2016 IEEE 23rd International Conference on High Performance Computing

Dynamic Data Layout Optimization for High Performance Parallel I/O

Everett Neil Rush, Bryan Harris, Nihat Altiparmak
Department of Computer Eng. & Computer Science
University of Louisville, KY 40292
{e.rush,bryan.harris.1,nihat.altiparmak}@louisville.edu

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

***Abstract*—Storage performance bottlenecks are one of the major threats limiting the scalability of I/O intensive applications. Parallel storage systems have the potential to alleviate I/O bottlenecks through concurrent operation of independent storage components if a parallelism-aware data layout can be continuously guaranteed. Existing systems use one-layout-fits-all data placement strategy that frequently results in sub-optimal I/O parallelism. Guided by association rule mining, graph coloring, bin packing, and network flow techniques, this paper proposes a general framework for self-optimizing parallel storage systems, with the goal of continuously providing a high-degree of I/O parallelism that is robust to changes in the parallel access patterns of applications and the coexistence of applications with different parallel access characteristics. Evaluation results indicate that the proposed framework is highly successful in adjusting to skewed parallel access patterns for both traditional hard disk drive (HDD) based storage arrays and solid-state drive (SSD) based all-flash arrays. In addition to the storage arrays, the proposed framework is sufficiently generic to be tailored to various other parallel storage scenarios including but not limited to key-value stores, parallel/distributed file systems, and internal parallelism of SSDs.**

***Keywords*-storage systems; parallel I/O; self-optimization**

I. Introduction

Many real-world applications are I/O intensive in nature and have strict latency requirements typically specified in service-level agreements (SLA). High performance parallel storage systems have the potential to provide low-latency and high-throughput I/O performance through concurrent operation of individual storage components if a parallelism-aware data layout can be continuously guaranteed. Storage arrays are well known examples of high performance parallel storage systems. Although hard disk drive (HDD) based storage arrays (HSA) still dominate the market, all-flash arrays (AFA) composed of solid state drives (SSD) have received a considerable attention recently due to their random access nature and superior parallel I/O potential [1]. Various manufacturers including NetApp, EMC, Pure Storage, and HP/3Par have already launched their AFAs. In addition to the storage arrays, other parallel storage scenarios include key-value stores and parallel/distributed file systems built on clusters, and SSDs themselves featuring various levels of internal parallelism. In the rest of this paper, we use storage arrays as an example parallel storage system; however, the proposed techniques are sufficiently generic and can easily be adapted to other parallel storage scenarios.

Striping and declustering are two common techniques for data placement in parallel storage systems. The data space is partitioned into disjoint regions (blocks, stripes, or chunks) and distributed over independent storage components (called hereafter *disks*) so that requests spanning different disks can be retrieved in parallel. As well as HSAs, existing AFAs come with traditional RAID [2] techniques originally designed for HDDs to *statically* distribute data over disks. In addition, several advanced declustering methods have also been proposed for the static placement of data [3, 4]. For better parallelism, a common approach in declustering is to assume a certain disk access pattern. For instance, the technique proposed in [4] is optimized specifically for range queries where a range of values are searched in a multi-dimensional dataset as in databases. However, the disk access patterns of applications change over time, and many realistic workloads are known to be skewed in practice [5, 6]. Also, various applications using the same storage system can have different access patterns. Therefore, a data placement optimized for a specific disk access pattern may not perform well in general. In order to utilize parallel disks to their full potential, parallelism-aware *dynamic* data layout optimization is necessary.

This paper proposes a framework for self-optimizing parallel storage systems that can automatically adapt themselves to skewed, changing, and coexisting disk access patterns. The proposed framework detects block-level disk access correlations using association rule mining techniques, periodically redistributes the correlated blocks into separate disks for improved I/O parallelism using graph coloring and bin packing techniques, and while doing this introduces a minimal amount of data movement using min-cost flow techniques. The main contributions of this work can be summarized as follows:

- We propose a novel I/O parallelism-aware self-optimization framework based on access correlations.
- We theoretically analyze the complexity of the correlation-based placement problem and show that it is NP-complete using reduction from graph coloring.
- We formulate placement and reorganization as separate optimization problems and propose an efficient heuristic for placement and a polynomial-time optimal algorithm for reorganization.
- We provide extensive performance evaluations and cost analysis on both HSAs and AFAs using real workloads.

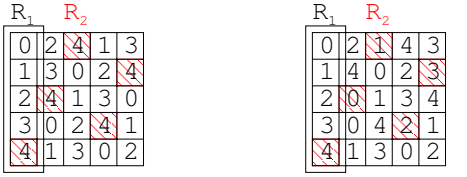
978-1-5090-5411-4/16 $31.00 © 2016 IEEE
DOI 10.1109/HiPC.2016.26

II. Background and Motivation

In this section, we first provide the preliminaries of data placement, access patterns, and correlations in parallel storage systems. Next, we present the motivation and related work.

A. Data Placement and Parallel Access Patterns

Efficient data placement is crucial in parallel disk architectures to enable concurrency and high performance parallel I/O. Figure 1(a) presents an example data placement strategy called *periodic declustering* [3], where a two-dimensional dataset composed of 25 data blocks is distributed over 5 disks. Each square of the grid denotes a data block (a disjoint region of the dataset) and the number in the square denotes the disk that holds the block. In this dataset, a sample disk request can be an $i \times j$ range query having i rows and j columns. For retrieval of $i \cdot j$ blocks from N disks, the best we can expect is $\lceil \frac{i \cdot j}{N} \rceil$ *parallel accesses*, and this happens if the blocks of the request are spread across the disks in a balanced way.



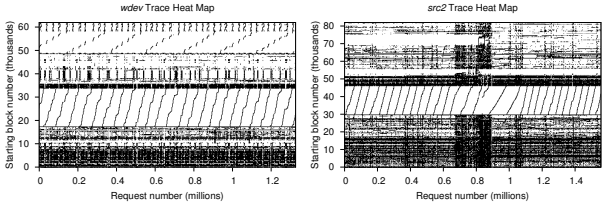
(a) Static Layout (b) Dynamic Layout

Fig. 1: Data Layout and Parallel Access

Consider the request R_1 shown in Figure 1(a) with a rectangular frame, which is a 1×5 range query requesting 5 data blocks. Since each of the requested data blocks are stored in a different disk, retrieval of this request requires 1 parallel access, which is the optimal number of parallel accesses we can achieve for 5 blocks and 5 disks ($\lceil \frac{i \cdot j}{N} \rceil = \lceil \frac{1 \cdot 5}{5} \rceil = 1$). This optimal result for the request R_1 is not actually surprising since *periodic declustering* is optimized for range queries. If the storage system receives range queries exclusively, then such a placement technique is expected to perform well. However, the disk I/O performance of applications will decrease dramatically when they change their parallel access patterns and issue requests other than range queries. For example, consider the request R_2 shown in Figure 1(a) marked with red diagonally pattered blocks. R_2 is also composed of 5 data blocks as R_1; however, R_2 is not a range query, it is an arbitrary query. In this case, since all the blocks of R_2 are stored in disk 4, retrieval of this request requires 5 parallel accesses, resulting in a sub-optimal device concurrency and parallel I/O performance. In order to boost the concurrency of parallel disks, a self-optimizing data reorganization framework that can automatically adapt to changing and coexisting disk access patterns is necessary. We believe that such a framework can be built using block correlations within each request and among consecutive requests.

B. Block Correlations

Block correlations indicate that two or more blocks are correlated if they are requested together, or if they are requested within a very short time interval so that they are queued and handled together by the storage sub-system [7]. For example, consider the request R_1 from Figure 1(a) again and assume that the notation $[i, j]$ denotes the block in row i and column j. Based on this notation, the correlated blocks for the request R_1 are $[0,0],[1,0],[2,0],[3,0],[4,0]$. When we consider the request R_2 itself, we can observe that the blocks $[0,2],[1,4],[2,1],[3,3],[4,0]$ are also correlated. Block correlations can exist intra-request (from the same request) or inter-request (among different requests). Once the block correlations are detected, I/O parallelism can be improved dramatically by placing the correlated blocks in separate disks. In addition, correlation strengths can be determined based on their frequencies and can be used in heuristic methods when the optimal data layout for all requests is not feasible. Consider the requests R_1 and R_2 in Figure 1(a) again and assume that these two requests are made frequently. Remember that R_2 was yielding sub-optimal device concurrency and I/O parallelism by requiring 5 parallel accesses while R_1 was optimal by requiring 1 parallel access. Based on their block correlations listed above, data layout optimization for both R_1 and R_2 can be performed as in Figure 1(b), resulting in optimal I/O parallelism for *both* requests.



(a) Web Server (b) Version Control Server

Fig. 2: Storage Heat Maps of Real Applications

Block correlations are commonly encountered in storage workloads. We observed this while analyzing the storage traces of various enterprise and production servers from Microsoft [8]. To illustrate, Figure 2 shows the storage heat maps of a web server and a version control server. The x-axis in the figures shows the request number starting from zero in chronological order and the y-axis shows the starting block number (may be shifted on y-axis to eliminate gaps) of the requests. Blocks falling in a certain range of x values are generally considered to be correlated. Existing patterns in the figures clearly indicate the occurrence of block correlations, and heavy repetition of these patterns motivates the use of a correlation-based dynamic layout optimization. In addition to Microsoft applications, Google also reveals the existence of block correlations in their workloads by specifically designing their Spanner database to co-locate multiple directories that are frequently accesses together [9]. As well as the correlations, various additional information can also be extracted from the storage traces and utilized in reorganization, including correlation strengths and correlation types (Read/Write).

III. Related Work

Although static placement received more attention [1–4], dynamic data reorganization techniques targeting single- and multi-disk HDDs were also proposed. Among the ones

targeting single disks [7, 10–12], block correlations were first utilized in [7] for prefetching purposes and arranging correlated blocks contiguously in a single HDD for reduced seek time. Similar techniques were also applied in [10] and [11] to reorganize hot data blocks sequentially on a dedicated partition of a single HDD, and in [12] to replicate hot blocks in free disk space. However, these techniques target single HDDs for reduced seek time without focusing on I/O parallelism.

Dynamic data reorganization for HDD-based parallel disks was first investigated in [13], in which the authors detect hot disks and use disk cooling heuristics to move some data from hot disks to cooler disks. Their heuristic keeps cooling disks until their heat drop below a given threshold. In order to achieve this, the authors assume that certain disk access patterns can be estimated a priori without a disk access monitoring mechanism, which is against the spirit of dynamic data reorganization. Authors in [14] focused on frequent seeks occurring within the individual disks of an HDD array and proposed a data reorganization technique decreasing the seek distance instead of focusing on device concurrency and I/O parallelism. A dynamic data reorganization technique for hybrid storage arrays composed of SSDs and HDDs is proposed in [15], where the blocks are labeled as *write-exclusive*, *read-exclusive*, and *read-write*, then placed in storage devices considering the distinct features of HDDs and SSDs such that read-heavy blocks are directed to SSDs and write-heavy blocks are directed to HDDs. This approach does not consider I/O parallelism and can be applied in hybrid storage systems together with our proposed parallelism techniques. Most recently, authors in [16] extended the hot block prefetching idea proposed in [7] for single disks to parallel disks and applied prefetching, where additional copies of the frequently accessed blocks are created and cached.

IV. Dynamic Layout Optimization Framework

Our proposed dynamic data layout optimization framework is composed of the following four building blocks:

- **Disk I/O Monitoring Module:** Monitors disk I/O requests and records the block numbers that are requested together in *sessions*.
- **Data Analysis Module:** Analyzes the recorded sessions to detect block correlations.
- **Placement Planning Module:** Plans a new placement strategy based on the detected block correlations.
- **Data Reorganization Module:** Performs the reorganization by minimizing the data movement.

An overview of the proposed framework is provided in Figure 3. In summary, our framework monitors disk I/O requests directed to the storage system and periodically optimizes the data layout based on the detected disk access correlations. The rest of this section describes each component of the proposed framework in more detail.

A. Disk I/O Monitoring Module

Block level requests can be monitored using a disk I/O tracing tool such as `blktrace` [17], which can provide detailed information about each disk request including its timestamp,

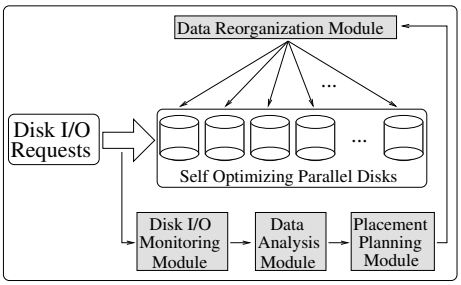


Fig. 3: Overview of the Proposed Framework

event type (queued, completed, etc.), process name/ID of the application making the request, request type (Read/Write), starting block ID of the request, and the size of the request in blocks. Using this information, the monitoring module can divide the requests into sessions, where each session represents a set of blocks requested (or handled) together by the storage subsystem, and store these sessions line by line. A sample monitoring output is provided in Figure 4(a) composed of three sessions, indicating that blocks 1, 2, and 3 are requested in the first session, blocks 1, 3, and 4 are requested in the second session, and blocks 4 and 5 are requested in the third session.

```
1 2 3
1 3 4
4 5
```

(a) Monitoring Output

```
1 2 (1)
1 3 (2)
1 4 (1)
2 3 (1)
3 4 (1)
4 5 (1)
```

(b) Analysis Output

Fig. 4: Monitoring and Analysis Outputs

B. Data Analysis Module

The data analysis module is responsible for analyzing the sessions generated by the monitoring module and finding correlations between blocks using association rule mining techniques. A common way to determine association rules is using Frequent Itemset Mining (FIM) algorithms [18].

1) Frequent Itemset Mining (FIM): FIM algorithms can find block correlations as well as the frequency of the correlations indicating their strengths. The original motivation of FIM was the need to analyze supermarket customer behavior to discover which products were purchased together and with what frequency. Using this information, supermarkets can place correlated products next to each other on the shelves to boost their sales. Our layout optimization idea is motivated by the product placement idea of supermarkets, with a minor difference that we propose to place the correlated blocks into separate disks to boost concurrency and parallel I/O performance. FIM algorithms accept a minimum frequency amount called *support* so that the algorithm can skip the correlations occurring less than this value.

Mining the sessions provided in Figure 4(a) using the FIM techniques for $support = 1$ returns the output shown in Figure 4(b). Note that mining only for the correlated block *pairs* (set size = 2) is enough for our purposes since the pairs will include larger size correlations within themselves. In each

row of the FIM output in Figure 4(b), the first two numbers represent the ID of the correlated blocks and the third number in parentheses represents the frequency of this correlation. In other words, blocks 1 and 2 are requested together once, blocks 1 and 3 are requested together twice, and so on.

C. Placement Planning Module

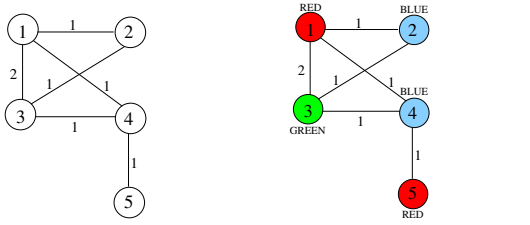
The aim of the placement planning module is to use block correlations produced by the data analysis module and to plan a new correlation-based data placement strategy that boosts concurrency and I/O parallelism. However, optimal placement is a challenging problem, even in simplified cases. In order to show its complexity, we formulate the correlation-based basic placement planning as an optimization problem as follows:

Problem Definition 1. Basic Layout Optimization Problem (BLOP): Given a set C of correlated block pairs (i, j), and N disks; plan a placement strategy so that for every block pair $(i, j) \in C$, blocks i and j are stored in different disks.

Theorem 1. *BLOP is NP-complete and equivalent to the* ***proper (vertex)*** **k-*coloring*** *problem [19] for* $k = N$.

Proof: Construct an undirected *correlation graph* $G(V,E)$ such that each vertex $v \in V$ represents a unique block in C and each edge $(u,v) \in E$ represents a correlation between blocks $(u,v) \in C$, as shown in Figure 5(a) for the correlations provided in Figure 4(b). Then, the proper k-coloring of G, which colors the vertices of G with a maximum of k colors such that adjacent vertices receive different colors (as shown in Figure 5(b)) is equivalent to BLOP for $k = N$ where each color represents a unique disk. Since proper k-coloring is NP-complete, BLOP is also NP-complete. ■

Based on the above proof, we are able to reduce our simplified placement planning problem into a type of classic NP-complete Vertex Coloring Problem (VCP) called the *proper k-coloring*. Vertex coloring is a well studied problem and various heuristics are proposed, analyzed, and optimized. Therefore, by reducing our problem to this well-known problem, we can adapt these heuristics as our placement planning strategy instead of proposing an entirely new and unproven heuristic.



(a) Original Graph (b) Colored Graph

Fig. 5: Placement Planning Output

Although BLOP outlines the main purpose of our placement planning strategy, it is simplified and requires additional considerations to be applied in real world settings. First of all, since the number of vertices (blocks) in the graph is expected to be much larger than the number of colors (disks) available, $|V| \gg N$; a proper N-coloring of the generated graph $G(V,E)$ is generally not expected to be feasible. Therefore, a more practical approach is to color the graph by minimizing the conflicts, the number of edges having both vertices with the same color. This technique is called *soft coloring* [20]. Also, in real world settings each disk has a maximum capacity not to be exceeded, and therefore disk capacities should be considered while planning the placement. In order to handle disk capacities, we can use traditional *bin-packing* techniques, where every color (disk) is assigned a maximum capacity in bytes based on the disk capacity limit and every vertex (block) has a weight in bytes based on the block size. Including these two modifications, our placement planning problem becomes equivalent to another NP-complete problem called the *Min-Conflict Bin Packing (MCBP)*. We skip the equivalence proof here since it follows from the proof of Theorem 1, and directly provide the definition of MCBP:

Problem Definition 2. Min-Conflict Bin Packing (MCBP): Given a set I of items i of size w_i, N bins of size W, and a conflict graph $G = (I,E)$ where $(i, j) \in E$ if items i and j cannot be packed in the same bin, compute the minimum number of conflicts that must occur if the set I is packed in N bins of size W.

MCBP is defined in [21] as a combination of soft coloring and bin packing problems, and an effective heuristic is provided for problems closer to coloring than packing as in our case. Based on this heuristic, initially colors are mapped to vertices randomly. Next, a random vertex i having conflicts with some other vertices is chosen, and the color c that *locally* minimizes the total number of conflicts without violating the capacity constraint of c is mapped to i. Ties are broken randomly and the algorithm continues until no constraint is violated or until some given termination criterion is met. Our proposed placement planning heuristic tailors the MCBP heuristic to our specific problem type and it has the following four properties:

- **P1** Instead of starting with a *random* color-to-vertex mapping, we map the colors to vertices based on the *original* data placement. This property allows us to eliminate unnecessary data movements in the future.
- **P2** Instead of a *random* iteration order, we perform local optimization in decreasing order of *Total Correlation Frequency (TCF)* of the vertices, where the TCF of a vertex can be calculated by summing the weights of all its edges. This property prioritizes the movement of blocks having more correlations with other blocks.
- **P3** We store the correlation strengths in the edge weights and our conflict calculation during the local optimization process considers these edge weights. In other words, we count each conflict based on the strength of the correlation instead of counting each of them only once. This way, block correlations occurring more frequently than others are given more importance.
- **P4** If there are more than one candidate color that can be mapped to a vertex, we break the tie by choosing the color having more available capacity instead of a random selection. This property helps to balance disk loads.

Our placement planning heuristic is provided in Algorithm 1. Each vertex $v \in \{1,\ldots,|V|\}$ represents a block and each color $c \in \{1,\ldots,N\}$ represents a disk, where N is the total number of disks in the system.

Algorithm 1 Placement Planning Heuristic

```
 1: Color the vertices based on the original data placement
 2: caps (1, ..., N) = initial disk capacities in bytes
 3: for v ∈ V do
 4:     v.tcf = 0
 5:     for u ∈ v.adj do
 6:         v.tcf += (u,v).weight
 7: S (1, ..., |V|) = vertices sorted by tcf in descending order
 8: repeat
 9:     for i ← 1 : |V| do
10:         v ←S[i]
11:         confs (1, ..., N) = 0
12:         for u ∈ v.adj do
13:             confs[u.clr] += (u,v).weight
14:         for c ← 1 : N do
15:             if (confs[c] < confs[v.clr] and caps[c] + w < W ) or
                (confs[c] == confs[v.clr] and caps[c] + w < caps[v.clr])
16:                 caps[v.clr] -= w
17:                 v.clr = c
18:                 caps[c] += w
19: until Δconficts < ε
```

The heuristic begins by initializing the graph and the necessary data structures at lines 1–7. At line 1, a vertex v is given a color c if the block corresponding to v was originally stored in the disk corresponding to c (P1). Line 2 initializes the *caps* array using the initial disk capacities in bytes. Next, TCF values of the vertices are calculated at lines 5–6 and sorted in descending order at line 7. Local optimizations are performed at lines 9–18 starting from the vertex having the maximum TCF value (P2). Line 11 initializes the *confs* array that records how many conflicts would result if vertex v was assigned to color c. Lines 12–13 fill the *confs* array by visiting all the adjacent vertices of v and summing their edge weights (P3). At line 15, we map the color c to vertex v if c has fewer conflicts than v's current color respecting the disk capacities. If c and v's current color yield the same number of conflicts, then we break this tie by choosing the color that has the most available capacity (P4), where W is the maximum *safe* disk capacity in bytes and w is the block size in bytes. A *safe* value for W can be determined based on the load balancing threshold of the disks and the maximum block movements permitted to a disk. Local optimizations shown in lines 9–18 are repeated until the number of conflicts does not improve from the previous iteration by a predefined percentage ε. Based on our experiments, we found $\varepsilon = 5\%$ to be an efficient stopping criterion according to the number of iterations it causes and the performance improvement it yields in the framework. Although it was not necessary for our case, an additional condition can be added in line 19 to bound the number of iterations by a constant.

1) Complexity Analysis: Our graph is undirected and we use an adjacency list for iteration. Since the size of an adjacency list in an undirected graph is $2|E|$, iterating through all the vertices in the adjacency list is $O(|V|+|E|)$, which is performed at lines 3–6 and lines 9–13. At line 7, sorting the vertices by their TCFs is $O(|V|\log|V|)$. Local optimizations are repeated a constant number of times at line 8, bounded by the condition at line 19, and the for loop at line 14 is also iterated a constant number of times since the number of colors (disks) N is limited. Therefore our heuristic has the worst-case time complexity of $O(|V|\log|V|+|E|)$. Note that our graph type is expected to be on the sparse side.

D. Data Reorganization Module

The purpose of the data reorganization module is to map the colors returned by the placement planning heuristic to the disks so that the block movement during reorganization is minimized. The proposed placement planning heuristic initially assumes a preliminary color-to-disk mapping based on property P1 of the heuristic, and the motivation behind this was to keep the original data placement as stable as possible, eliminating unnecessary block movements. However, since the original block colors can change considerably at the end of the heuristic, it is necessary to reconsider the color-to-disk mapping again to guarantee the minimum block movement. Therefore, by using the colored graph returned by the placement planning module, the data reorganization module reorganizes the block placement on the disks considering the following two criteria:

- Each color is mapped to a separate disk.
- The number of block movements are minimized.

A brute force solution satisfying the two criteria listed above would work as follows:

- Consider all possible $\frac{N!}{(N-C)!}$ color-to-disk mappings for C colors and N disks; where $C \leq N$
- Calculate the amount of blocks to be moved for each possible mapping.
- Choose the mapping yielding the minimum amount of block movements.

This brute force solution guarantees the optimal solution, and it might work efficiently for small values of N or C. However, it is obvious that this brute force technique would require unacceptably high execution time for larger values of N due to its factorial time complexity. Therefore, the brute force technique is impractical for real systems, even for storage systems having more than $N = 13$ disks. A better solution would be finding the optimal mapping in polynomial time, which can be achieved by constructing the problem as a flow network and solving it using the minimum cost flow techniques [22] as follows:

- Using the colored graph returned by the placement planning module (see Figure 5(b)), generate a directed flow graph $G(V,E)$ as shown in Figure 6 such that for each color and for each disk in the system, a vertex is created. In addition, two more vertices called source (s) and sink (t) are created.
- Draw an edge from source to every *color* vertex, from every *color* vertex to every *disk* vertex, and from every *disk* vertex to sink.
- Set the capacities of all the edges to 1.

- Set the costs of the edges to 0 except the edges between a *color* vertex and a *disk* vertex. The cost of the edge between a *color* vertex and a *disk* vertex is set to the amount of the data movement caused by such mapping.
- Set the source as the *supply* node supplying flows in the amount of colors C, sink as the *demand* node demanding C flows, and the rest of the vertices as the *transshipment* nodes not supplying or demanding any flow.

Running a minimum cost flow algorithm on the constructed flow graph shown in Figure 6 will return the optimum color-to-disk mappings yielding the minimum data movement. The flow directions shown in Figure 6 with thick lines indicate the optimum reorganization schedule for the example shown in Figures 4(a), 4(b), 5(a), and 5(b) assuming an initial round-robin placement. Minimum cost flow problems can be solved in polynomial time and complexity can be reduced for graphs with unit capacities as in our case. Recently, authors in [23] proposed a $O(|E|^{3/2}\log(|V|C_{max}))$ complexity algorithm for unit capacity graphs, where C_{max} is the maximum cost value.

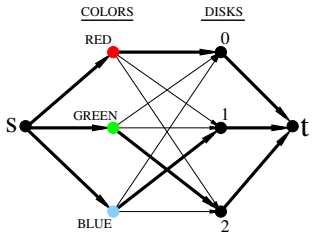


Fig. 6: Min-Cost Flow Graph for Minimum Data Movement

E. Additional Optimizations for HDD-based Systems

Flash is an inherently random access medium not including the variable access time of spinning disks. Flash accesses the desired address with almost uniform access time, regardless of where the requested data resides in the SSD's logical address space. *Therefore, concurrency is very crucial for SSD-based all-flash arrays.* On the other hand, random I/O in an HDD requires the device to first position the read/write head on the correct cylinder (seek time), and then wait while the disk rotates to the correct sector (rotational latency); causing a variable access (positioning) time. Although the proposed concurrency techniques will immediately boost the performance of HDD-based parallel disks for many realistic workloads, the internal disk geometry of HDDs should not be ignored completely since an additional performance improvement can be achieved for certain workloads if the access sequentiality of the individual HDDs can be preserved as much as possible while boosting the concurrency of parallel HDDs.

As an illustration, assume a request R, which is composed of ten correlated blocks and assume also that these blocks are stored in a single disk sequentially. In this case, retrieval of R requires a single positioning time (seek+rotational latency) since the blocks are sequential. However, if these 10 blocks are broken and placed into ten different HDDs, then the retrieval of R will require ten *parallel* positioning operations, where the maximum of these ten positioning times will determine the actual positioning time of R. Since we do not have any control over the initial location of the read/write heads, the second case has a higher probability to produce a larger positioning time in this example. In addition to the positioning time, transfer time is another factor affecting the response time of a disk request. Positioning time is generally assumed to be the dominant factor in response time if the block size is small such as 512 B to 4 KB as commonly used in local file systems. However, transfer time can be the dominant factor for larger block sizes. For instance, the block size in distributed file systems can be 256 MB, requiring around 20 seconds to transfer 10 sequential blocks from a single HDD with 1 Gb/s of transfer rate.

Considering the sequentiality of the correlated blocks in individual disks and the total size of these sequential blocks, the proposed framework can be optimized even further for HDD-based parallel disks by grouping the sequential blocks from the same HDD and reorganizing these groups together for better concurrency without breaking their sequentiality. For this purpose, before generating the graph structure shown in Figure 5(a), the placement planning module needs to group the sequential blocks located in the same disk and create a single vertex in the graph for each group. The resulting graph will be a hybrid graph including single block vertices not having any sequential correlations with other blocks and group vertices representing sequentially correlated blocks. Edges of a group vertex and its edge weights will continue to represent the correlations of the particular group with other single or group vertices and their correlation strength, respectively, and all these information can be calculated easily by considering the group memberships. In addition, a maximum group size in bytes should be set based on the transfer rate of the disks so that groups larger than this size should not be allowed since larger group size will work against the concurrency. This way, grouping in large block sizes will also be eliminated.

V. Evaluation

In this section, we share our performance evaluation and cost analysis using real application workloads.

A. Experimental Setup

Our simulations were run in DiskSim 4.0 [24] using the SSD patch from Microsoft Research [25]. DiskSim is a widely used storage system simulator available from Carnegie Mellon Parallel Data Lab. It is able to simulate small to medium sized storage systems of up to an array of 100 HDDs. Microsoft Research's SSD patch is designed as an extension of DiskSim and it is able to simulate up to an array of 14 SSDs.

We ran experiments on both HDD-based storage arrays (HSA) and SSD-based all-flash arrays (AFA). In both cases, we used the maximum number of disks supported by the simulator, 100 for HSAs and 14 for AFAs. Our HDDs are modeled after the Seagate Cheetah 15K ST3146855FC, which is a validated disk model by Disksim. Our SSDs are modeled after Samsung's K9XXG08UXM as done by the SSD patch. In both HSAs and AFAs, we set the block/page size to 512B to match the block size of our workloads. As our performance metric, we use read and write I/O latency (response time) values reported by DiskSim.

1) Frequent Itemset Miner: In our data analysis module, we used the Borgelt implementation of the Eclat miner [18] for finding correlations between blocks. This implementation is efficient, highly customizable, and easy to adapt for our framework. It accepts a support parameter, and also allows mining only the correlated pairs by setting the minimum and maximum number of items both to two.

2) Initial Data Placement: Our workloads are from various HDD-based storage systems with a 512 B block size. Using 512 B blocks/page size for HDDs/SSDs, we initially distribute the blocks/pages over disks using a set of Zipf-like distributions [26]. Zipf-like distributions allow us to control the skew of the parallel access pattern, and observe the behavior of our framework under various skews. Based on these Zipf-like distributions, the relative probability of accessing a block on the i-th most popular disk is proportional to $1/i^{\alpha}$ for $0 \leq \alpha \leq 1$. Various natural data access patterns such as web servers and general purpose key-value stores have been shown to be skewed in a manner that can be approximated by a Zipf-like distribution using higher values of α [5, 26]. Zipf-like distributions are commonly used for controlled performance evaluation in storage system research [27, 28]. We tested the performance of our framework for $\alpha = \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. For $\alpha = 0.0$, the disks are accessed evenly, and for $\alpha = 1.0$, the distribution follows a true Zipf distribution (rather than Zipf-like) where the parallel access pattern is skewed towards the most popular disk.

3) Workloads: We evaluate our framework using five publicly available [29] real-world storage workloads provided by Microsoft [8]. These traces include block level I/O requests of various enterprise and production servers running in Microsoft, including a web server, a version control server, a research projects server, and a hardware monitoring server. These traces are widely used in literature [15, 16], and include a mix of applications with various I/O behaviors as shown in Table I.

TABLE I: Trace Statistics

Trace	Application	R/W %	Avg. R/W Size (KB)
hm	Hardware Monitor	45.14 / 54.86	9.0 / 7.0
rsrch	Research Projects	9.43 / 90.57	7.5 / 8.7
src2	Version Control	11.02 / 88.98	6.0 / 6.5
stg	Staging Server	12.06 / 87.94	9.6 / 8.5
wdev	Test Web Server	17.56 / 82.44	12.2 / 8.1

4) Evaluation Methodology: Different workloads include different numbers of requests, affecting a fair comparison of results based on the length of the history observed. In order to have consistency between workloads, we consider the first 100,000 requests of each workload. Next, we split each trace into a training set and a testing set. The training set is the first 50% of the requests, and the testing set is the remaining 50%. After splitting the traces, only the training set is used by our framework for detecting the patterns and determining the reorganization plan, and only the testing set is used to evaluate the performance of our reorganization framework. *This separation is crucial for realistic evaluation since no improvement will be achieved if the detected patterns based on the history are not re-observed in the future.*

B. Experimental Results

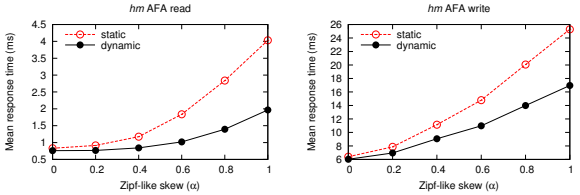
We first share the I/O performance evaluation for AFAs and HSAs, including the effect of additional HDD optimizations described in Section IV-E. Next, we analyze the cost of the proposed framework in terms of the data movement and explain how this cost can be reduced by demonstrating the trade-off between the data movement and the I/O performance.

1) SSD-based All-Flash Arrays (AFA): Figures 7, 8, 9, 10, and 11 show the disk I/O latency (response time) performance of our self-optimizing *dynamic* framework compared to the *static* (no reorganization) placement for AFAs. Each figure corresponds to a separate workload, where the left sub-figure shows the read performance and the right sub-figure shows the write performance. In each graph, the x-axis shows the α of the Zipf-like distribution, and the y-axis shows the mean response time over all requests. $\alpha = 0.0$ represents a parallel access pattern that is close to optimal and the parallel access pattern becomes more skewed as α increases.

Disk I/O latency of the static placement consistently increases as the parallel access pattern becomes more skewed. However, for many workloads, our dynamic self-optimization framework can keep the I/O latency as stable as possible without being affected by the skew in the parallel access patterns of the workloads. This can be clearly observed from the read performance of *src2*, *stg*, and *wdev* traces presented in Figures 9(a), 10(a), and 11(a), respectively, and the write performance of *rsrch* and *src2* traces shown in Figures 8(b) and 9(b), respectively. Considering all workloads and all α values, our framework achieves 111% Read, 52% Write, and 53% overall (R+W) performance improvement over the static placement on average.

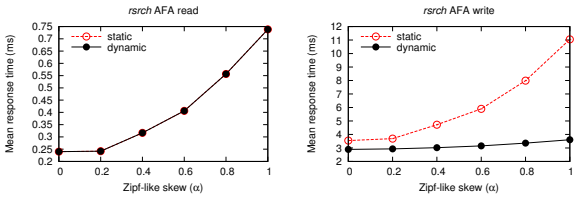
Although I/O performance of some workloads improves equally for for both read and write requests (*hm, src2*), some traces achieve a better read performance improvement (*stg, wdev*), and some traces achieve a better write performance improvement (*rsrch*). There may be a few factors causing this behavior: (i) The I/O characteristics of the workload (Read or Write intensive), (ii) The number of block correlations discovered from the specific request type (Read/Write), and (iii) The recurrence rate of the discovered correlations in future accesses. For instance, *rsrch* is a write intensive workload where 90% of the I/O requests are writes (see Table I). In addition, the limited number of correlations discovered from read requests are not re-accessed in the future. Therefore, although 89% of write performance improvement is achieved on average for all α values, no considerable read performance improvement is observed. *For applications having such behavior, the proposed framework can be configured such that only certain request types (only Reads or only Writes) are monitored and block correlations among this particular request type are analyzed and optimized.*

2) HDD-based Storage Arrays (HSA): In Figures 12, 13, 14, 15, and 16, we present the disk I/O latency performance of our framework for HSAs. In addition to the *static* and *dynamic* results as reported for the AFA case, here we also share the effect of our additional HDD optimizations proposed



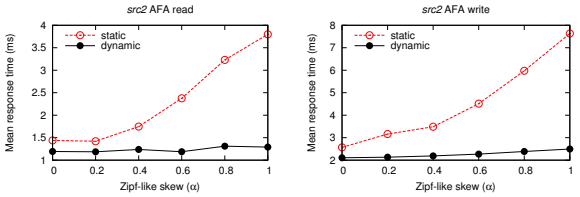
(a) Read Performance (b) Write Performance

Fig. 7: I/O Latency Performance - *hm* Trace - AFA



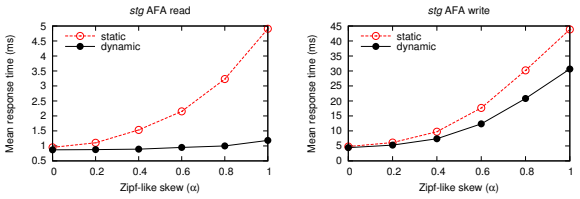
(a) Read Performance (b) Write Performance

Fig. 8: I/O Latency Performance - *rsrch* Trace - AFA



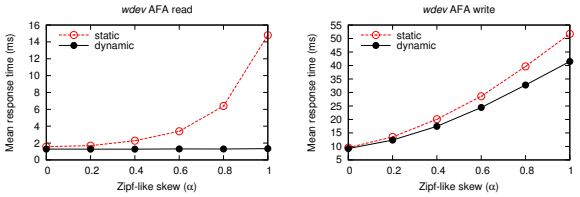
(a) Read Performance (b) Write Performance

Fig. 9: I/O Latency Performance - *src2* Trace - AFA



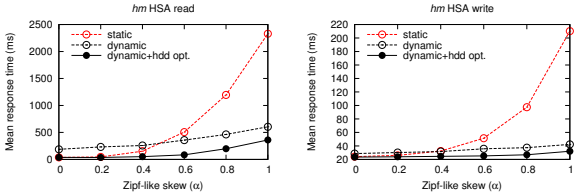
(a) Read Performance (b) Write Performance

Fig. 10: I/O Latency Performance - *stg* Trace - AFA



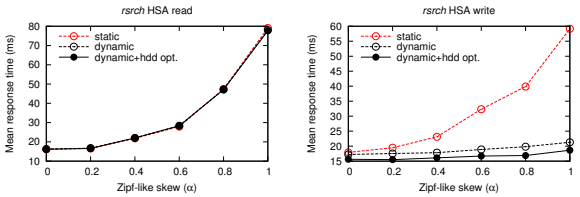
(a) Read Performance (b) Write Performance

Fig. 11: I/O Latency Performance - *wdev* Trace - AFA



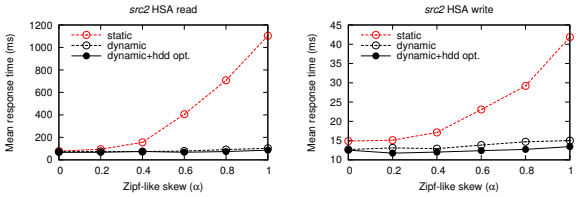
(a) Read Performance (b) Write Performance

Fig. 12: I/O Latency Performance - *hm* Trace - HSA



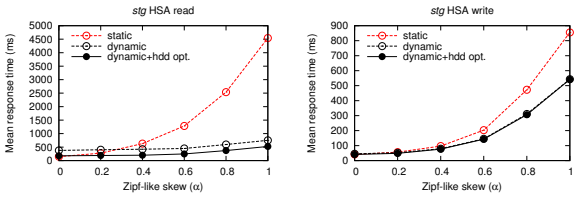
(a) Read Performance (b) Write Performance

Fig. 13: I/O Latency Performance - *rsrch* Trace - HSA



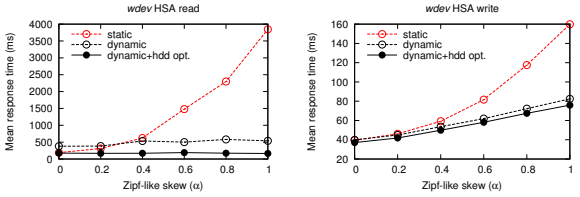
(a) Read Performance (b) Write Performance

Fig. 14: I/O Latency Performance - *src2* Trace - HSA



(a) Read Performance (b) Write Performance

Fig. 15: I/O Latency Performance - *stg* Trace - HSA



(a) Read Performance (b) Write Performance

Fig. 16: I/O Latency Performance - *wdev* Trace - HSA

in Section IV-E, denoted by *dynamic+hdd opt.* The results are similar to the AFA case; except, the damage of not performing dynamic reorganization is more severe for HSAs since HDDs are substantially slower devices compared to SSDs, especially in their read performance. As it can be observed from the figures, response time for the static placement generally increases exponentially as the access pattern gets skewed. However, similar to the AFA case, our framework keeps the response time and I/O performance of the storage system as stable as possible, especially for the read requests of the *src2*, *stg*, and *wdev*, and the write requests of the *hm*, *rsrch*, and *src2*. Considering all workloads and all α values, our framework including the additional HDD optimizations achieves 366% Read, 82% Write, and 170% overall (R+W) performance improvement over the static placement on average.

Performing additional HDD optimizations is especially crucial for workloads having frequent sequential access patterns in their individual HDDs. When we reorganize the sequen-

TABLE II: Cost vs. *Overall(R+W)* Performance - AFA

Trace	Support = 1		Support = 5		Support = 10	
	Cost(*MB*)	Per.(%)	Cost(*MB*)	Per.(%)	Cost(*MB*)	Per.(%)
hm	102.63	30.46	5.70	9.86	0.95	5.66
rsrch	76.19	89.26	4.09	41.90	0.76	32.89
src2	63.03	97.14	3.37	66.23	1.46	60.07
stg	83.40	31.97	5.94	13.57	2.22	10.64
wdev	82.09	16.97	9.39	9.40	1.23	6.28
AVG	81.5	53.2	5.7	28.2	1.3	23.1

TABLE III: Cost vs. *Overall(R+W)* Performance - HSA

Trace	Support = 1		Support = 5		Support = 10	
	Cost(*MB*)	Per.(%)	Cost(*MB*)	Per.(%)	Cost(*MB*)	Per.(%)
hm	109.94	258.19	8.03	103.68	1.09	14.38
rsrch	89.50	87.02	4.98	40.78	0.84	32.63
src2	78.96	172.58	4.05	121.01	1.70	37.09
stg	107.31	78.20	7.61	9.35	2.61	5.29
wdev	101.68	256.54	13.19	194.92	1.36	28.56
AVG	97.5	170.5	7.6	94	1.5	23.6

tial blocks of individual disks together as described in Section IV-E, we achieve an additional 216% Read, 24% Write, and 86% overall (R+W) performance improvement over the static placement averaged over all workloads and α values. The workloads that are most benefited from the additional HDD optimizations are *hm*, *stg*, and *wdev* due to their sequential read access patterns. As discussed in Section IV-E, performing reorganization without considering the internal disk geometry of HDDs can actually damage the I/O performance of certain workloads. We can clearly observe this issue in Figures 12(a), 15(a), and 16(a) for the *hm-read*, *stg-read*, and *wdev-read* workloads, respectively, for lower values of α.

3) Cost Analysis: In this section, we analyze the cost of the proposed framework and demonstrate how this cost can be controlled easily using the support value of FIM. As mentioned in Section IV-B1, FIM algorithms accept a minimum frequency value called *support* to ignore correlations occurring less than this value. By increasing the support value, it is possible to decrease the number of correlations to be passed to the placement planning, and reorganization modules considerably. The reduced number of correlations will cause faster reorganization times. On the other hand, reducing the number of correlations is also expected to cause a reduced I/O performance as a side effect.

In Tables II and III, we illustrate the trade-off between the reorganization cost (blocks moved in MB) and the I/O performance improvement (in %) using various support values for AFAs and HSAs, respectively. The provided values are averages for all α. For the previously discussed experimental results using $support = 1$, our framework moves 81.5 MB of data for AFAs and 97.5 MB of data for HSAs on average, and gains an average of 53.2% and 170.5% performance improvement, respectively. Such and improvement with less than 100 MB of data movement was possible due to frequent re-access of the same data and low inter-arrival time between requests. On average, the testing traces request 386 MB of data, where only 138 MB of this data is unique. In addition, 77.8% of the requests have inter-arrival times of less than 100 microseconds creating a bursty I/O pattern and convoy effect. Nevertheless, data movement cost can be decreased even further by increasing the support value and slightly losing from the performance. For AFAs shown in Table II, increasing the support from 1 to 5 reduces the cost an average of 15.6x with 2.2x loss in performance. Moreover, increasing the support from 1 to 10 reduces the cost an average of 71.2x while decreasing the performance improvement 3.1x considering all workloads. Table III presents a similar trade-off for HSAs; where increasing the support from 1 to 10 reduces the cost an average of 74x while causing only 9.8x performance loss.

VI. Discussion

The proposed framework can directly be applied to various parallel storage scenarios including key-value stores, parallel/distributed file systems, and internal parallelism of SSDs. In this section, we provide our insights for further applicability of our framework to heterogeneous storage systems and replicated datasets. In addition, we also provide further discussions on reorganization cost and frequency.

A. Heterogeneous Storage Architectures

In addition to the availability of homogeneous HSAs and AFAs, hybrid storage arrays composed of SSDs and HDDs have also emerged recently. In hybrid arrays, reorganization should also consider the individual device characteristics while boosting the parallelism. An important characteristic of SDDs is that they provide a significantly faster read performance compared to HDDs. Considering this, our placement planning heuristic can be adjusted so that correlated blocks are distributed considering their correlation frequencies stored on the graph edges. Since these frequency values indicate the popularity of the correlations, popular blocks having high correlation frequency can be reorganized to faster SSDs since these blocks are expected to be retrieved more frequently.

Another important characteristic of SSDs is that they provide faster reads than writes and additional writes negatively affect the endurance of SSDs. Considering these characteristics, authors in [15] associated event types (Read/Write) with blocks and proposed reorganization for directing read-heavy blocks to SSDs and write-heavy blocks to HDDs as discussed in the related work. This idea can be adapted to our framework such that correlations of read-heavy blocks and write-heavy blocks can be planned separately in two different correlation graphs. Then, read-heavy block reorganizations can be performed over SSDs while write-heavy block reorganizations can be performed over HDDs. This way, parallel I/O is boosted while considering individual device characteristics.

B. Replicated Datasets

Replication is a common technique used in distributed file systems to improve performance, reliability, and availability [30]. In replicated systems, replica selection plays an important role in I/O performance such that skewed disk accesses can be alleviated by selecting the replicas efficiently [31–34]. However, some existing systems use replication mainly for fault tolerance and utilize a predetermined (*primary*) replica for servicing I/O request, as in MongoDB [35]. In such cases,

reorganization can be performed among these replicas only. On some other systems, replicas are ordered by their distance from the reader and the closest replica to the reader is selected [30], in which case block reorganizations can be performed zone by zone, among the replicas of the correlated blocks that are physically close to each other. All these cases can be handled in our placement planning module by making slight modifications in the graph construction and coloring stages.

C. Reorganization Cost and Frequency

Although slightly increasing the support value reduces the reorganization cost considerably as shown in Tables II and III, reorganization rate limitation techniques can also be incorporated to keep layout optimization from overwhelming regular application I/O by limiting the number of active block movement operations both for the entire storage system and for each disk. Similar techniques are shown to be effective in large scale storage systems, such as the rate-limited re-replication technique implemented in the Google File System [30]. In addition, the effect of additional writes on the endurance of SSDs installed in AFAs can be considered as another cost of reorganization. Although endurance damage can be reduced considerably by using high support values, internally SSDs also perform optimizations for wear leveling and a commonly applied AFA design choice is to lean on the internal capabilities of SSDs and perform reorganization across SSDs through the storage array controller. Therefore, internal wear leveling algorithms used in SSDs are expected to distribute these additional writes evenly to their flash cells.

An important system decision to be made is how often to trigger reorganization. Although reorganization can be triggered in fixed intervals, especially in idle or low activity times, nonetheless such static reorganization would be against the nature of dynamic self-optimization. Instead, reorganization should be automatically triggered based on the disk I/O performance of the storage system, when the I/O performance drops below a predefined threshold. Disk I/O latency based service-level agreements (SLA) are commonly used in cloud computing since many real-time applications running in the cloud as well as enterprise data centers have strict response time requirements [1]. Based on SLA requirements or QoS guarantees provided, such thresholds can easily be determined and self-optimization can be triggered automatically if disk I/O performance drops below this threshold.

VII. Conclusion

In this paper, first we show evidence that block correlations exist in storage workloads, and then introduce a framework to detect the correlated blocks and reorganize them to improve I/O parallelism. Our analysis shows that the proposed self-optimization framework is highly successful in adjusting to skewed disk access patterns by keeping the I/O response time fairly stable. This property would be especially valuable to meet service-level agreements in cloud computing, where various applications with different disk access patterns can utilize the same storage system. The proposed framework is generic and can be applied to various parallel storage systems.

References

[1] N. Altiparmak *et al.*, "Replication based qos framework for flash arrays," in *CLUSTER '12*, Beijing, China, September 2012.

[2] D. A. Patterson *et al.*, "A case for redundant arrays of inexpensive disks (raid)," in *SIGMOD '88*, 1988, pp. 109–116.

[3] N. Altiparmak *et al.*, "Equivalent disk allocations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 3, 2012.

[4] M. J. Atallah *et al.*, "(Almost) optimal parallel block access for range queries," in *Proc. ACM PODS*, Dallas, Texas, May 2000, pp. 205–215.

[5] B. Atikoglu *et al.*, "Workload analysis of a large-scale key-value store," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 53–64, Jun. 2012.

[6] A. Miranda *et al.*, "Analyzing long-term access locality to find ways to improve distributed storage systems," in *PDP '12*, 2012, pp. 544–553.

[7] Z. Li *et al.*, "C-miner: Mining block correlations in storage systems," in *FAST '04*, Berkeley, CA, USA, 2004, pp. 173–186.

[8] D. Narayanan *et al.*, "Write off-loading: Practical power management for enterprise storage," *Trans. Storage*, vol. 4, no. 3, Nov. 2008.

[9] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *OSDI'12*, Berkeley, CA, USA, 2012, pp. 251–264.

[10] M. Bhadkamkar *et al.*, "Borg: Block-reorganization for self-optimizing storage systems," in *FAST '09*, Berkeley, CA, USA, 2009, pp. 183–196.

[11] W. W. Hsu *et al.*, "The automatic improvement of locality in storage systems," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, Nov. 2005.

[12] H. Huang *et al.*, "Fs2: Dynamic data replication in free disk space for improving disk performance and energy consumption," in *SOSP '05*. New York, NY, USA: ACM, 2005, pp. 263–276.

[13] G. Weikum *et al.*, "Dynamic file allocation in disk arrays," in *SIGMOD '91*. ACM, 1991, pp. 406–415.

[14] R. Arnan *et al.*, "Dynamic data reallocation in disk arrays," *Trans. Storage*, vol. 3, no. 1, Mar. 2007.

[15] T. Xie *et al.*, "Dynamic data reallocation in hybrid disk arrays," *IEEE Trans. on Parallel and Distributed Systems*, vol. 21, no. 9, Sept 2010.

[16] A. Miranda *et al.*, "Craid: Online raid upgrades using dynamic hot data reorganization," in *FAST '14*, Santa Clara, CA, 2014, pp. 133–146.

[17] J. Axboe, *blktrace User Guide*, Feb 2007, available: http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html.

[18] C. Borgelt, "Frequent item set mining," *WIREs Data Mining Knowl. Discov.*, vol. 2, p. 437456, Nov. 2012.

[19] T. Jensen *et al.*, *Graph coloring problems*. John Wiley & Sons, 2011.

[20] S. Fitzpatrick *et al.*, "An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs," in *Stochastic Algorithms: Foundations and Applications*, 2001, vol. 2264, pp. 49–64.

[21] A. Khanafer *et al.*, "The min-conflict packing problem," *Computers & Operations Research*, vol. 39, no. 9, pp. 2122 – 2132, 2012.

[22] L. R. Ford *et al.*, *Flows in Networks*. Princeton University Press, 1962.

[23] A. V. Goldberg *et al.*, "Minimum Cost Flows in Graphs with Unit Capacities," in *STACS '15*, 2015, pp. 406–419.

[24] J. S. Bucy *et al.*, "The disksim simulation environment version 4.0," Carnegie Mellon University Parallel Data Lab, Tech. Rep., May 2008.

[25] N. Agrawal *et al.*, "Design tradeoffs for ssd performance," in *ATC'08: Usenix Annual Technical Conference*, Berkeley, CA, USA, 2008.

[26] L. Breslau *et al.*, "Web caching and zipf-like distributions: evidence and implications," in *INFOCOM '99*, vol. 1, Mar 1999, pp. 126–134.

[27] Y. Zhang *et al.*, "Warming up storage-level caches with bonfire," in *"FAST '13"*, San Jose, California, "February" "2013".

[28] J. C. Chou *et al.*, "Exploiting replication for energy-aware scheduling in disk storage systems," *IEEE TPDS*, vol. 26, no. 10, 2015.

[29] *SNIA IOTTA Trace Repository*, Storage Networking Industry Association, http://iotta.snia.org.

[30] S. Ghemawat *et al.*, "The google file system," in *SOSP '03*, pp. 29–43.

[31] N. Altiparmak *et al.*, "Integrated maximum flow algorithm for optimal response time retrieval of replicated data," in *41st International Conference on Parallel Processing (ICPP 2012)*, Pittsburgh, Sep 2012.

[32] N. Altiparmak *et al.*, "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.

[33] N. Altiparmak *et al.*, "Continuous retrieval of replicated data from heterogeneous storage arrays," in *MASCOTS '14*, Paris, France, September 2014.

[34] N. Altiparmak *et al.*, "Multithreaded maximum flow based optimal replica selection algorithm for heterogeneous storage architectures," *IEEE Transactions on Computers*, vol. 65, no. 5, May 2016.

[35] K. Chodorow *et al.*, *MongoDB: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2010.