Multithreaded Maximum Flow Based Optimal Replica Selection Algorithm for Heterogeneous Storage Architectures

Nihat Altiparmak, *Member, IEEE* and Ali Şaman Tosun, *Member, IEEE*

Abstract—Efficient retrieval of replicated data from multiple disks is a challenging problem, especially for heterogeneous storage architectures. Recently, maximum flow based optimal replica selection algorithms were proposed guaranteeing the minimum retrieval time in heterogeneous environments. Although optimality of the retrieval schedule is an important property, execution time of the replica selection algorithm is also crucial since it might significantly affect the performance of the storage sub-system. Current replica selection mechanisms achieve the optimal response time retrieval schedule by performing multiple runs of a maximum flow algorithm in a black-box manner. Such black-box usage of a maximum flow algorithm results in unnecessary flow calculations since previously calculated flow values cannot be conserved. In addition, most new generation multi-disk storage architectures are powered with multi-core processors motivating the usage of multithreaded replica selection algorithms. In this paper, we propose multithreaded and integrated maximum flow based optimal replica selection algorithms handling heterogeneous storage architectures. Proposed and existing algorithms are evaluated using various homogeneous and heterogeneous multi-disk storage architectures. Experimental results show that proposed sequential integrated algorithm achieves 5X speed-up in homogeneous systems, and proposed multithreaded integrated algorithm achieves 21X speed-up using 16 threads in heterogeneous systems over the existing sequential black-box algorithm.

Index Terms—Optimal replica selection, maximum flow, push-relabel, multithreading

1 Introduction

Massive amounts of data is generated everyday through sensors, Internet transactions, social networks, video surveillance systems, and scientific applications. Many organizations and researchers store this data to enable breakthrough discoveries and innovation in science, engineering, medicine, and commerce. Such massive scale of data poses new research problems called big data challenges. As the amount of data grows, disk I/O performance requires further attention since it can significantly limit the performance and scalability of applications. Multi-disk distributed storage architectures have emerged as a promising technology to address the challenges of scalable storage and efficient retrieval of growing data. Especially with the increase in performance gap between processing power and storage device performance, massively parallel I/O systems become crucial to maintain the throughput of data intensive applications. Applications performing space observation and imaging, genomic sequencing, financial analysis, and computational fluid dynamics can generate files that are measured in gigabytes and even terabytes, creating demand for highly concurrent access and high data throughput.

Storage array is a well known example of a multi-disk storage architecture. Besides having hundreds of disk drives, a typical storage array includes controllers with processing units and caching memories. A storage array controller manages 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 percent 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]. 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 while a heterogeneous storage array includes disks 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] and heterogeneous arrays combining magnetic and flash disks have been launched recently [10], [11], [12].

As well as storage arrays, state-of-the-art storage systems in both high-performance computing domain and enterprise storage domain are typically distributed over a network composed of multi-disk storage clusters. Set of I/O server nodes connected to the multi-disk storage clusters are responsible for serving disk requests of I/O clients or compute nodes

- *N. Altiparmak is with the Department of Computer Engineering and Computer Science, University of Louisville, Louisville, KY 40292. E-mail: nihat.altiparmak@louisville.edu.*
- *A.Ş. Tosun is with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249. E-mail: tosun@cs.utsa.edu.*

Manuscript received 19 June 2014; revised 28 May 2015; accepted 23 June 2015. Date of publication 30 June 2015; date of current version 13 Apr. 2016. Recommended for acceptance by B. Ravindran.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2451620

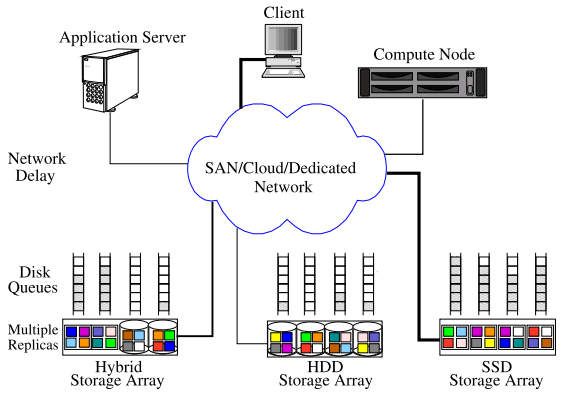


Fig. 1. Distributed and Heterogeneous storage arch.

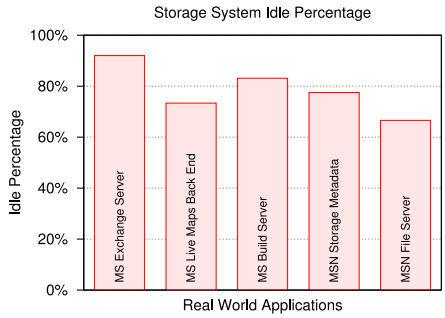


Fig. 2. Real world storage system idle percentages.

that are running data intensive HPC applications [13]. Since such storage clusters generally evolve over time, as newer storage devices are added to them to expand the storage capacity and older/failing storage devices are replaced, they typically end up being heterogeneous in storage components.

The most crucial part of exploiting I/O parallelism is to develop storage techniques that can access data in parallel. Declustering or striping is a common technique for efficient data distribution. Data space is partitioned into disjoint regions (buckets/blocks/chunks) and distributed over multiple disks so that upcoming disk requests can be retrieved in parallel. In addition to single replica declustering [14], [15], [16], [17], many replicated declustering techniques were proposed in the literature [18], [19], [20], [21], [22], [23]. Replication improves the retrieval performance using multiple replicas of the data buckets. In addition to offering lower response time, replication provides better fault-tolerance and support for queries of arbitrary shape. Recently, full or partial replication strategies are proven to be effective in HPC clusters using parallel file systems [24], [25], [26], [27] as well. Readers are directed to [28] for an in-depth comparison and analysis of replicated declustering schemes.

Efficient retrieval of replicated data from multiple disks is a challenging problem, especially for distributed and heterogeneous storage architectures. Given a disk request composed of multiple data buckets (blocks/chunks) that are replicated among multiple disks; the problem is finding a retrieval schedule so that the response time (total retrieval time) of the disk request is minimized. This problem is called the optimal response time retrieval (*replica selection*) problem and the solution should indicate the replica to be used in retrieval for each data bucket. Fig. 1 illustrates the challenges of this problem in distributed and heterogeneous storage architectures, where variable initial load values depending on the disk queue lengths, variable network delays, storage device heterogeneity, and placement of the replicas should be considered for the optimal retrieval schedule. Optimal response time retrieval problem can be solved in polynomial time using a maximum flow formulation. Maximum flow based retrieval algorithms are provided for centralized homogeneous [29], [30], [31], distributed homogeneous [32], and distributed heterogeneous [33] multi-disk storage architectures.

Deciding the retrieval schedule of a disk request is a time critical issue since the execution time of the retrieval algorithm can significantly affect the response time of the request. This affect will be directly visible on the response time of a request if the storage system (the disks to be used in retrieval) is idle when that request arrives. In order to investigate the potential benefit of having a faster retrieval algorithm, we calculated the idle storage percentages of real world applications running inside Microsoft. These applications are Microsoft's own enterprise and production servers, and their storage traces are publicly available online [34]. As shown in Fig. 2, the storage system is between 66 to 92 percent idle while running these applications, which indicates that the storage system will be idle with high probability when a new request arrives at the system. Even if the storage system is busy when a new request arrives, its response time can be affected indirectly if the previous request is delayed due to its execution time. This motivates us to investigate the ways to improve the execution time of the retrieval algorithms further for better overall storage system performance.

Current maximum flow based retrieval algorithms are either integrated with an inefficient maximum flow calculation technique or they use maximum flow algorithms as a black-box method without integrating it into the retrieval algorithm. Such black-box usage of a max-flow calculation results in unnecessary flow calculations since multiple runs of max-flow is performed and previously calculated flows are not conserved within consecutive max-flow runs due to this black-box nature. In addition to this, most new generation storage arrays are powered with multi-core processors and multithreaded retrieval algorithms can also be used to reduce the execution time of retrieval algorithms even further. In this paper, we propose sequential and parallel integrated maximum flow algorithms for the optimal response time retrieval problem. Our algorithms support distributed and heterogeneous storage architectures while handling centralized and homogeneous cases as well. Earlier version of this paper appeared in [35]. This paper included a multithreaded capacity setting algorithm, integration of the fastest known sequential maximum flow algorithm, and extensive evaluation of the proposed and existing retrieval algorithms using simulations driven by real world storage workloads on various homogeneous and heterogeneous multi-disk storage configurations. Experimental results show that proposed multithreaded integrated algorithm achieves an average of 7X speed-up over the algorithm proposed in the earlier version of this paper, and an average of 21X speed-up over the original sequential black-box algorithm proposed in [33] for heterogeneous storage architectures using 16 threads. Furthermore, our proposed sequential integrated algorithm achieves an average of 5X speed-up over the existing algorithm for homogeneous storage architectures.

TABLE 1
Notation

Notation	Meaning
N	Number of disks
d	Disk ID; where $d = \{0,1,\ldots,N-2,N-1\}$
S	Request size; number of buckets in the disk request
r	Replication factor; number of replicas for each bucket
$[i,j](k)$	Replica k of the bucket in row i and column j of an $i \times j$ grid
$G = (V,E)$	A directed graph G with set of edges E and vertices V; $u,v \in V$
C_d	Average retrieval cost of a single bucket from disk d
D_d	Network delay to reach disk d
X_d	Initial load; time it takes for disk d to be idle if busy, 0 otherwise

2 Preliminaries

In this section, we present the necessary background on replicated desclustering, maximum flow algorithms, and maximum flow based replica selection techniques. The notation used in the rest of the paper is summarized in Table 1.

2.1 Replicated Declustering

A replicated declustering of 7×7 grid using seven disks is given in Fig. 3. The grid on the left represents the first replica and the grid on the right represents the second replica. Each square denotes a data bucket and the number on the square denotes the disk that bucket is stored at. Request R_1 in Fig. 3 has six buckets. For retrieval of six buckets from a centralized homogeneous storage array, the best we can expect is $\lceil \frac{S}{N} \rceil = \lceil \frac{6}{7} \rceil = 1$ 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 [36]. 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 can be retrieved in 1 disk access. However, since the first replica of the buckets $[0,0](1)$ and $[2,1](1)$ are both stored in Disk 0 as shown in Fig. 4, retrieval using the first replica requires 2 accesses. When we consider both replicas, we can retrieve R_1 in 1 access using the second replica of the bucket $[2,1](2)$ located in Disk 5.

2.2 Maximum Flow Problem

Maximum flow is a general technique used in optimal response time retrieval problems [29], [30], [31], [32], [33]. An instance of the maximum flow problem is a flow network $G = (V, E, s, t, c)$, where G is a directed graph, $s \in V$ is a distinguished vertex called the *source*, $t \in V$ is a distinguished vertex called the *sink*, and *cap* is a capacity function with $cap(u,v) \geq 0$ for every edge $(u,v) \in E$. A *flow* is a pseudoflow that satisfies the *flow conservation constraint*:

$$\forall v \in V - \{s,t\} : \sum_{u \in V:(u,v) \in E} f(u,v) = \sum_{u \in V:(v,u) \in E} f(v,u), \quad (1)$$

and the *capacity constraint*:

$$\forall u,v \in V : f(u,v) \leq cap(u,v). \quad (2)$$

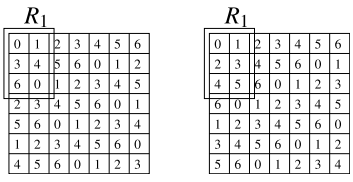


Fig. 3. Replicated declustering.

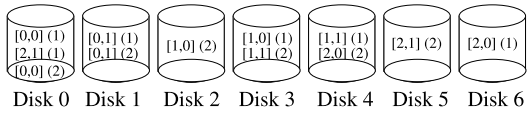


Fig. 4. Placement of buckets on the disks for request R_1.

Equations (1) and (2) state that for all vertices except the source and the sink, the net flow leaving that vertex is zero and an edge cannot carry a flow larger than its capacity. Then, *value* of a flow f is the net flow into the sink as in Equation (3):

$$|f| = \sum_{v \in V:(v,t) \in E} f(v,t). \quad (3)$$

In the maximum flow problem, the goal is sending as much flow as possible between the source and the sink, subject to the capacity and flow conservation constraints. The maximum flow problem has been studied for over 50 years. It has a wide range of applications including the transshipment and assignment problems. Known solutions include Ford-Fulkerson *augmenting path method* [37], [38], closely related *blocking flow method* [39], [40], *network simplex method* [41], [42], and *push-relabel method* of Goldberg and Tarjan [43].

2.2.1 Ford-Fulkerson Method

The motivation behind the Ford-Fulkerson *augmenting path method* is as follows: An *augmenting path* is a residual s-t path. If there exists an augmenting path in G_f (*residual network* of G induced by f), then we can improve f by sending flow along this path. Ford and Fulkerson [37] showed that the converse is also true.

Theorem 1. *A flow f is a maximum flow if and only if G_f has no augmenting paths.*

This theorem motivates the augmenting path algorithm of Ford and Fulkerson's [37], which repeatedly sends flow along augmenting paths, until no such paths remain.

2.2.2 Push-Relabel Method

Push-relabel methods send flow along individual edges instead of the entire augmenting path. This leads to a better performance both in theory and practice [43]. The push-relabel algorithm works with *preflows*, which is a flow that satisfies capacity constraints except additional flows into a vertex is allowed called *excess*. A vertex with positive excess is said to be active. Each vertex is assigned a height, where initially all the heights are zero except $height[s] = |V|$. An iteration of the algorithm consists of selecting an active vertex, and attempting to push its excess to its neighbors with lower heights. If no such edge exists, the vertex's height is increased by 1. The algorithm terminates when there are no more active vertices with label less than $|V|$.

2.3 Optimal Response Time Retrieval

In optimal response time retrieval problem, we have N disks and S buckets. Each bucket can be replicated among multiple disks. The aim is finding a retrieval schedule minimizing the retrieval time of all buckets. Retrieval schedule includes the disks where each bucket should be retrieved from and retrieval (replica selection) algorithms are used to determine the retrieval schedule. Note that the retrieval schedule is

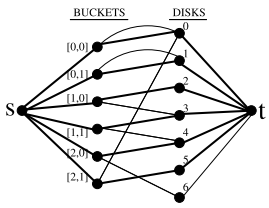


Fig. 5. BRP.

trivial if there is no replication in the system, in which there is only one candidate disk that a bucket can be retrieved from. Depending on the complexity of the system, the problem can be classified as either the Basic Retrieval Problem (BRP) or the Generalized Retrieval Problem (GRP).

2.3.1 Basic Retrieval Problem (BRP)

BRP assumes that all the disks in the system are homogeneous ($C_0 = C_1 = \cdots = C_{N-1}$), network delays to reach the disks are equivalent ($D_0 = D_1 = \cdots = D_{N-1}$), and all the disks are idle ($X_0 = X_1 = \cdots = X_{N-1} = 0$). In this case, response time of the request is determined by the disk that is used to retrieve the maximum amount of buckets. In other words, we need to retrieve as few buckets as possible from the disk that is used to retrieve the maximum amount of buckets.

BRP can be solved as a max-flow problem using graph theory [29]. Maximum flow representation of request R_1 provided in Fig. 3 is shown in Fig. 5. 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_b denoting bucket b and vertex v_d denoting disk d if bucket b is stored on disk d. Next step is to set the capacities of the edges. Let A be the edge set holding every edge e_d between the disk vertex d and the sink (*disk edges*; all the edges going to the sink). All the edges except the ones in A have capacity 1. The capacity of the edges in A are set to the theoretical lower bound for the number of disk accesses; $\lceil \frac{S}{N} \rceil$. Since the request R_1 has 6 buckets and there are 7 disks in the system, all the edges have capacity 1 in this example. Now, we can run the max-flow.

As a result of running the maximum flow algorithm, if the maximum flow value of S is reached, then it means that all the buckets can be retrieved successfully. Otherwise, we need to increment the capacities of all *disk edges* (all the edges going into the sink) by one and re-run the max-flow algorithm. We repeat this incrementation process until the flow of S is achieved. Maximum flow is shown using thick lines in Fig. 5. Flow information indicates the replica to be chosen for the optimal response time retrieval of request R_1. In the worst case, max-flow algorithm might run $O(S)$ times; when all the buckets of a request are stored at a single disk.

2.3.2 Generalized Retrieval Problem (GRP)

GRP generalizes BRP in a way that storage devices can be heterogeneous, disks might have different network delays, and they can have initial loads to be processed before handling the current buckets. Consider the request R_1 given in Fig. 3 again, but this time assume that the system parameters are as in Table 2. GRP can also be solved using a max-flow formulation [33]; however, the capacity setting process of the *disk edges* in the edge set A is more complicated this time. As in BRP, capacity of all the edges except the ones in A are set to 1. However in GRP, we cannot initialize or increment the capacities of the edges in A all together. In BRP, since retrieval cost of the disks were the same ($(C_0 + D_0 + X_0) = (C_1 + D_1 + X_1) = \cdots = (C_{N-1} + D_{N-1} + X_{N-1})$), it was possible to initialize and increment the capacities of the edges is A all at the same time. However in GRP, retrieval costs of the disks might be different depending on the heterogeneity (C_d), initial load (X_d), and network delay (D_d). Therefore, capacity setting and incrementation steps of the algorithm should be performed considering the individual retrieval costs of the disks.

TABLE 2
System Parameters

Disk ID (d)	C_d (ms)	D_d (ms)	X_d (ms)
0-3	8.3	2	1
4	13.2	1	0
5-6	6.1	1	0

In order to set the capacity values of the edges in A efficiently, [33] proposes Binary Capacity Scaling (*BCS*) and Capacity Incrementation (*CI*) Algorithms. *BCS* is used to initialize the capacities by scaling them in binary steps and *CI* is used to increment the capacities in an iterative step. *BCS* first defines a range where the optimal response time is known to be within this range. In each step, the algorithm picks the middle value of this range, calculates the capacities of the edges in A for this middle value, and runs the maximum flow algorithm. Depending on the flow value, the algorithm either decreases the top range or increases the bottom range by half. After the range is small enough, *BCS* stops and *CI* starts to execute with this initial capacity values calculated by *BCS*. *CI* increments the capacity of the edge yielding the minimum cost by one and runs the max-flow algorithm after each and every incrementation step until the flow of S is reached. Fig. 6 shows the proper values of the capacities for the parameters defined in Table 2. Using these capacity values, the optimal response time retrieval schedule of the request R_1 is shown in Fig. 6 with thick lines. Readers are directed to [33] for further detais on *BCS* and *CI* Algorithms.

2.4 Motivation

Deciding the retrieval schedule of a disk request is a time critical issue since the execution time of the retrieval algorithm can dramatically affect the response time of the request. Any improvement on the execution time has a potential effect on increasing the storage system performance. Existing retrieval algorithms are either integrated with an inefficient maximum flow calculation technique, or use maximum flow algorithms as a black-box method without integrating it into the retrieval algorithm. For instance, BRP can be solved using a Ford-Fulkerson based integrated retrieval algorithm proposed in [29]; however, push-relabel method is known to be superior over the Ford-Fulkerson method both in theory and practice [43]. Besides, this solution cannot handle GRP. On the other hand, the solution proposed for GRP in [33] requires multiple maximum flow

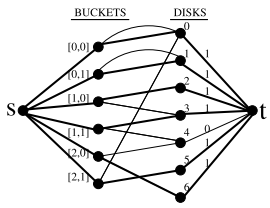


Fig. 6. GRP.

runs using graphs with similar capacity values. However, since the maximum flow calculation is performed as a black-box technique, each run starts with zero flows all over again. Actually, the algorithm increments the capacities used in a previous run and current calculation can be build on the flows calculated previously. This approach would obviously save a considerable amount of flow calculations; however, it requires an integrated maximum flow solution for GRP. In addition to these, most new generation storage arrays are powered with multi-core processors. Multithreaded algorithms for capacity setting and maximum flow calculation steps should be investigated for better utilization of the computing resources and possible execution time improvement.

3 Integrated Retrieval Algorithms

In this section, we present our maximum flow based retrieval algorithms where the maximum flow calculation is integrated into the retrieval algorithm in order to eliminate unnecessary flow calculations. First, we provide the following propositions to be used in optimizing the proposed algorithms later:

Proposition 1. *Let $G = (V, E, s, t, c)$ be a flow network, which is specifically constructed for the retrieval problem. Then, the retrieval problem does not have a solution if:*

$$\sum_{v \in V:(v,t) \in E} cap(v,t) < S. \quad (4)$$

Proof. Retrieval problem has a solution if the maximum flow value equals to the number of buckets to be retrieved such that $|f| = S$. In order to achieve the maximum flow of S, we need at least total of S capacities for the edges going into the sink since a flow more than capacity cannot pass through an edge by the capacity constraint given in Equation (2). □

Based on Proposition 1, there is no need run max-flow until the total capacity value of S is achieved for the *disk edges*; i.e. all the edges going into the sink. Besides, this proposition can be used to define a lower range for the binary capacity scaling algorithm.

Proposition 2. *Let $G = (V, E, s, t, c)$ be a flow network, which is specifically constructed for the retrieval problem. Then, the following statement holds:*

$$\forall v \in V, where\ (v,t) \in E : f(v,t) \leq indeg(v). \quad (5)$$

Proof. This proposition holds by the flow conservation constraint of the max-flow problem provided in Equation (1) and the specific construction of the flow network for the retrieval problem such that for every edge $(v,t) \in E$, all incoming edges to the vertex v has the capacity of 1. □

Based on Proposition 2, it is unnecessary to increment the capacity of a *disk edge* (v,t) more than the indegree of v.

Proposition 3. *Let $G = (V, E, s, t, c)$ be a flow network, which is specifically constructed for retrieving the request R, T_{opt} be the optimal response time of R, and T_d be the retrieval time of disk d. Then, the following two statements hold:*

$$\forall v_d \in V; (v_d, t) \in E : T_d \leq D_d + X_d + indeg(v_d) * C_d, \quad (6)$$

$$T_{opt} \leq Max\{T_0, T_1, \ldots, T_{N-1}\}. \quad (7)$$

Proof. This proposition holds by the definition of optimal response time retrieval, which states that optimal response time retrieval of a request is defined by the disk yielding the maximum retrieval time. Since a disk reaches to its maximum retrieval time when it retrieves all possible buckets it stores from the request, then this number can be achieved using the indegree of the vertex v_d representing the disk d. □

Based on Proposition 3, we can easily define an upper-range for the binary capacity scaling algorithm.

3.1 Ford-Fulkerson Based Solution

The first integrated algorithm we propose for GRP uses the Ford-Fulkerson method as shown in Algorithm 1. This algorithm is modified from the original algorithm provided for BRP in [29]. Algorithm 1 assumes that flow values of the edges going out of the source vertex are all initialized to 1 at the beginning. Assume A is an edge set holding all the edges going into the sink. Line 1 of the Algorithm 1 calls Algorithm 2, which initializes the capacities of the edges in A based on Proposition 1 in order to eliminate unnecessary flow calculations. Algorithm 2 accomplishes this by incrementing the capacities until the total capacity value of the edges in A reaches to S using the capacity incrementation algorithm presented in Algorithm 3. In each incrementation step, Algorithm 3 determines the edges in A yielding the minimum retrieval cost in lines 6-8 and increments the capacities of the edges with this minimum cost in lines 12-13. Note that, if there are more than one edge yielding the same retrieval cost, then their capacities are incremented at the same time as in BRP. Lines 4-5 and lines 10-11 do not consider an edge if the disk associated with that edge cannot be used to retrieve any more buckets based on Proposition 2.

Algorithm 1. Ford-Fulkerson Based Integrated Algorithm

```
1: TestAndSetMinCapacities()
2: for i ← 1 to S do
3:    dfs_success = DFS(G, v[i], t, caps, flow, path)
4:    while (!dfs_success) do
5:       total_caps ← CapacityIncrement()
6:       dfs_success = DFS(G, v[i], t, caps, flow, path)
7:    for all e ∈ path do
8:       if target(e) ≠ t then
9:          G.ReverseEdge(e)
10:      if IsReverse(e) then
11:         flow[e]- -
12:      else
13:         flow[e]++
14: FixReversedEdges()
```

Algorithm 2. $TestAndSetMinCapacities()$

```
1: total_caps ← 0
2: for all incoming edges (e, t) do
3:     total_caps += caps[e]
4: while total_caps < S do
5:     total_caps ← CapacityIncrement()
```

After the capacities in A are initialized, for each bucket i of the request, the algorithm searches for an augmenting path from the vertex representing the bucket i to the sink vertex in lines 2-3 of Algorithm 1. If no augmenting path exists, capacities of the edges in A are incremented by one using Algorithm 3 until a path is found through the lines 4-6. For each edge in the path, line 9 reverses its direction if the edge is between a bucket vertex and a disk vertex. This reversal is necessary to be able to change the retrieval decision of a previously assigned bucket. Finally, lines 10-13 increments or decrements the flow of each edge in the path depending on its direction. If the edge direction is not in its original direction, then the flow is decremented meaning that the retrieval choice is changed; otherwise, the flow is incremented. At the end of the algorithm, we have to fix the directions of the edges since some of them might have a reverse direction.

Applying Proposition 2 in Algorithm 3 ensures that the number of incrementation steps are bounded by $r * S$ in the worst case, where r is the number of replicas or the replication factor. Since the DFS might try $r * S$ edges in the worst case, worst case time complexity of the Algorithm 1 is $O(r^2S^2)$.

Algorithm 3. $CapacityIncrement()$

```
 1: total_caps ← 0
 2: min_cost ← MAXDOUBLE
 3: for all incoming edges (e, t) do
 4:     v ← G.source(e)
 5:     if G.indeg(v) > caps[e] then
 6:         costs[e] ← D[e] + X[e] + (caps[e] + 1) * C[e]
 7:         if costs[e] < min_cost then
 8:             min_cost ← costs[e]
 9: for all incoming edges (e, t) do
10:     v ← G.source(e)
11:     if G.indeg(v) > caps[e] then
12:         if costs[e] == min_cost then
13:             caps[e]++
14:     total_caps += caps[e]
15: return total_caps
```

3.2 Push-Relabel Based Solution

Although Ford-Fulkerson based algorithms are simple and easy to implement, most practical maximum flow implementations are based on the push-relabel algorithm. Therefore, we also propose a push-relabel based integrated retrieval algorithm for GRP. Algorithm 4 presents a basic push-relabel based maximum flow algorithm. Lines 1-8 show the initialization step. Push/relabel operations of the algorithm are performed in lines 9-10, which we skip details there for the sake of simplicity; however, readers are directed to [43] for the implementation of push/relabel operations. When the algorithm terminates, excess value of the sink holds the maximum flow amount that can be pushed from source to sink. Depending on the vertex selection rule, complexity of the algorithm changes. FIFO ordering yields the complexity of $O(|V^3|)$ [43] while high-level selection rule leads to the complexity of $O(|V^2|\sqrt{|E|})$ [44], where $|V|$ is the number of vertices and $|E|$ is the number of edges in the flow graph. For different implementation techniques and optimization methods (global/gap relabelling heuristics) for the push-relabel algorithm, readers are referred to [45].

Algorithm 4. Push-Relabel Algorithm

```
 1: for all out edges(e,s) do
 2:     v ← target(e)
 3:     QUEUE.append(v)
 4:     flow[e] ← cap[e]
 5:     excess[v] += cap[e]
 6: for all nodes(v,G) do
 7:     height[v] ← 0
 8: height[s] ← G.number_of_nodes()
 9: while QUEUE ≠ ∅ do
10:     apply push/relabel operations by updating the QUEUE
11: return excess[t]
```

Algorithm 5. $PushRelabelIncremental()$

```
1: TestAndSetMinCapacities()
2: while (true) do
3:     INIT()
4:     while QUEUE ≠ ∅ do
5:         apply push/relabel operations by updating the QUEUE
6:     if excess[t] == S then
7:         break
8:     CapacityIncrement()
9: return excess[t]
```

Algorithm 6. $INIT()$

```
 1: QUEUE.Clear()
 2: for all nodes(v,G) do
 3:     height[v] ← 0
 4: height[s] ← G.NumberOfNodes()
 5: excess[s] ← 0
 6: for all out edges(e,s) do
 7:     v ← Target(e)
 8:     δ ← cap[e] − flow[e]
 9:     if d > 0 then
10:         QUEUE.Append(v)
11:         flow[e] ← cap[e]
12:         excess[v] += cap[e]
```

Algorithm 5 presents a push-relabel based integrated maximum flow algorithm for GRP. Similar to Algorithm 1, Algorithm 5 sets the capacities in line 1 based on Proposition 1. For every iteration, we need to follow an initialization step slightly modified from the initialization step of the original push-relabel algorithm presented in Algorithm 4. Initialization step for the integrated algorithm is presented in Algorithm 6. First, we clear the queue as in line 1 and initialize the height values as in line 2-4. Push-relabel operations ensure that excess values of the vertices except the source and the sink vertices are all set to 0 when the algorithm terminates. Therefore, we only set the excess value of

source to 0 as in line 5. Note that our aim is conserving the flows found in the previous runs, therefore flow values are not initialized back to 0. In addition, the algorithm should add vertices to the queue only if they can pass more flow in the next push/relabel operations. We check this by using the δ value calculated in line 8 and initialize only such vertices in lines 9-12. After initialization, Algorithm 5 applies the push-relabel operations in lines 4-5 and checks the max-flow of S. If max-flow of S is not achieved, then capacity incrementation is performed respecting Proposition 2. This iteration is repeated until the max-flow of S is reached.

Algorithm 5 solves GRP using an integrated push-relabel based maximum flow algorithm. Although we can conserve the flows calculated in the previous run, the algorithm still considers all possible retrieval times starting from the minimum in an exhaustive search manner leading to $O(c * S)$ max-flow runs. Since we are conserving the flows, the algorithm is expected to run faster than its black-box counterpart; however, we can still improve the worst case complexity further by using the Binary Capacity Scaling technique presented in [33]. *BCS* will bring the capacity values up to an initial value very close to the optimal in $O(log_2(S))$ operations before the incrementation step is started by Algorithm 5.

Algorithm 7 presents our final push-relabel based integrated maximum flow algorithm using the Binary Capacity Scaling technique. First, it calls Algorithm 8 in line 1 to define a range $(t_{min}, t_{max}]$, where the optimal response time is guaranteed to lie within. Such a range can be defined using the Proposition 1 for t_{min} and the Proposition 3 for t_{max}. In order to find t_{min} based on Proposition 1, Algorithm 8 sets the capacities using Algorithm 2 in line 1 and finds the response time of this minimum capacity setting in lines 6-8. In order to find t_{max} based on Proposition 3, Algorithm 8 calculates the response time in lines 9-12 assuming that all the buckets stored in every disk are used in the retrieval decision. Finally, since we want to ensure that there is no solution for t_{min}, we find the min_speed value in line 13-14 representing the smallest C_d value and subtract this value from t_{min} in line 15.

After defining the range, Algorithm 7 calculates the middle value (t_{mid}) of the range in line 3 and finds the capacities corresponding to t_{mid} in line 4 using the capacity scaling algorithm presented in Algorithm 9. Initialization and push/relabel operations are performed in lines 5-7. If there is no solution such that $excess[t] \neq S$, it stores the current flow state of the graph as in lines 9-10 to be used later to eliminate unnecessary flow calculations. Also, it increases t_{min} to t_{mid} as in line 11 to eliminate the bottom range. If there is a solution such that $excess[t] = S$, since we cannot guarantee the optimality of the result, previously saved flow values are restored and t_{max} is decreased to t_{mid} as in lines 13-15 to eliminate the top range. The algorithm stops when the range is smaller than min_speed, restores the saved flows, calculates the final capacities corresponding to t_{min}, and calls the Algorithm 5 through the lines 16-19. In the worst case, Algorithm 7 performs $O(log_2(S))$ incrementation steps and calls Algorithm 5 to reach to the optimal response time. Since the capacity values that Algorithm 5 starts with are very close the optimal values, optimal response time will be achieved by the Algorithm 5 in constant incrementation steps.

Algorithm 7. Push-Relabel Based Integrated Algorithm

```
1: GetRanges(&t_min, &t_max, &min_speed)
2: while (t_max − t_min) ≥ min_speed do
3:    t_mid ← t_min + (t_max − t_min) ∗ 0.5
4:    total_caps ← CapacityScale(t_mid)
5:    INIT()
6:    while QUEUE ≠ ∅ do
7:       apply push/relabel operations by updating the
         QUEUE
8:    if excess[t] != S then
9:       StoreFlows()
10:      tmp_excess_t ← excess[t]
11:      t_min ← t_mid
12:   else
13:      LoadFlows()
14:      excess[t] ← tmp_excess_t
15:      t_max ← t_mid
16: LoadFlows()
17: excess[t] ← tmp_excess_t
18: CapacityScale(t_min)
19: PushRelabelIncremental()
```

4 Multithreaded Implementations

Most new generation storage arrays are powered with multi-core processors. Since retrieval decision is a time critical issue, it is necessary to use multithreaded implementations in order to reduce the execution time of retrieval algorithms further. For instance, a single EMC Symmetrix Vmax 10K storage array supports eight six-core 2.8 GHz Intel Xeon Processors [46], where each core can run two threads concurrently allowing 96 concurrent thread runs. Parallelization can be applied at two different stages, either during the maximum flow calculation step or while determining the correct capacity values.

4.1 Multithreaded Maximum Flow Calculation

Many push-relabel based parallel maximum flow algorithms were proposed in the literature [47], [48], [49]. However, synchronization is the main performance barrier for most of these algorithms. A general technique is the usage of locks to perform push/relabel operations and locks are known to have expensive overheads [50]. An asynchronous parallelization method proposed by Hong and He claims to outperform other parallel algorithms by eliminiating locks and using atomic hardware instructions instead [49]. The algorithm presented in [49] implements the same push/relabel techniques proposed in [43]; however, it does not require any locks or barriers to protect the push/relabel operations. Instead, they use atomic read-modify-write instructions. We implemented a parallel version of our Algorithm 7 using POSIX threads (pthreads) library and the techniques described in [49]. Since the parallelization should take place in the push/relabel operations, line 7 of the Algorithm 7 is modified to support multithreaded push/relabel operations as it is described in [49].

4.2 Multithreaded Capacity Setting Algorithm

Another opportunity of parallel processing for *GRP* is the Binary Capacity Scaling stage of Algorithm 7. *BCS* algorithm first defines a range where the optimal response time is known to be within this range. In each iteration, the

algorithm picks the middle value of the range, calculates the capacities for this middle value, and runs the maximum flow algorithm. Depending on the flow value, the algorithm shrinks the range by half either eliminating the top half portion or the bottom half portion. After the range is small enough, the algorithm terminates. In order to shrink this range faster, multiple threads can be used to calculate the maximum flow of different sub-ranges in parallel. By this way, we can improve the execution time of the binary capacity scaling stage, which will directly improve the execution time of the Algorithm 7.

Algorithm 8. $GetRanges(*t_{min}, *t_{max}, *min\_speed)$

```
1: TestAndSetMinCapacities()
2: (*min_speed) ← MAXDOUBLE
3: (*t_min) ← 0
4: (*t_max) ← 0
5: for all incoming edges (e, t) do
6:    edge_cost ← D[e] + X[e] + caps[e] * C[e]
7:    if edge_cost > (*t_min) then
8:       (*t_min) ← edge_cost
9:    v = G.source(e)
10:   cur_max ← D[e] + X[e] + G.indeg(v) * C[e]
11:   if cur_max > (*t_max) then
12:      (*t_max) ← cur_max
13:   if C[e] < (*min_speed) then
14:      (*min_speed) ← C[e]
15: (*t_min) -= min_speed
```

Algorithm 9. $CapacityScale(t_{mid})$

```
1: total_caps ← 0
2: for all incoming edges (e, t) do
3:    caps[e] ← ⌊(t_mid − D[e] − X[e])/C[e]⌋
4:    if caps[e] < 0 then
5:       caps[e] ← 0
6:       total_caps += caps[e]
7: return total_caps
```

For a disk request composed of S buckets and interval size of min_speed, there is $S \approx \frac{D_d+X_d+C_d*S}{min\_speed}$ intervals in the worst case when all the buckets are stored in a single disk. Using the *BCS*, this range can be shrunk in $O(log_2(S))$ max-flow runs. However, with the help of t threads, the same range can be shrunk in $O(log_{t+1}(S))$ max-flow runs. Note that max-flow run is a costly operation, where each run of a push-relabel based algorithm requires $O(|V^2|\sqrt{|E|})$ to $O(|V^3|)$ computation depending on the implementation. Using t threads, we can decrease the max-flow runs $\frac{log_2(S)}{log_{t+1}(S)} = \frac{log_2(S)}{\frac{log_2(S)}{log_2(t+1)}} = log_2(t+1)$ times. Consider the following example for better understanding of the multithreading process.

Example 1. Fig. 7 defines the initial range to be shrunk between *Max. Range* (t_{max}) and *Min Range* (t_{min}). Assuming $S = 8$ and $t = 7$, execution sequence on the right represents the sequential BCS and the execution sequence on the left represents the multithreaded version of this algorithm. In this case, sequential BCS requires a total of $\log_2 S = 3$ max-flow runs. As a result of the first sequential run (*Seq. run 1*), the upper half range is eliminated, and after the third sequential run, *targeted sub-range* is determined. On the other hand, parallel implementation shown on the left of Fig. 7 requires a total of $S - 1 = 7$ max-flow calculations compared to 3 calculation required by the sequential algorithm. However, since these 7 calculation can be performed in parallel using $t = 7$ threads, *targeted sub-range* can be determined in 1 parallel max-flow run. In this particular example, we achieve $log_2(7 + 1) = 3$X speed-up for the capacity scaling process using $t = 7$ threads.

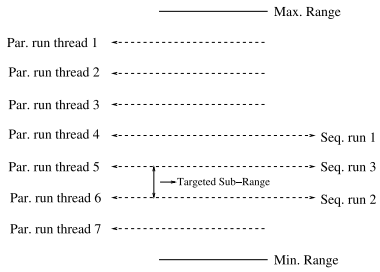


Fig. 7. Multithreaded binary capacity scaling algorithm.

Parallel version of Algorithm 7 adapting multithreaded BSC as explained above is provided in Algorithm 10.

Algorithm 10. Parallel Push-Relabel Integrated Algorithm

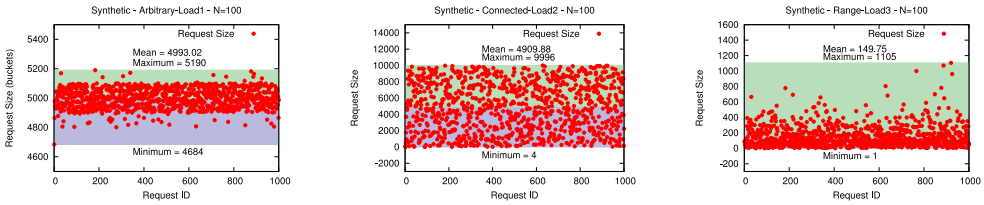
```
1: GetRanges(&t_min, &t_max, &min_speed)
2: total_caps ← CapacityScale(t_min)
3: while (t_max − t_min) ≥ min_speed do
4:    for all sub-range i of [t_min, t_max] in parallel do
5:       total_caps ← CapacityScale(t_i)
6:       INIT()
7:       while QUEUE ≠ ∅ do
8:          apply push/relabel operations by updating the
            QUEUE
9:       if excess[t] == S then
10:         success[i] = 1
11:      else
12:         success[i] = 0
13:   for all sub-range i of [t_min, t_max] do
14:      if success[i] == 0 && success[i + 1] == 1 then
15:         t_min ← success[i]
16:         t_max ← success[i + 1]
17:   total_caps ← CapacityScale(t_min)
18: PushRelabelIncremental()
```

5 Evaluation

In this section, we evaluate the execution time performance of the proposed algorithms as well as its effect on the end-to-end performance of the requests using simulations driven by synthetic and real world storage workloads on various homogeneous and heterogeneous multi-disk storage configurations.

5.1 Allocation Scheme

As the first step of the experimental setup, we need to assume a replicated data allocation (declustering) strategy to distribute the buckets to the disks. For this distribution, we used Random Duplicate Allocation (RDA) that stores a bucket into c disks chosen randomly from the set of disks in the system [51]. Our motivation behind this choice is that RDA performs equally well for all query types while other allocation strategies are generally



(a) Arbit-Load1 (b) Conn-Load2 (c) Range-Load3

Fig. 8. Synthetic workloads - request size statistics - N = 100.

optimized for certain query types. For instance, while orthogonal allocations [18], [52] generally perform better for arbitrary queries, periodic allocation strategies [17], [53] are more suitable for range queries.

5.2 Workloads

In order to evaluate the performance of the algorithms, we also need to use a certain workload information specifying the request details. This information can be obtained from storage traces. We evaluated the algorithms using synthetic and real world multi-disk storage system traces. For data space, we assumed $N \times N$ grid containing total of N^2 buckets in it, where N is the number of disks available in the system.

5.2.1 Synthetic Workloads

For synthetic workloads, we used three popular query types that can happen in a rectangular grid; arbitrary queries, connected queries, and range queries. These query types are combined with different probabilistic query load distributions. We use the notation p_k^i to denote the probability that a query in load i can be retrieved in k disk accesses optimally. Once the optimal number of disk accesses k is selected, the number of buckets is selected uniformly from the range $[(k-1)N+1, kN]$.

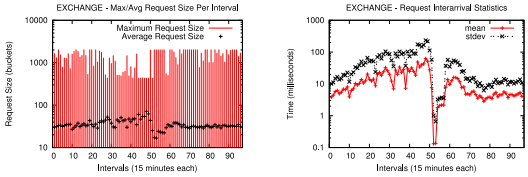
- *Arbitrary-Load1*: Arbitrary queries have no geometric shape. Any subset of the set of all available buckets is an arbitrary query. We can denote arbitrary queries as a set and the number of arbitrary queries is $\sum_{i=1}^{N^2} \binom{N^2}{i}$, which is equal to 2^{N^2} (number of subsets of a set with N^2 elements). The query/request size of *load1* is determined based on the query size distribution of all possible arbitrary queries in an $N \times N$ grid. As a result of this distribution, medium size queries are more likely to happen since the expected request size based on this distribution is $\frac{N^2}{2} + O(\frac{1}{N})$.
- *Connected-Load2*: The buckets in a connected query form a connected graph. Create a node for each bucket in the query and connect two buckets $[i, j]$ and $[m, n]$ by an edge if they are neighbors in the wraparound grid. If the resulting graph is connected then it is a connected query. Query size is determined based on a uniform distribution since finding query size distribution of all possible connected queries in an $N \times N$ grid is not an easy task. We achieve this uniform distribution in *load2* by setting p_k^2 to exactly $\frac{1}{N}$. Expected request size of load 2 queries is $\frac{N^2}{2}$.
- *Range-Load3*: Range queries are rectangular in shape in the wraparound grid. A range query is identified with four parameters (i, j, r, c) $0 \leq i,\ j \leq N-\ 1, 1 \leq r, c \leq N$. i and j are indices of the top left corner of the query and r, c denote the number of rows and columns in the query. The number of distinct range queries on an $N \times N$ grid is $(\frac{N*(N+1)}{2})^2$. Our aim in *load3* is creating smaller size queries compared to *load1* and *load2*. We achieve this by setting p_k^3 to $\frac{2^N}{(2^N-1)*2^k}$. In this case $p_k^3 = \frac{1}{2}p_{k-1}^3$, $2 \leq k \leq N$. Expected request size of load 3 queries is $\frac{3N}{2}$.

Synthetic workload statistics are provided in Fig. 8 showing the request size and interarrival times of the requests per trace interval. By the request size, we mean the number of buckets requested in one request.

5.2.2 Real World Workloads

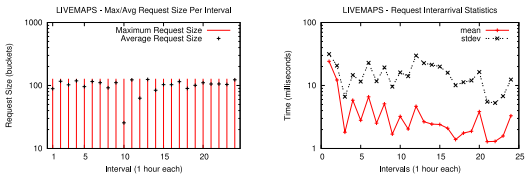
In addition to the synthetic workloads, we also use five popular real storage traces previously used in various storage related studies [54], [55], [56]. These real storage traces include the query size and load information of real storage systems and they are publicly distributed via the online trace repository provided by the Storage Networking Industry Association (SNIA) [34].

- *Exchange*: Exchange is taken from a server running Microsoft Exchange 2007 inside Microsoft [57]. It is a mailing server for 5,000 corporate users and covers a 24-hour weekday period, taken on 12/12/2007, and broken into 96 intervals of 15 minutes. Exchange trace statistics are shown in Fig. 9 for each trace interval shown on the x-axis.
- *LiveMaps*: LiveMaps is a trace of the tile back-end server for the Virtual Earth feature of Live Maps [57]. The tile back-end server holds satellite images and photographs of locations. The trace covers a 24-hour period starting on 2/21/2008 at 1:30 PM. The trace is broken into 24 one hour intervals. LiveMaps trace statistics are provided in Fig. 10 for each trace interval shown on the x-axis.
- *Build*: Build is a trace of a Windows build server (WBS) [57], which does a complete build of the 32-bit version of the Windows Server Operating System every 24 hours. The trace covers a 24-hour period starting on 11/28/2007 at 8:40 PM. The trace is broken into 96 intervals of 15 minutes each and its statistics are shown in Fig. 11.
- *TPC-C*: TPC-C is an online transaction processing (OLTP) benchmark simulating an order-entry environment [58]. It is a mix of five concurrent transactions of different complexities. The TPC-C trace covers 36 minutes of workload taken on 2/26/2008 and broken into 6 intervals of 6 minutes. TPC-C trace statistics are provided in Fig. 12.



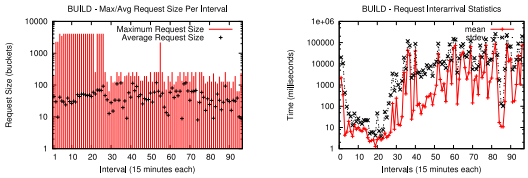
(a) Request Size (b) Interarrival Time

Fig. 9. Exchange trace statistics.



(a) Request Size (b) Interarrival Time

Fig. 10. LiveMaps trace statistics.



(a) Request Size (b) Interarrival Time

Fig. 11. Build trace statistics.

- *TPC-E*: TPC-E is another OLTP benchmark simulating the workload of a brokerage firm [59]. 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 taken on 10/18/2007 and broken into six intervals of 10-16 minutes. TPC-E trace statistics are shown in Fig. 13.

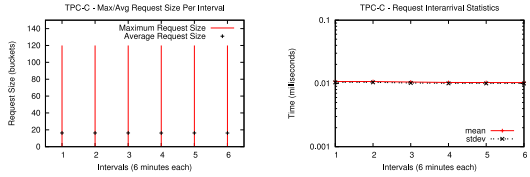
5.3 Storage Configurations and Parameters

We performed simulations using five different disk models; three hard-disk drives (HDD) with different revolutions per minute (RPM) and two solid-state disks (SSD), one high-end and one low-end. Specifications of the disks are provided in Table 3. All values except the *Average Access Time* are obtained from the factory specifications. *Average Access Time* is the average time spent to reach a data bucket (positioning time) in a disk and we calculated it experimentally running a read only benchmark on the real disk. Since the *Average Access Time* value should roughly be the sum of average seek time (*Seek Time*) and rotational latency (*Latency*), factory specifications of the disks seem matching our results.

In order to calculate the retrieval schedule, 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 used the bucket size of 4 KB since it is the default block size

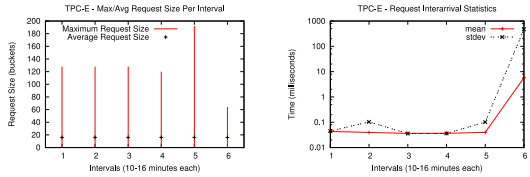
TABLE 3
Disks

Producer	Model	Type	RPM	Seek T.	Latency	Bandwidth	Avg. Access T.
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



(a) Request Size (b) Interarrival Time

Fig. 12. TPC-C trace statistics.



(a) Request Size (b) Interarrival Time

Fig. 13. TPC-E trace statistics.

for many general purpose filesystems. Using 4 KB of bucket size, transfer time of a bucket can be calculated using its *Bandwidth*, which is 68 microseconds for a Barracuda HDD and 15 microseconds for a X25-E SSD. Since the *Average Access Time* value is the dominating factor in retrieval of a single bucket, it is a good approximation to be used by retrieval algorithms. For larger bucket sizes, retrieval algorithms should also consider the transfer time. Please note that transfer time is still considered while evaluating the end-to-end performance using DiskSim in Section 5.6.4, it is only omitted by retrieval algorithms while finding the retrieval schedule based on the bucket size we used in our experiments.

Using the disks provided in Table 3, we created four different homogeneous and heterogeneous multi-disk storage architectures. Table 4 provides these configurations. *disk_conf1* and *disk_conf2* represent homogeneous storage architectures, the former using the slowest disk (Barracuda HDD 7.2 K RPM) of Table 3 and the latter using the fastest disk (Intel X25-E SSD) of Table 3. *disk_conf3* and *disk_conf4* represent heterogeneous storage architectures. In *disk_conf3*, disks are chosen with equal probabilities. In *disk_conf4*, extra 10 percent 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.

5.4 Number of Disks and Replicas

For synthetic workloads and all real workloads except TPC-C and TPC-E, we use $N = 100$ disks, for the TPC-C and TPC-E workloads, we use $N = 1{,}000$ disks. TPC workloads are replayed with more number of disks since they put more pressure on the storage sub-system due to their low request interarrival times. As it can be seen from Figs. 12 and 13, TPC workloads have an interarrival time of

TABLE 4
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%

0.1 to 0.01 milliseconds, issuing tens of millions of requests in every interval. This variation in the number of disks also allows us to observe the effect of number of disks on the execution time of the algorithms. Besides the number of disks, we also performed experiments using various replication factors; for $r = 2, 3, 4, 5$. However, we share the results for $r = 3$ since we did not observe a major change in the execution time performance of the algorithms and $r = 3$ is the most commonly used replication factor in real systems (HDFS, GoogleFS etc).

5.5 Algorithms

We implemented the algorithms listed below for evaluation. Time complexities are given in terms of the number of vertices $|V|$ and the number of edges $|E|$ of the flow graph and assuming the usage of the BCS algorithm when applicable. Note that the flow graph of the retrieval problem has $|V| = S + N + 2$ vertices and $|E| = S * (r + 1) + N$ edges.

- *leda-pr-bb* is a push-relabel based black-box retrieval algorithm proposed in [33] implemented using the LEDA [60] library version 3.4.1 and its maximum flow implementation based on Goldberg and Tarjan's push-relabel technique [43]. This implementation of maxflow has the complexity of $O(|V^3|)$. This is possible since it uses the FIFO ordering for selecting vertices and exact height calculation heuristics suggested by [45]. *leda-pr-bb* has the time complexity of $O(|V|^3 \log S)$.
- *leda-pr-int* is the integrated version of *leda-pr-bb* supporting the flow conservation property based on Algorithm 7 presented in this paper. Similar to *leda-pr-bb*, *leda-pr-int* also has the time complexity of $O(|V|^3 \log S)$; however, due to flow conservation property, *leda-pr-int* is expected to perform better in practice.
- *leda-ff-int* is a Ford-Fulkerson based integrated retrieval algorithm implemented based on Algorithm 1 presented in this paper using the LEDA [60] library version 3.4.1. *leda-ff-int* has the time complexity of $O(r^2 S^2)$.
- *hi-pr-int* is another push-relabel based integrated retrieval algorithm implementing Algorithm 7 presented in this paper. However, max-flow calculation of *hi-pr-int* is based on Goldberg's *hi-pr* implementation. *hi-pr* is currently the fastest sequential implementation available that we are aware of. It has the time complexity of $O(|V^2|\sqrt{|E|})$ [44]. It achieves a good performance by using the highest level vertex selection strategy combined with global and gap relabelling heuristics [45]. *hi-pr-int* has the time complexity of $O(V^2\sqrt{E} \log S)$.
- *amf-int* is a multithreaded push-relabel based integrated algorithm where the parallelization is applied at the max-flow calculation step using the parallelization technique of [49] as described in Section 4.1. *amf-int* has the time complexity of $O(|V|^2|E| \log S)$.
- *mbcs-int* is another multithreaded push-relabel based integrated algorithm based on Goldberg's *hi-pr* implementation where the parallelization is applied at the binary capacity scaling algorithm as described in Section 4.2, Algorithm 10. Similar to *hi-pr-int*, *mbcs-int* also has the time complexity of $O(V^2\sqrt{E} \log S)$.

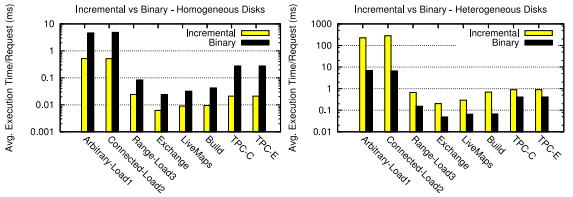
We implemented the algorithms described above in C programming language and compiled using gcc version 4.4.3 optimization level 3 (-O3) except the ones requiring the C++ LEDA library (*leda-pr-bb*, and *leda-ff-int*), which are compiled using g++ version 4.4.3 optimization level 3 (-O3). The machine we used for evaluation has a 10 core Intel Xeon E5-2680 processor supporting 20 threads simultaneously due to hyper-threading and each core is running at 2.8 GHz clock rate. The system has 32 GB of memory and it runs Ubuntu 14.04.2 LTS operating system. For multithreading, we used the POSIX threads (pthreads) library.

5.6 Results

In this section, we provide some of the experimental results that are interesting for our purposes. Our aim is comparing the execution time performance of the algorithms described in Section 5.5, and investigate its effect on the response time of the disk requests. As a result of the experiments, we observed that *disk_conf1* performs similar to *disk_conf2* and *disk_conf3* performs similar to *disk_conf4* in terms of the execution time performance. Therefore, we use *disk_conf1* to represent homogeneous storage architectures and *disk_conf3* to represent heterogeneous storage architectures in the rest of the paper. In addition to this, we found out that push-relabel based retrieval algorithms are superior to the Ford-Fulkerson based retrieval algorithm (*leda-ff-int*) as expected by their time complexities. Therefore, we do not share these comparisons to save space for more interesting results.

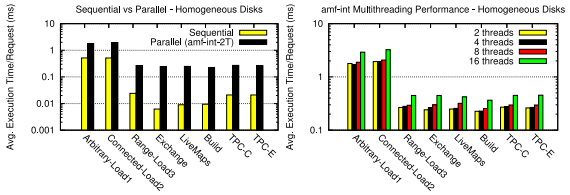
5.6.1 Incremental versus Binary

In this section, we investigate the effect of using Binary Capacity Scaling in homogeneous and heterogeneous storage architectures. Fig. 14 presents this comparison using the *hi-pr-int* algorithm. *Incremental* implements Algorithm 5 and *Binary* implements Algorithm 7. In each figure, *x*-axis shows the workload and *y*-axis shows the average execution time per request for both incremental and binary algorithms. As it is clear from Fig. 14b that *BCS* is extremely beneficial if the storage architecture is heterogeneous. Especially for large requests of Arbitrary-Load1 and Connected-Load2 workloads, *BSC* saves up to 0.3 seconds per request. On the other hand, incremental algorithm performs clearly better than *BCS* for homogeneous storage architectures as plotted in Fig. 14a. The reason for this lies in the number of incrementation steps performed. Remember that each incrementation step is followed by a max-flow calculation to check if the optimal retrieval schedule for all buckets is achieved or not. When the disks are homogeneous, capacities of all disk edges can be incremented at once yielding a total of N capacity incrementation in one step, where N is the number of disks in the system. This is possible in the homogeneous case because the cost of retrieval is the same for every disk in the system. Therefore, after one or two max-flow calculations, the optimal solution is generally achieved in the homogeneous case. When the disks are heterogeneous, each disk might have a unique retrieval cost causing a single incrementation (the one yielding the minimum retrieval cost) in each incrementation step. Therefore, the number of max-flow runs is much higher compared to the homogeneous case.



(a) Homogeneous Disks (b) Heterogeneous Disks

Fig. 14. Incremental versus Binary.



(a) Sequnetial vs Parallel Perf. (b) amf Multithreading Performance

Fig. 16. Sequential versus parallel - homogeneous disks.

As a result of these findings, we suggest the use of a hybrid algorithm based on the heterogeneity of the storage system. If the storage devices are homogeneous, just a capacity incrementation algorithm without *BCS* should be used as in Algorithm 5; however, if the storage devices are heterogeneous, then *BSC* should also be integrated into the retrieval algorithm as in Algorithm 7. We apply this hybrid approach in the rest of the experimentation.

5.6.2 Blackbox versus Integrated

In this section, we compare the execution time performance of the black-box push-relabel algorithm (*leda-pr-bb*) proposed in [33] with our sequential integrated push-relabel implementations (*leda-pr-int* and *hi-pr-int*) presented in this paper as Algorithm 7. Fig. 15 presents this comparison for homogeneous (Fig. 15a) and heterogeneous (Fig. 15b) disks, where x-axis shows the workload and y-axis shows the average execution time per request for both black-box and integrated algorithms. In both homogeneous and heterogeneous cases, the integrated algorithms clearly outperform the black-box algorithm. Among the integrated implementations, *hi-pr-int* clearly outperforms *leda-pr-int* for every workload. As a result, integrated *hi-pr-int* algorithm achieves an average of 5X speed-up over the black-box *leda-pr-bb* algorithm for homogeneous disks, and an average of 5.5X speed-up for heterogeneous disks, considering all workloads we used. The main reason behind this performance gain lies in the flow conservation property of the integrated algorithm. Since the black-box algorithm does not have the flow conservation property, it starts with zero flows in each max-flow run and recalculates the previously calculated flows instead of conserving them.

5.6.3 Sequential versus Parallel

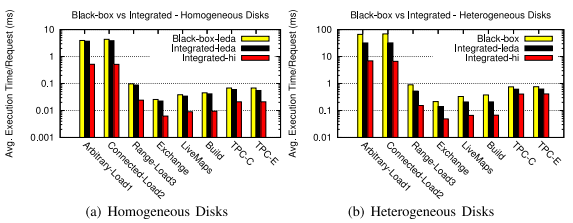
In this section, we compare the execution time performance of the multithreaded implementations (*amf-int* and *mbcs-int*) with the best sequential retrieval algorithm we have achieved so far (*hi-pr-int*) in Figs. 16 and 17 for homogeneous and heterogeneous disks respectively. Since purely incremental algorithms are performing clearly better than *BCS* based algorithms in homogeneous disks, we only compare the performance of *hi-pr-int* and *amf-int* in Fig. 16. In Figs. 16b and 17b, we show the performance of the parallel algorithms for different number of threads, then select the optimum number of threads for each algorithm and use this number in Figs. 16a and 17a to compare their performance with the sequential algorithm.

One immediate observation we make out of these results is that *amf-int* consistently performs the worst as shown in Figs. 16a and 17a. This is actually not very surprising because *amf-int* applies parallelization in the maximum flow calculation stage and parallel maximum flow calculations can generally provide performance improvements for very large graph sizes; especially graphs having millions of edges due to better load balancing among the threads [49]. This issue can also be observed in Fig. 16b showing the performance of *amf-int* for different number of threads, where *amf-int*'s performance generally decreases as the number of threads increase, especially for small graph sizes of Range-Load3, Exchange, LiveMaps, Build, TPC-E and TPC-C workloads. Table 5 shows the average graph sizes of the workloads in our experiments based on their average request size S, number of disks N, and the number of replicas r used in the evaluation.

In all the experiments we performed for homogeneous disks using different workloads and replication factors, we found out that the sequential algorithm *hi-pr-int* performs the best without exception. On the other hand, the parallel algorithm *mbcs-int* shown in Algorithm 10 consistently outperforms the other parallel and sequential implementations for heterogeneous disks. Also, the performance of *mbcs-int* increases as the number of threads increases as shown in Fig. 17b. This proves that parallelizing the *BCS* approach as in *mbcs-int* scales better with the processing resources of the system compared to parallelizing the maximum flow calculation as in *amf-int*. Considering all the workloads we used, the proposed multithreaded and integrated *mbcs-int* algorithm achieves an average of 4.5X speed-up with 16 threads over the best sequential integrated implementation (*hi-pr-int*), and an average of 21X speed-up over the existing sequential black-box algorithm proposed in [33] for heterogeneous architectures.

5.6.4 End-to-End Storage Performance

In this section, we investigate the effect of improving execution time on the end-to-end performance of the disk

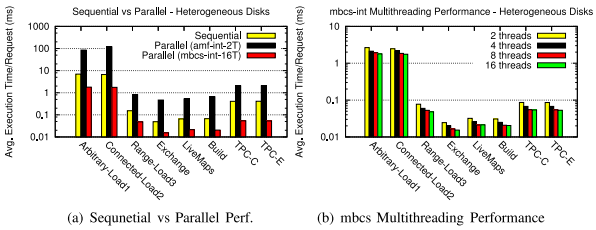


(a) Homogeneous Disks (b) Heterogeneous Disks

Fig. 15. Black-box versus Integrated.

TABLE 5
Average Graph Sizes of Workloads

Workload	S	N	r	$\|V\| = S+N+2$	$\|E\| = S*(r+1)+N$
Arbitrary-Load1	4993	100	3	5095	20072
Connedted-Load2	4909	100	3	5011	19736
Range-Load3	149	100	3	251	696
Exchange	35	100	3	137	240
LiveMaps	101	100	3	203	504
Build	44	100	3	146	276
TPC-C	16	1000	3	1018	1064
TPC-E	16	1000	3	1018	1064



(a) Sequnetial vs Parallel Perf. (b) mbcs Multithreading Performance

Fig. 17. Sequential versus parallel - heterogeneous disks.

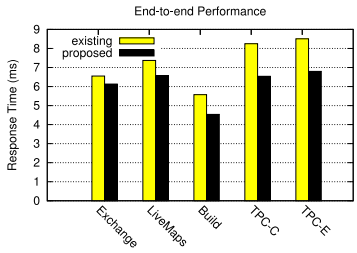


Fig. 18. End-to-end performance - averaged over all intervals.

requests, represented by the *response time* metric. *Response time* indicates the time elapsed between the arrival of a disk request to the storage system and the completion of that request, including its *Service Time* by the storage devices, the *Waiting Time* spent in underlying storage queues, and the *Execution Time* of the retrieval algorithm. For end-to-end performance evaluation, we use the DiskSim simulator [61], which is an efficient, accurate, and highly-configurable disk system simulator developed by the Parallel Data Lab at the Carnegie Mellon University to support research into various aspects of storage subsystem architecture. It has been used in a variety of published studies to understand modern storage performance. We modified the source code of DiskSim so that we can include the execution time delay caused by the retrieval algorithms. For this purpose, we created a custom controller design assumed to calculate the retrieval schedule, which applies the execution time delay of the retrieval algorithm on the requests before they are dispatched to the individual disks queues. For this approach to work, we disabled the queuing applied at the device driver as DiskSim performs by default; instead, controllers of individual disks are enabled to perform queuing and disk scheduling.

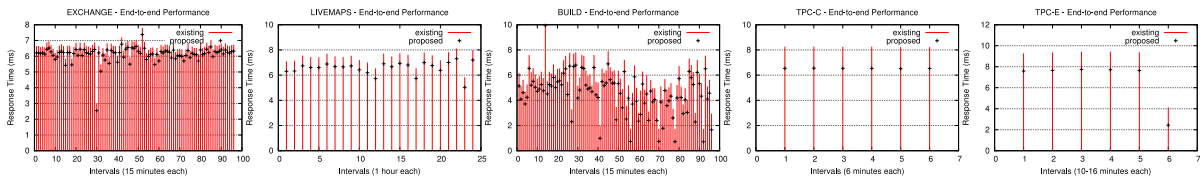
Figs. 18 and 19 provide the average response time values of the requests performed by the real world workloads averaged over all intervals and for individual intervals of each workload, respectively. We compare the performance of the existing retrieval algorithm (*leda-pr-bb*) published in [33] with the multithreaded and integrated retrieval algorithm proposed in this paper (*mbcs-int*) for heterogeneous disks. In order to emulate a similar queuing effect, we followed the original disk topologies used while running these applications, and performed the requests on the corresponding bytes of the storage devices as listed in the trace files using the original request size and arrival time information.

As it is clear from the figures, improving the execution time directly affects the response time of the requests. Although the request sizes of the real world workloads shown here are relatively small compared to the synthetic workloads used in this paper, they still provide up to 2 milliseconds performance improvement per request, which translates to 20 percent improvement in the end-to-end performance. Besides, individual interval performance shown in Fig. 19 indicates that even low interarrival times of the TPC-C and TPC-E traces shown in Figs. 12b and 13b respectively translates into a decent performance improvement, proving the indirect effect of execution time caused by previous requests on the response time of the current request.

Figs. 18 and 19 prove that the execution time improvement directly or indirectly translates into the end-to-end performance improvement of the storage sub-system. Therefore, workloads having larger request sizes as in the synthetic workloads used in this paper are expected to provide a better end-to-end performance improvement since we achieved around 60 milliseconds execution time saving per request for requests having the average request size of $\sim$5,000 buckets. Similarly, a better end-to-end performance improvement will be be achieved for larger storage systems. Modern multi-disk distributed storage architectures are generally composed of thousands of disk drives and deal with files measured in gigabytes and even terabytes. Even in a single storage array, number of disks can reach up to tens of thousands now. One example is EMC VMAX 40K, which supports 6,400 disks in total; 3,200 HDDs and 3,200 SSDs [46].

6 Discussion

One interesting issue to be discussed is that parallelization of the max-flow calculation using the *amf* technique is not very suitable for the special graph structure of the retrieval problem. Performance improvement of *amf* over *hi-pr* is achieved in [49] for either complete graphs having 2,000 to 4,000 vertices ($\sim$2 to $\sim$8 million edges), or for dense graphs having hundreds of thousands of vertices. Authors of *amf* claim this issue being due to the limited number of vertices that overflow at the same time, which they found to be orders of magnitudes higher in their dense graphs than their sparse graphs. This number directly translates to the number of available push and lift operations that can keep the threads busy. Therefore, *amf* algorithm can utilize the threads more efficiently for dense and large graphs. Since the graph structure of the retrieval problem is sparse as having $|V| = S + N + 2$ vertices and $|E| = S * (r + 1) + N$



(a) Exchange (b) LiveMaps (c) Build (d) TPC-C (e) TPC-E

Fig. 19. End-to-end performance - individual intervals.

edges, and the values of N, S, r are limited (see Table 5 for the average graph sizes of the workloads in our experiments), we conclude that the parallelization technique used in *mbcs-int* is more suitable for the retrieval problem.

Second, it should be also noted that proposed multithreaded *mbcs-int* algorithm achieves $log_2(t+1)$ times speed-up using t threads for the capacity scaling process compared to its sequential counterpart. However, this speed-up is achieved by performing more computation than the sequential algorithm. In order to achieve $log_2(t+1)$ times speed-up, *mbcs-int* algorithm performs $S - log_2(S)$ times more max-flow computations in total. However, due to parallel execution of these extra max-flow computations, *mbcs-int* still achieves a better execution time assuming the underlying hardware structure supports simultaneous execution of t threads.

Next, it is possible to implement the multithreaded version of the incremental algorithm (Algorithm 5) using a similar parallelization technique applied at the BCS stage of *mbcs-int* such that different threads can calculate the flow values of the same graph simultaneously for different capacities. However, as a result of our experimental evaluation, we observed that the incremental algorithm does not provide much potential for parallelization since the max-flow calculation is performed very limited amount of time. We found out that max-flow calculation is generally performed only once, and rarely twice in the incremental stage of the retrieval algorithm. For all the experiments we performed, we have never observed a case where the incremental algorithm performs more than two max-flow calculations. This was actually our intention in designing the algorithms and the reasons behind this is the application of Propositions 1 and 2 in the homogeneous case, and the final reduced range produced by the BCS algorithm being less than the min_speed value for the heterogeneous case.

Finally, this paper draws on the full replication of the dataset where every bucket is replicated based on a predefined replication factor. However, for many applications dealing with large datasets, full replication might not be feasible and partial replication can be used instead where only a subset of the whole dataset is replicated. In addition to this, the number of replicas for each bucket does not have to be uniform as well. Nevertheless, the proposed parallel and sequential retrieval algorithms in this paper support all aforementioned replication scenarios and the formulation of the retrieval problem remains the same. One optimization in partial replication can be not including the buckets having a single replica to the flow graphs (shown in Figs. 5 and 6) since the retrieval decision is obvious for these buckets. Otherwise, having a single edge or multiple edges between a bucket vertex and a disk vertex does not affect the correctness or the optimality of the solution.

7 Conclusion

In this paper, we proposed multithreaded and integrated maximum flow based replica selection algorithms for distributed and heterogeneous parallel disk architectures guaranteeing the optimal response time retrieval. In our algorithms, we used various maximum flow calculation and parallelization techniques. As a result of our experimentation, we discovered the followings: (a) purely incremental algorithms are performing better in homogeneous cases; however, applying the binary capacity scaling mechanism improves the performance considerably in heterogeneous settings, (b) push-relabel based algorithms are superior to Ford-Fulkerson based algorithms in retrieval schedule calculation, (c) parallelizing the max-flow calculation does not provide a performance benefit for the retrieval problem due to its sparse and relatively small graph structures; instead, parallelization in binary capacity scaling stage is a better approach and scales well with the number of available processors, (d) proposed integrated sequential *hi-pr-int* algorithm performs the best in homogeneous case achieving 5X speed-up on average, and proposed integrated multithreaded *mbcs-int* algorithm performs the best in heterogeneous case achieving 21X speed-up on average over the existing black-box sequential algorithm using 16 threads.

References

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

[2] A. Kros, M. Suzuki, S. Low, R. W. Cox, A. Kim, J. Chang, A. Munglani, S. KB, and S. Deshpande. (2013, Jun.). *Quart. Statist.: Disk Array Storage, All Countries* [Online]. Available: http://www.gartner.com/id=2504815, Gartner Res..

[3] *EMC VMAX Storage System*. (2011) [Online]. Available: http://www.emc.com/collateral/hardware/specification-sheet/h6176-symmet rix-vmax-storage-system.pdf

[4] *EMC DMX-4 Storage System*. (2010) [Online]. Available: http://www.emc.com/collateral/hardware/specification-sheet/c1166-dmx4-s s.pdf

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

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

[7] *Nimbus Data S-class Enterprise Flash Storage Systems*. (2010) [Online]. Available: http://www.nimbusdata.com/products/Nimbus_S-class_Datasheet.pdf

[8] *RamSan-630 Flash Solid State Disk*. (2010, Aug.) [Online]. Available: http://www.ramsan.com/files/download/212, Texas Memory Systems White Paper

[9] *Violin 6000 Flash Memory Array*. (2011) [Online]. Available: http://www.violin-memory.com/wp-content/uploads/Violin-Datasheet-6000.p df?d=1

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

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

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

[13] *Fusion HPC Cluster*. (2009) [Online]. Available: http://www.lcrc.anl.gov/jazz/Presentations/Fusion-Briefing-091029z.pdf

[14] C. Chen, R. Bhatia, and R. Sinha, "Declustering using golden ratio sequences," in *Proc. 16th Int Conf. Data Eng.*, San Diego, CA, USA, Feb. 2000, pp. 271–280.

[15] J. Lee, M. Winslett, X. Ma, and S. Yu, "Declustering large multidimensional data sets for range queries over heterogeneous disks," in *Proc. 15th Int. Conf. Sci. Statistical Database Manag.*, 2003, pp. 212–224.

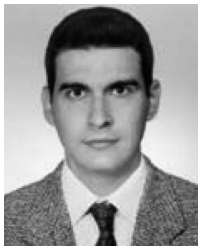
[16] A. S. Tosun,"Threshold-based declustering," *Inf. Sci.*, vol. 177, no. 5, pp. 1309–1331, 2007.

[17] N. Altiparmak and A. S. Tosun, "Equivalent disk allocations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 3, pp. 538–546, Mar. 2012.

[18] H. Ferhatosmanoglu, A. S. Tosun, and A. Ramachandran, "Replicated declustering of spatial data," in *Proc. 23rd ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst.*, 2004, pp. 125–135.

[19] C.-M. Chen and C. Cheng, "Replication and retrieval strategies of multidimensional data on parallel disks," in *Proc. 12th Int. Conf. Inf. Knowl. Manag.*, 2003, pp. 32–39.
[20] K. Frikken, "Optimal distributed declustering using replication," in *Proc. 10th Int. Conf. Database Theory*, 2005, pp. 144–157.
[21] K. Frikken, M. Atallah, S. Prabhakar, and R. Safavi-Naini, "Optimal parallel i/o for range queries through replication," in *Proc. 13th Int. Conf. Database Expert Syst. Appl.*, 2002, pp. 669–678.
[22] K. Y. Oktay, A. Turk, and C. Aykanat, "Selective replicated declustering for arbitrary queries," in *Proc. 15th Int. Euro-Par Conf. Parallel Process.*, 2009, pp. 375–386.
[23] A. Turk, K. Y. Oktay, and C. Aykanat, "Query-log aware replicated declustering," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 5, pp. 987–995, May 2013.
[24] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur, "Pattern-direct and layout-aware replication scheme for parallel i/o systems," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 345–356.
[25] S. W. Son, S. Lang, R. Latham, R. B. Ross, and R. Thakur, "Reliable MPI-IO through layout-aware replication," presented at the IEEE 7th Int. Workshop Storage Network Architecture and Parallel I/O, Denver, CO, USA, 2011.
[26] J. Jenkins, X. Zou, H. Tang, D. Kimpe, R. Ross, and N. F. Samatova, "Radar: Runtime asymmetric data-access driven scientific data replication," in *Proc. 29th Int. Conf. Supercomput. - Vol. 8488*, 2014, pp. 296–313.
[27] J. Jenkins, X. Zou, H. Tang, D. Kimpe, R. B. Ross, and N. F. Samatova, "Parallel data layout optimization of scientific data through access-driven replication," Argonne Nat. Lab., Lemont, IL, USA, Tech. Rep. ANL/MCS-P5072-0214, 2014.
[28] A. S. Tosun, "Analysis and comparison of replicated declustering schemes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 11, pp. 1578–1591, Nov. 2007.
[29] L. T. Chen and D. Rotem, "Optimal response time retrieval of replicated data," in *Proc. 13th ACM SIGACT-SIGMOD-SIGART Symp. Principles Database Syst.*, 1994, pp. 36–44.
[30] J. Korst, "Random duplicated assignment: An alternative to striping in video servers," in *Proc. 5th ACM Int. Conf. Multimedia*, 1997, pp. 219–226.
[31] P. Sanders, "Asynchronous scheduling of redundant disk arrays," *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1170–1184, Sep. 2003.
[32] A. S. Tosun, "Multi-site retrieval of declustered data," in *Proc. 28th Int. Conf. Distrib. Comput. Syst.*, 2008, pp. 486–493.
[33] N. Altiparmak and A. S. Tosun, "Generalized optimal response time retrieval of replicated data from storage arrays," *ACM Trans. Storage*, vol. 9, no. 2, pp. 5:1–5:36, Jul. 2013.
[34] *Storage Networking Industry Association* [Online]. Available: http://iotta.snia.org
[35] N. Altiparmak and A. S. Tosun, "Integrated maximum flow algorithm for optimal response time retrieval of replicated data," in *Proc. 41st Int. Conf. Parallel Process.*, 2012, pp. 11–20.
[36] K. A. S. Abdel-Ghaffar and A. El Abbadi, "Optimal allocation of two-dimensional data," in *Proc. 6th Int. Conf. Database Theory*, Delphi, Greece, 1997, pp. 409–418.
[37] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Can. J. Math.*, vol. 8, pp. 399–404, 1956.
[38] L. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ, USA: Princeton Univ. Press, 1962.
[39] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Sov. Math. Dok*, vol. 11, pp. 1277–1280, 1970.
[40] A. V. Karzanov, "Determining the maximal flow in a network by the method of preows," *Sov. Math. Dok*, vol. 15, pp. 434–437, 1974.
[41] G. B. Dantzig, "Application of the simplex method to a transportation problem," in *Activity Analysis of Production and Allocation*. New York, NY, USA: Wiley, 1951.
[42] P. Jensen and J. Barnes, *Network Flow Programming*, series Board of advisors, engineering. New York, NY, USA: Wiley, 1980.
[43] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," *J. ACM*, vol. 35, pp. 921–940, 1988.
[44] J. Cheriyanand S. N. Maheshwari,"Analysis of preflow push algorithms for maximum network flow," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1057–1086, 1989.
[45] B. V. Cherkassky and A. V. Goldberg, "On implementing the push-relabel method for the maximum flow problem," vol. 19, no. 4, pp. 390–410, Dec. 1997.
[46] *Emc Symmetrix VMAX 40K*. (2015) [Online]. Available: http://www.emc.com/storage/vmax10k-20k-40k/index.htm
[47] R. J. Anderson and J. a. C. Setubal, "On the parallel implementation of Goldberg's maximum flow algorithm," in *Proc. ACM 4th Annu. ACM Symp. Parallel Algorithms Archit.*, 1992, pp. 168–177.
[48] D. A. Bader and V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," in *Proc. 18th ISCA Int. Conf. Parallel Distrib. Comput. Syst.*, 2005, pp. 41–48.
[49] B. Hong and Z. He, "An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 1025–1033, Jun. 2011.
[50] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Mateo, CA, USA: Morgan Kaufmann, Aug. 1998.
[51] P. Sanders, S. Egner, and K. Korst, "Fast concurrent access to parallel disks," in *Proc. 11th ACM-SIAM Symp. Discr. Algorithms*, 2000, pp. 849–858.
[52] A. S. Tosun, "Replicated declustering for arbitrary queries," in *Proc. 19th ACM Symp. Appl. Comput.*, Mar. 2004, pp. 748–753.
[53] A. S. Tosun and H. Ferhatosmanoglu, "Optimal parallel I/O using replication," in *Proc. Int. Conf. Parallel Process. Workshops*, 2002, pp. 506–513.
[54] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *Proc. Annu. Tech. Conf.*, 2008, pp. 57–70.
[55] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowston, "Everest: Scaling down peak loads through i/o off-loading," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 15–28.
[56] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowston, "Migrating server storage to SSDs: Analysis and tradeoffs," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 145–158.
[57] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Proc. IEEE Int. Symp. Workload Characterization*, 2008, pp. 119–128.
[58] *TPC Benchmark C* [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf, 2010.
[59] *TPC Benchmark E* [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpce-v1.14.0.pdf, 2015.
[60] K. Mehlhornand S. Näher,"LEDA: A platform for combinatorial and geometric computing," *Commun. ACM*, vol. 38, no. 1, pp. 96–102, 1995.
[61] J. S. Bucy, J. Schindler, S. W. Schlosser, G. R. Ganger, and Contributors., "The DiskSim simulation environment version 4.0 reference manual," Carnegie Mellon Univ. Parallel Data Lab, Tech. Rep. CMU-PDL-08-101, May 2008.

Nihat Altiparmak (M'10) received the BS degree in computer engineering from Bilkent University, Ankara, Turkey in 2007, and the MS and PhD degrees in computer science from the University of Texas at San Antonio in 2012 and 2013, respectively. He is currently an assistant professor in the Computer Engineering and Computer Science Department, University of Louisville. His research interests include storage systems, distributed systems, networking, and network security. He is a member of the IEEE.



Ali Şaman Tosun (M'06) received the BS degree in computer engineering from Bilkent University, Ankara, Turkey in 1995, and the MS and PhD degrees from the Ohio State University in 1998 and 2003, respectively. He joined the Department of Computer Science, University of Texas at San Antonio in 2003. He is currently an associate professor in the Department of Computer Science, University of Texas at San Antonio. His research interests include storage systems, large-scale data management, and security. He is a member of the IEEE.