal service time of $(t_2 - t_0)$ and Figure 6(b) illustrates the retrieval schedule of this request. Different than Figure 4, black rectangles in Figure 6 represent the initial loads of the disks at time t_1. Based on the representation provided in Figure 6, an example of immediate-conservative technique is presented in Figure 7.

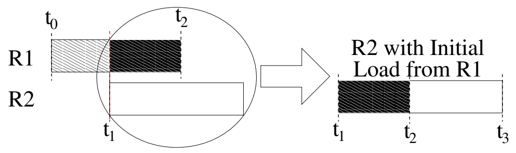
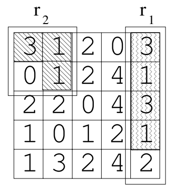


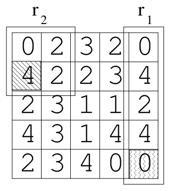
Fig. 7. *Immediate-conservative* example

Assume that the request R_1 is already scheduled at time t_0 and a new request R_2 arrives at time t_1. Using the *immediate-conservative* technique, we can immediately determine the initial load of each disk at time t_1 and provide this information to the retrieval algorithm together with the new request R_2. By this way, we eliminate the extra waiting time that the batching technique introduces for R_2. Besides, since the retrieval schedule is determined immediately considering the slack times of the disks, there is a possibility to retrieve some requests *apriori*, before completing the retrieval of previously arrived requests. An illustration of the *apriori* retrieval concept is provided in Example 1.

Example 1: Consider the allocation scheme presented in Figure 8 for a 5×5 grid using a total of 5 disks. Assume that there are two copies for each data bucket. Figure 8(a) denotes the allocation scheme for the first copy and Figure 8(b) denotes the allocation scheme for the second copy. For each allocation scheme, a square denotes a data bucket and the number on a square denotes the disk that data bucket is stored at.



(a) Copy 1



(b) Copy 2

Fig. 8. Data allocation for Example 1

Request r_1 is composed of 5 data buckets as shown in Figure 8. Optimal service time retrieval of r_1 is illustrated using vertical zigzag patterned blocks in both Figure 8 and Figure 9. The x-axis in Figure 9 represents the disks, the numbers below the disks represent their retrieval costs, and the y-axis represents the time. Request r_1 arrives at time 8 msec and the retrieval of r_1 is completed at time 16.3 msec. At time 10 msec, storage system receives another request r_2, which is composed of 4 data buckets as shown in Figure 8. Immediately retrieving r_2 using the initial load information of r_1 results in an optimal service time of 6.1 msec. Therefore, retrieval of r_2 is completed at time 16.1 msec. In this case, since r_2 is fully retrieved before the previous request r_1 is fully retrieved; r_2 is considered as an *apriori* retrieval, i.e., idle disks are exploited while r_1 is being processed.

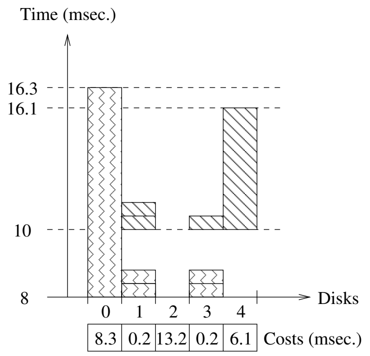
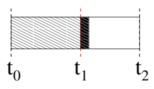


Fig. 9. Data retrieval for Example 1

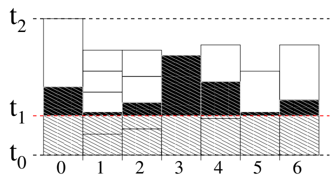
Note that considering the initial load information has a crucial effect on retrieving a request *apriori* with no or minimal waiting. If the retrieval algorithm did not know that *disk 0* had a large initial load at time 10, then it might have scheduled bucket $[0, 0]$ to *disk 0* resulting in a larger retrieval time. Although *immediate-conservative* eliminates the extra waiting time of the batching idea and introduces the notion of an *apriori* retrieval, it is still expected to yield a larger total service time than the batching technique as stated in Theorem 1. *Immediate-conservative* technique is aware of disk slack times; however, its performance can be improved even further by using an adaptive retrieval strategy.

C. Immediate-adaptive

As an improvement on *immediate-conservative*, we propose *immediate-adaptive* that allows rescheduling of the previously scheduled but unretrieved buckets together with a new request. Figure 10(a) represents a single request, black rectangles represent the initial load and white rectangles represent the scheduled but unretrieved buckets at time t_1. Retrieval schedule of this request is given in Figure 10(b) showing initial loads and unretrieved buckets of each disk.



(a) Request



(b) Disks

Fig. 10. Request representation for *immediate-adaptive*

Based on the representation given in Figure 10, an example of *immediate-adaptive* is presented in Figure 11. Assume that the request R_1 is already scheduled at time t_0 and a new request R_2 arrives at time t_1. Using the *immediate-adaptive* technique, in addition to determining the initial loads of each disk as in the *immediate-conservative* case; we can also determine the unretrieved buckets of each disk by checking the related disk queue. Then, these unretrieved buckets can be combined with the buckets of the new request R_2. Modified R_2 together with the initial load information will provide more flexibility to the retrieval algorithm. By this way, we can reschedule the buckets of the previous requests considering the buckets of the new request and the disk slack times.

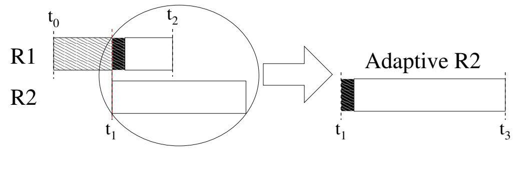


Fig. 11. *Immediate-adaptive* example

Note that this technique is not expected to provide any benefit to the heuristic based solutions since they perform the retrieval decision in an online fashion without considering the big picture. However, it is expected to decrease the response time of requests when the maximum flow retrieval algorithm is in use based on Theorem 1, especially if the request sizes are large and the request interarrival times are low. Low request interarrival times with large request sizes will enable more buckets to be added to the current request from the previous requests. By this way, more flexibility will be provided to the retrieval algorithm.

However, this technique might also cause starvation depending on the disk scheduling algorithm used in the disk queues. In other words, retrieval of some buckets might continuously be delayed causing very large response times for the requests including these buckets. Starvation will automatically be eliminated if first-come, first-served (FCFS) scheduling is used in the disk queues. However, even with other scheduling algorithms, an upper limit on the amount of time to reschedule a bucket can be set to eliminate possible starvation issues.

IV. Evaluation

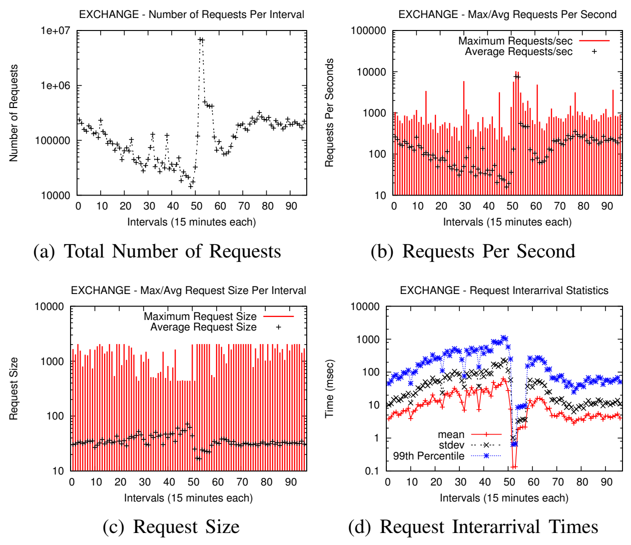
In this section, we evaluate the performance of the proposed continuous retrieval techniques using simulations driven by real storage traces and considering execution times, waiting times and service times of the requests.

A. Traces

We used three commonly used multi-disk storage system traces previously used in various storage related studies [39], [40], [41]. These traces are publicly distributed via the online trace repository provided by the Storage Networking Industry Association (SNIA) [42].

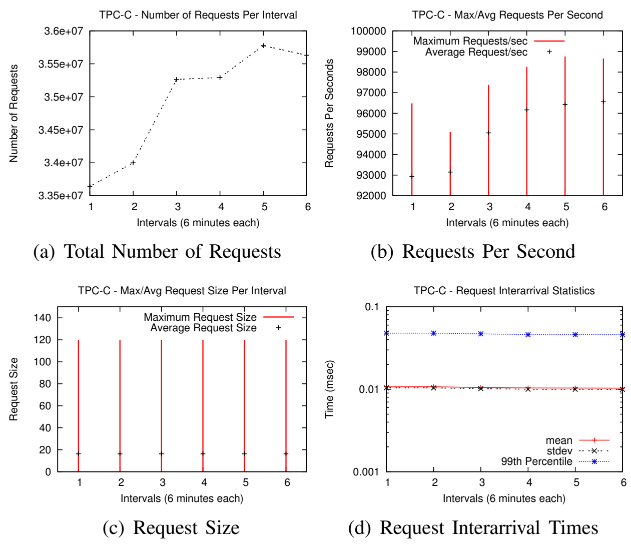
- ***Exchange***: The first workload we use is the Exchange workload, which is taken from a server running Microsoft Exchange 2007 inside Microsoft [43]. It is a mail server for 5000 corporate users consisting of around 40 million read requests. The trace covers a 24-hour weekday period starting at 2:39pm on the 12th of December, 2007 and it is broken into 96 intervals of 15 minutes each.
- ***TPC-C***: The second workload we use is TPC-C, which is an online transaction processing (OLTP) benchmark simulating an order-entry environment [44]. It is a mix of five concurrent transactions of different complexities. The TPC-C trace covers 36 minutes of workload and consists of around 210 million read requests. The trace is taken on 26th of February, 2008 and it is broken into 6 intervals of 6 minutes each.
- ***TPC-E***: The third workload we use is TPC-E, which is another OLTP benchmark simulating the workload of a brokerage firm [45]. TPC-E is the successor of TPC-C, its transactions are more complex than those of TPC-C, and they more closely resemble modern OLTP transactions. The TPC-E trace covers 84 minutes of workload consisting around 101 million read requests. The trace is taken on 18th of October, 2007 and it is broken into 6 intervals of 10 to 16 minutes each.

Exchange trace statistics are provided in Figure 12 for each trace interval shown on the x-axis. Figure 12(a) shows the number of requests per interval. The minimum number of requests is performed in interval 48 with around 15000 requests and the maximum number of requests is performed in



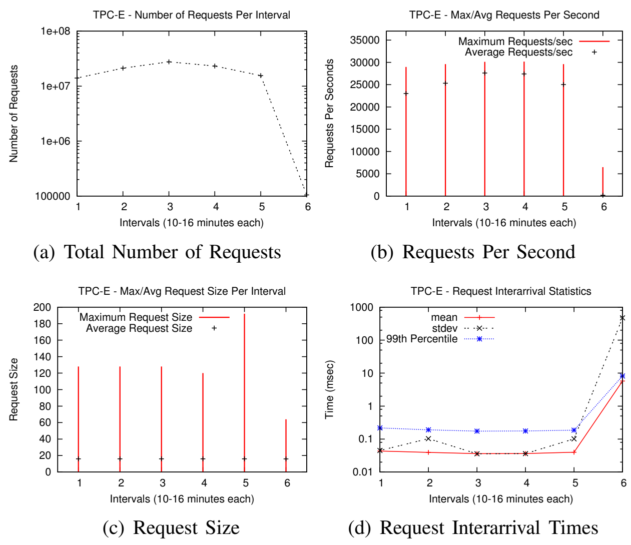
(a) Total Number of Requests (b) Requests Per Second (c) Request Size (d) Request Interarrival Times

Fig. 12. Exchange Trace Statistics



(a) Total Number of Requests (b) Requests Per Second (c) Request Size (d) Request Interarrival Times

Fig. 13. TPC-C Trace Statistics



(a) Total Number of Requests (b) Requests Per Second (c) Request Size (d) Request Interarrival Times

Fig. 14. TPC-E Trace Statistics

interval 52 with around 7 million requests. Figure 12(b) shows the average and the maximum number of requests performed in one second for each interval. Figure 12(c) plots the average and the maximum request size per interval. By the request size, we mean the number of buckets requested in one request. Finally, Figure 12(d) shows the statistics for request interarrival times per interval, which is a crucial factor for the *immediate-adaptive* approach. Similar statistics for TPC-C and TPC-E are also provided in Figures 13 and 14 respectively. In order to keep up with the request arrival rates and the request sizes of these traces, we simulated the Exchange trace using 100 disks and the TPC-C and the TPC-E traces using 1000 disks.

B. Storage Configurations and Parameters

We performed simulations using five different disk models; three hard-disk drives (HDD) with different revolutions per minute (RPM) and two solid-state disk (SSD), one high-end and one low-end. Specifications of the disks are provided in Table I. All values except the *Average Access Time* value are obtained from the factory specifications of the related disk. *Average Access Time* is the average time spent to reach a data bucket in a disk and we calculated it experimentally running a read only benchmark on the real disks. Since the *Average Access Time* value should roughly be the sum of *Seek Time* and *Latency*, factory specifications of the disks seem to be consistent with the values we calculated experimentally.

TABLE I. DISKS

Producer	Model	Type	RPM	Seek T.	Latency	Bandwidth	Avg. Access Time
Seagate	Barracuda	HDD	7.2 K	8.5 ms	4.1 ms	57 MB/s	13.2 ms
WD	Raptor	HDD	10 K	4.2 ms	5.5 ms	68 MB/s	8.3 ms
Seagate	Cheetah	HDD	15 K	3.6 ms	2.0 ms	86 MB/s	6.1 ms
OCZ	Vertex	SSD	-	-	0.1 ms	197 MB/s	0.5 ms
Intel	X25-E	SSD	-	-	0.07 ms	250 MB/s	0.2 ms

In order to calculate the *Service Time* in our simulations, we need to know the average time it takes to retrieve a bucket from a disk. Therefore, we should consider both the *Average Access Time* and the transfer time of a bucket. In our simulations, we assumed that the bucket size is equal to the filesystem block size (4KB is the default value in *ext2/ext3/ext4*). Using 4KB for each bucket, the transfer time of a bucket can be calculated using the average *Bandwidth* value provided in the table. The transfer time of a Barracuda HDD is calculated as 0.068 milliseconds and the transfer time of an X25-E SSD is calculated as 0.015 milliseconds. These values indicate that the average access time value is the dominating factor in the retrieval time calculation. Therefore, we believe that the average access time is a good approximation by itself to be used for the *Service Time* calculation in our simulations. For larger bucket sizes, transfer time can also be incorporated into the retrieval time calculation easily.

Using the disks provided in Table I, we created four different homogeneous and heterogeneous multi-disk storage architectures. Table II provides these configurations. *disk_conf1* and *disk_conf2* represent homogeneous storage architectures, the former using the slowest disk (Barracuda HDD 7.2K RPM) of Table I and the latter using the fastest disk (Intel X25-E SSD) of Table I. *disk_conf3* and *disk_conf4* represent heterogeneous storage architectures. In *disk_conf3*, disks are chosen with equal probabilities. In *disk_conf4*, extra 10% of the data is randomly placed in a fast SSD so that it can act as a cache in front of the HDDs. Note that no cache replacement policy is applied in *disk_conf4* in order to evaluate the proposed algorithms fairly. Instead, we used a static caching strategy where the cached buckets remain constant during the experiment period.

TABLE II. STORAGE CONFIGURATIONS

Storage Config.	Barracuda	Raptor	Cheetah	Vertex	X25-E
disk_conf1	100%	-	-	-	-
disk_conf2	-	-	-	-	100%
disk_conf3	20%	20%	20%	20%	20%
disk_conf4	33.3%	33.3%	33.3%	-	10%

C. Allocation Scheme and Query Type

Storage traces disclose which bucket is retrieved from which disk; however, they do not provide any information on where the other copies of the requested buckets reside. For this reason, we assume that Random Duplicate Allocation (RDA) [46] is used to distribute the copies of a data bucket to the disks. RDA stores a bucket into c disks chosen randomly from the set of disks used in the system. We have conducted simulations for the copy sizes (c) of 2, 3, 4, and 5. Since the data is distributed to the disks randomly, we also used queries of arbitrary shape using the query size information retrieved from the trace.

D. Algorithms and Heuristics

The following algorithms and heuristics are implemented for the experiments:

- ***random*** is a heuristic based on a complete random selection. Retrieval of a bucket is performed from the disk randomly chosen among the pool of the disks having a copy of the bucket.
- ***shortest-queue*** is another heuristic first proposed in [27] and also used in [25], [26] as a retrieval technique. To make the retrieval decision of a bucket, first all the disks carrying a copy of the bucket is determined and then the disk with the smallest number of buckets in its queue is selected.
- ***power-of-two-choices*** is a heuristic based solution inspired by [35] for the retrieval of replicated data. In order to retrieve a data bucket, two candidate disks are randomly selected among the pool of the disks having a copy of the bucket. Retrieval is performed from the disk resulting in the shortest retrieval time. The disk resulting in the shortest retrieval time can be determined considering the number of buckets in the disk queue and the *Average Access Time* of the particular disk.
- ***online*** is a heuristic achieved by generalizing the *power-of-two-choices* to all disks [33]. Instead of two random disks, all the disks carrying a copy of the bucket to be retrieved are checked and the disk yielding the shortest retrieval time is selected. Note that *shortest-queue* is expected to perform similar to *online* in homogeneous storage architectures since access/transfer times of the disks are equivalent for homogeneous disks.

- ***max-flow*** is a network flow based retrieval algorithm guaranteeing the minimum service time. We implemented the integrated technique proposed in [32] using Goldberg's *hi_pr* implementation for the maximum flow calculation. *hi_pr* is based on push-relabel method [47] and currently is the fastest sequential implementation available that we are aware of. It has the time complexity of $O(|V^2|\sqrt{|E|})$ for a graph having $|V|$ vertices and $|E|$ edges [48]. Note that among these implementations, *maxflow* is the only technique guaranteeing the minimum service time.

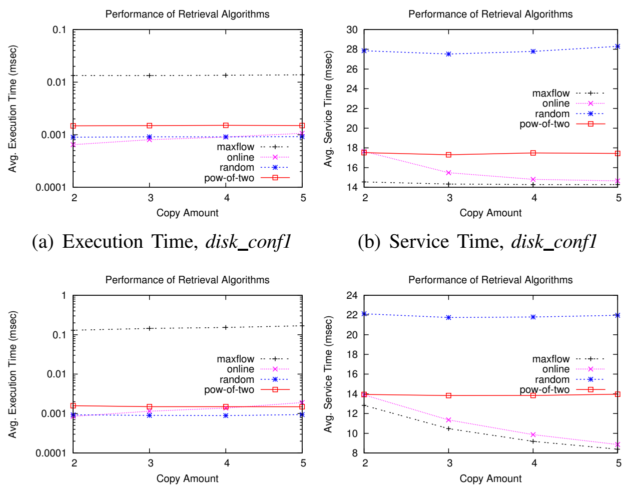
We implemented the algorithms and heuristics described above together with the proposed continuous retrieval techniques in C programming language and compiled using gcc optimization level 3 (-O3). The machine we used for simulation has two Intel Xeon X5672 quad-core processors with total of 8 cores and 32GB of main memory. It runs Ubuntu 10.04.03 LTS operating system and each core has 3.2 GHz clock speed. In experiments, we used a single core only.

E. Results

We perform the evaluation in two steps. First, we show that *online* clearly outperforms *random* and *power-of-two-choices*. Next, we compare the performance of continuous retrieval techniques using *online*, *shortest-queue*, and *max-flow*.

1) Performance of Retrieval Heuristics: In this section, we look at the performance of *online*, *random*, and *power-of-two-choices* heuristics, and compare their performance with *maxflow*. Our aim here is to investigate the trade-off between the execution time of the retrieval technique and the service time of the request. In order to make the comparison without the effect of continuous retrieval techniques, we handled the requests in a discrete way. In other words, we found the retrieval decision of every disk request individually assuming that all the disks are idle for every request. This was necessary since continuous retrieval techniques could result in different batch sizes for different retrieval algorithm/heuristic. We simulated the storage traces and calculated the average service time and average execution time values for different storage configurations. Results are shown in Figure 15 for the Exchange trace, where x-axis shows the copy amount and y-axis shows either execution time or service time. We do not show the results for the TPC-C and TPC-E traces since they behave similarly.

As it can be seen from Figures 15(a) and 15(c), execution time of *maxflow* is clearly larger than the heuristics while the execution time of heuristics are very close to each other. Beside this, execution time of *maxflow* increases when the storage system becomes heterogeneous. This is expected because the retrieval decision gets harder and the incrementation steps of *maxflow* increases when the storage system becomes heterogeneous. Nonetheless, average execution time difference between *maxflow* and heuristics is always less than 0.2 milliseconds per request. On the other hand, *maxflow* guarantees the optimal service time and it yields equal or smaller service time than the heuristic based solutions. This can clearly be observed from Figures 15(b) and 15(d). Among the heuristics, *online* clearly performs the best in terms of service time and its performance gets closer to *maxflow* when the number of copies



(a) Execution Time, *disk_conf1* (b) Service Time, *disk_conf1*

(c) Execution Time, *disk_conf4* (d) Service Time, *disk_conf4*

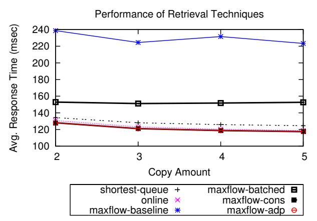
Fig. 15. Discrete, Exchange

increase. Since *online* makes the retrieval decision considering the buckets one by one in an online fashion, probability of making a mistake for *online* decreases when the number of copies increases. Nevertheless, *maxflow* performs a couple of milliseconds better than *online* per disk request.

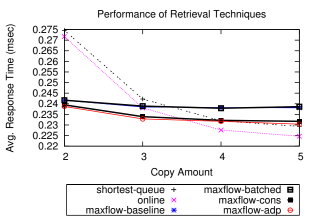
These results clearly show the trade-off between service time and execution time, especially among *maxflow* and *online*; however, inclusion of waiting time is also expected to effect the values of service time and execution time, as well as the response time. Therefore, we next look at the response time values in continuous retrieval case and observe how these trade-offs affect the response time of disk requests.

2) Performance of Continuous Retrieval Schemes: In this section, we evaluate the performance of the proposed continuous retrieval techniques (*batched*, *immediate-conservative*, and *immediate-adaptive*) using *online*, *shortest-queue*, and *maxflow* retrieval algorithms. Since *online* and *shortest-queue* cannot get any benefit from batching or adaptively rescheduling by their nature, they are only integrated into the immediate-conservative technique. On the other hand, *maxflow* is integrated into all three continuous retrieval techniques (*maxflow-batched*, *maxflow-cons*, *maxflow-adp*). Beside these, we also compare the results with *maxflow-baseline*. *maxflow-baseline* is used as a baseline comparison and it outlines what would happen if we did not use any continuous retrieval technique. *maxflow-baseline* handles the request in a discrete manner as in Section IV-E1 by assuming the disks are idle in each case.

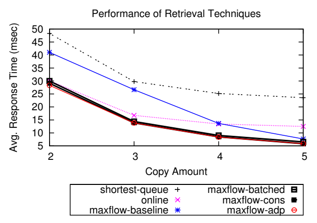
Figure 16 shows copy amount vs. average response time values for the Exchange trace using the four different disk configurations. Although batching has a service time advantage, because of the waiting times introduced during the batching process, it generally performs worse than the immediate techniques. This shows that while the batching technique awaits requests in the storage array controller, immediate techniques can retrieve some buckets *apriori* by utilizing the idle disks as explained in Example 1. Expectedly, *maxflow-baseline* generally yields larger response time values compared to others since it is not using any continuous retrieval technique. It is also clear that the performance of *shortest-queue* degrades as the system gets heterogeneous. *online* performs competitive



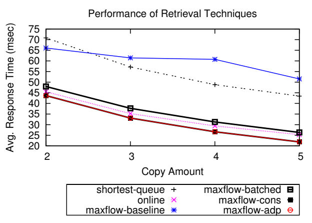
(a) *disk_conf1*



(b) *disk_conf2*

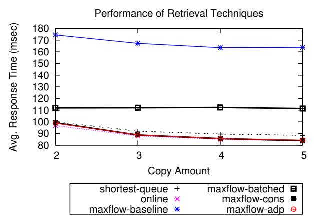


(c) *disk_conf3*

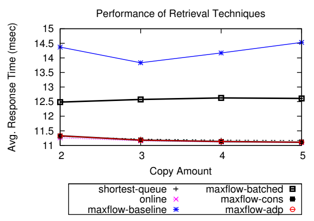


(d) *disk_conf4*

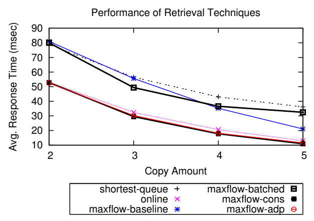
Fig. 16. Continuous, Exchange



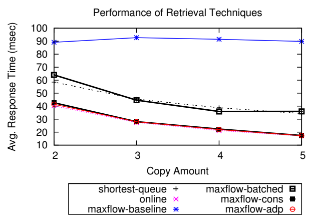
(a) TPC-E, *disk_conf1*



(b) TPC-E, *disk_conf2*



(c) TPC-C, *disk_conf3*



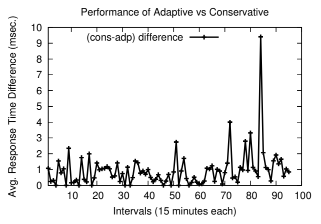
(d) TPC-E, *disk_conf4*

Fig. 17. TPC-C and TPC-E, Continuous Requests

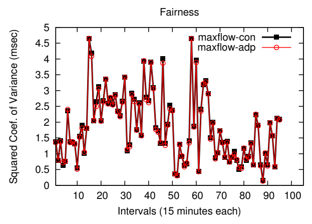
with *maxflow-cons* and *maxflow-adp* but generally performs slightly worse than them. For *disk_conf2*, performances are very close since *disk_conf2* is a homogeneous configuration composed of high-end SSDs. In this configuration, all the disks are very fast and request interarrival times of the *Exchange* trace are larger compared to the disk retrieval times (see Figure 12(d)). Therefore, the retrieval choice does not really affect the response time a lot.

Figure 17 shows copy amount vs. average response time values for the TPC-C and the TPC-E traces using one homogeneous and one heterogeneous disk configuration for each trace. Most of the observations pointed out for the Exchange trace in Figure 16 applies to the TPC-C and TPC-E traces. One difference here is that *online* generally performs close to *maxflow*. This happens because the number of disks in the system are generally larger than the request size of these traces. In such a case, even the buckets are scheduled to the disks in an online fashion, possibility of making an error for the *online* is very small if a fair declustering scheme like RDA is used as in our case.

maxflow-cons and *maxflow-adp* performs generally better than the other techniques and their performances are also very similar to each other by *maxflow-adp* being slightly better in some cases. This makes sense because *maxflow-adp* allows the retrieval decision of the previous requests to be changed for only better performance and it is expected to help if there are unretrieved buckets from previous requests when a new request arrives. Only in such a case *maxflow-adp* can adaptively reschedule the previously scheduled buckets. The positive effect of this rescheduling is mostly observed for heterogeneous storage configurations like *disk_conf3*. Figure 18(a) shows the average response time difference between *maxflow-cons* and *maxflow-adp* ($maxflow\text{-}cons - maxflow\text{-}adp$) for different intervals of the Exchange trace. As it is clear from the figure, adaptive technique improves average response time of the conservative technique up to an average of 9-10 milliseconds per request. Such a difference would impact the performance of the storage system tremendously, especially in rush hours where the system is overloaded. Beside this, adaptive technique does not introduce any additional execution time on the response time of a request when it is carefully implemented. An important implementation detail is not to reschedule the buckets whose remaining queue wait times are very short such that the execution time of the rescheduling process is larger than their remaining queue wait times. If these buckets are not included in the rescheduling process, no additional execution time overhead is introduced by the adaptive retrieval since these buckets are already waiting on the disk queues.



(a) *Avg. Response Time Difference*



(b) *Fairness*

Fig. 18. Adaptive vs Conservative, copy amount (c) = 2

Another issue that should be addressed for the adaptive retrieval technique is possible starvation of requests because of excessive rescheduling. Starvation issue can be eliminated in various stages. One way is depending on the disk scheduling algorithm and assuming that the scheduling algorithm will handle the starvation issue. For example, usage of first-come, first-served (FCFS) disk scheduling algorithm in the disk queues will make sure that requests are sorted in individual disk queues by their arrival times and this sorting will eliminate possible starvation. Another approach is setting up an upper limit of rescheduling value. By this way, if a bucket reaches this upper limit, then it cannot be rescheduled anymore and possible starvation is eliminated. In our simulation, we eliminated starvation problem by using the FCFS scheduling algorithm in individual disk queues. In order to observe the elimination of possible starvation issue in adaptive technique experimentally, we compare the fairness of conservative and adaptive retrieval techniques in Figure 18(b). As the metric of this comparison, we use squared coefficient of variation of response time (σ^2/μ^2). This is a common metric used traditionally to evaluate the fairness of disk scheduling algorithms [49], [50]. Given a constant average response time, a decrease in the coefficient of variation implies reduced response time variance (i.e. improved starvation resistance). It is clear from the figure that adaptive retrieval does not cause a starvation issue; on the contrary, it is improving the starvation resistance of conservative technique in many trace intervals.

V. Conclusion

In this paper, we explored efficient continuous retrieval strategies of replicated data from multi-disk storage architectures considering the heterogeneity of the disks, execution time of the retrieval technique, and individual disk loads. We theoretically proved that batching multiple requests can yield lower total service time; however, experimental results using real world storage traces showed that retrieving the requests immediately considering the disk loads is superior to batching since batching introduces an additional waiting time and it causes the underutilization of the idle disks in the system. As a result, we combined the idea of batching with immediate retrieval and proposed an adaptive retrieval strategy. Experimental results demonstrated that proposed adaptive retrieval strategy benefits from both the service time advantage of the batching idea by rescheduling previous buckets combined with new request, and the waiting time advantage of the immediate retrieval idea by utilizing the idle disks in the system immediately. We also showed that proposed adaptive retrieval technique is superior to existing solutions and helps the storage system more in rush hours when the request interarrival times decrease.

References

[1] V. Filks and S. Zaffos, *MarketScope for Monolithic Frame-Based Disk Arrays*, http://www.gartner.com/id=1591014, Gartner Research, 2011.

[2] A. Kros *et al.*, *Quarterly Statistics: Disk Array Storage, All Countries*, http://www.gartner.com/id=2504815, Gartner Research, June 2013.

[3] *EMC VMAX Storage System*, http://www.emc.com/collateral/hardware/specification-sheet/h6176-symmetrix-vmax-storage-system.pdf, 2011.

[4] *EMC DMX-4 Storage System*, http://www.emc.com/collateral/hardware/specification-sheet/c1166-dmx4-ss.pdf, 2010.

[5] *Hitachi Virtual Storage Platform*, http://www.hds.com/products/storage-systems/hitachi-virtual-storage-platform.html, 2012.

[6] *HP P10000 3PAR Storage Systems*, http://h20195.www2.hp.com/v2/GetPDF.aspx/4AA3-2351ENW.pdf, 2011.

[7] *Sun Storage F5100 Flash Array*, http://www.oracle.com/us/043970.pdf.

[8] *Nimbus Data S-class Enterprise Flash Storage Systems*, http://www.nimbusdata.com/products/Nimbus_S-class_Datasheet.pdf, 2010.

[9] *RamSan-630 Flash Solid State Disk*, http://www.ramsan.com/files/download/212, Texas Memory Systems White Paper, August 2010.

[10] *Violin 3000 Flash Memory Arrays*, http://www.violin-memory.com/wp-content/uploads/Violin-Datasheet-3000.pdf?d=1, 2012.

[11] *Violin 6000 Flash Memory Array*, http://www.violin-memory.com/wp-content/uploads/Violin-Datasheet-6000.pdf?d=1, 2011.

[12] *Sun Storage 7000 Unified Storage Systems Family*, http://www.oracle.com/us/products/servers-storage/039224.pdf, Oracle Datasheet, 2009.

[13] *EqualLogic PS6100XS Hybrid Storage Array*, http://www.equallogic.com/products/default.aspx?id=10653, Dell, Inc., 2011.

[14] *Zebi Hybrid Storage Array*, http://tegile.biz/wp-content/uploads/2012/01/Zebi-White-Paper-012612-Final.pdf, Tegile Systems, Inc., 2012.

[15] *Adaptec High-Performance Hybrid Arrays*, http://www.adaptec.com/nr/rdonlyres/a1c72763-e3b9-45f7-b871-a490c29a9b11/0/hpha5_fb.pdf, PMC-Sierra, Inc., 2010.

[16] N. Altiparmak and A. S. Tosun, "Equivalent disk allocations," *IEEE TPDS*, vol. 23, no. 3, pp. 538–546, March 2012.

[17] J. Lee *et al.*, "Declustering large multidimensional data sets for range queries over heterogeneous disks," in *SSDBM '03*, pp. 212–224.

[18] H. Ferhatosmanoglu, A. S. Tosun, and A. Ramachandran, "Replicated declustering of spatial data," in *ACM PODS'04*, pp. 125–135.

[19] C.-M. Chen and C. Cheng, "Replication and retrieval strategies of multidimensional data on parallel disks," in *CIKM'03*.

[20] K. Frikken, "Optimal distributed declustering using replication," in *ICDT'05*, pp. 144–157.

[21] K. Frikken *et al.*, "Optimal parallel i/o for range queries through replication," in *DEXA'02*, pp. 669–678.

[22] K. Y. Oktay, A. Turk, and C. Aykanat, "Selective replicated declustering for arbitrary queries," in *Euro-Par'09*, pp. 375–386.

[23] A. Turk, K. Y. Oktay, and C. Aykanat, "Query-log aware replicated declustering," *IEEE TPDS*, vol. 24, no. 5, pp. 987–995, 2013.

[24] A. S. Tosun, "Analysis and comparison of replicated declustering schemes," *IEEE TPDS*, vol. 18, no. 11, pp. 1578–1591, November 2007.

[25] J. R. Santos, R. R. Muntz, and B. Neto, "Comparing random data allocation and data striping in multimedia servers," in *SIGMETRICS'00*.

[26] R. Muntz, J. R. Santos, and S. Berson, "A parallel disk storage system for realtime multimedia applications," *International Journal of Intelligent Systems*, vol. 13, pp. 1137–1174, 1998.

[27] W. H. Tetzlaff and R. Flynn, *Block allocation in video servers for availability and throughput*, IBM US Research Centers, 1996.

[28] L. T. Chen and D. Rotem, "Optimal response time retrieval of replicated data," in *ACM PODS'94*, pp. 36–44.

[29] J. Korst, "Random duplicated assignment: an alternative to striping in video servers," in *MULTIMEDIA'97*, pp. 219–226.

[30] P. Sanders, "Asynchronous scheduling of redundant disk arrays," *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1170–1184, Sep. 2003.

[31] A. S. Tosun, "Multi-site retrieval of declustered data," in *ICDCS'08*.

[32] N. Altiparmak and A. S. Tosun, "Integrated maximum flow algorithm for optimal response time retrieval of replicated data," in *ICPP'12*.

[33] N. Altiparmak and A. S. Tosun, "Generalized optimal response time retrieval of replicated data from storage arrays," *ACM Transactions on Storage*, vol. 9, no. 2, pp. 5:1–5:36, Jul. 2013.

[34] K. A. S. Abdel-Ghaffar and A. El Abbadi, "Optimal allocation of two-dimensional data," in *ICDT'97*, Delphi, Greece, pp. 409–418.

[35] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE TPDS*, vol. 12, pp. 1094–1104, 2001.

[36] A. Dan, D. Sitaram, and P. Shahabuddin, "Scheduling policies for an on-demand video server with batching," in *ACM MULTIMEDIA'94*.

[37] C. N. Potts and M. Y. Kovalyov, "Scheduling with batching: a review," *European Journal of Operational Research*, vol. 120, no. 2, 2000.

[38] C. C. Aggarwal, J. L. Wolf, and P. S. Yu, "On optimal batching policies for video-on-demand storage servers," in *ICMCS'96*, pp. 253–258.

[39] N. Agrawal *et al.*, "Design tradeoffs for ssd performance," in *ATC'08*.

[40] D. Narayanan *et al.*, "Everest: Scaling down peak loads through i/o off-loading," in *OSDI'08*.

[41] D. Narayanan *et al.*, "Migrating server storage to ssds: Analysis and tradeoffs," in *EuroSys'09*.

[42] *Storage Networking Industry Association*, http://iotta.snia.org.

[43] S. Kavalanekar *et al.*, "Characterization of storage workload traces from production windows servers," in *IISWC'08*, pp. 119 –128.

[44] *TPC Benchmark C*, http://tpc.org/tpcc/spec/tpcc_current.pdf.

[45] *TPC Benchmark E*, http://tpc.org/tpce/spec/v1.12.0/TPCE-v1.12.0.pdf.

[46] P. Sanders, S. Egner, and K. Korst, "Fast concurrent access to parallel disks," in 11^{th} *ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[47] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," *Journal of the ACM*, vol. 35, pp. 921–940, 1988.

[48] J. Cheriyan and K. Mehlhorn, "An analysis of the highest-level selection rule in the preflow-push max-flow algorithm," *Inf. Process. Lett.*, vol. 69, no. 5, pp. 239–242, Mar. 1999.

[49] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in *SIGMETRICS'94*, pp. 241–252.

[50] T. J. Teorey and T. B. Pinkerton, "A comparative analysis of disk scheduling policies," in *Comm. of the ACM*, March 1972, pp. 177–184.