Towards Adaptive Parallel Storage Systems

Erica Tomes, *Student Member, IEEE*,
Everett Neil Rush, *Student Member, IEEE*,
and Nihat Altiparmak, *Member, IEEE*

Abstract—Disk I/O is a major bottleneck limiting the performance and scalability of data intensive applications. A common way to address disk I/O bottlenecks is using parallel storage systems and utilizing concurrent operation of independent storage components; however, achieving a consistently high parallel I/O performance is challenging due to static configurations. Modern parallel storage systems, especially in the cloud, enterprise data centers, and scientific clusters are commonly shared by various applications generating dynamic and coexisting data access patterns. Nonetheless, these systems generally utilize one-layout-fits-all data placement strategy frequently resulting in suboptimal I/O parallelism. Guided by association rule mining, graph coloring, bin packing, and network flow techniques, this paper proposes a general framework for adaptive parallel storage systems, with the goal of continuously providing a high-degree of I/O parallelism. Evaluation results indicate that the proposed framework is highly successful in adjusting to skewed parallel access patterns for both hard disk drive (HDD) based traditional storage arrays and solid-state drive (SSD) based all-flash arrays. In addition to the storage arrays, the proposed framework is sufficiently generic and can 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.

Index Terms—Storage systems, parallel I/O, self-optimization

1 Introduction

Today's most critical applications, including genome analysis, climate simulations, drug discovery, space observation & imaging, and numerical simulations in computational chemistry and high energy physics, are all data intensive in nature [1]. Although parallel processing techniques can address the computational requirements, developments in the storage subsystem do not keep pace with the computing power. Consequently, disk I/O bottlenecks can significantly limit the performance and scalability of data intensive applications [2]. Parallel storage systems have the potential to provide high-performance I/O through concurrent operation of individual storage components if a parallelism-aware data layout can be continuously guaranteed [3].

Storage arrays are well known examples of 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 considerable attention recently due to their random access nature and superior parallel I/O potential [3]. 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. Striping and declustering (grouping *dissimilar* objects together) 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 [4]. As well as HSAs, existing AFAs come with traditional RAID [5] techniques originally designed for HDDs to *statically* distribute data over disks [6]. In addition, several advanced declustering methods have also been proposed for the static placement of data [7], [8], [9]. For better parallelism, a common approach in declustering is to assume a certain disk access pattern. For instance, the technique proposed in [7] is optimized specifically for range queries where a range of values are searched in a multi-dimensional dataset as in relational databases, spatial databases, visualization, and GIS applications. However, the disk access patterns of many real-world applications change over time, and many realistic workloads are known to be skewed in practice [10], [11]. Also, various applications using the same storage system can have different access patterns. Therefore, a data layout optimized for a specific disk access pattern may not perform well in general. *In order to utilize parallel storage systems to their full potential, parallelism-aware adaptive 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. An earlier version of this article appeared at the *IEEE HiPC 2016* conference [12]. This version eliminates the usage of costly frequent itemset mining algorithms for finding block correlations, and introduces a multithreaded block correlator resulting in significantly faster storage workload analysis time.

2 Background and Related Work

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.

2.1 Data Placement and Parallel Access Patterns

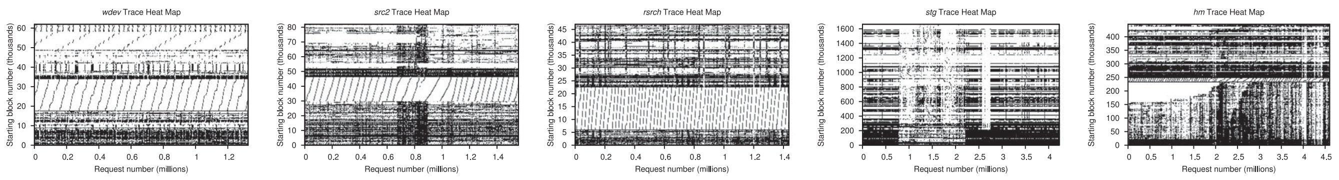
Efficient data layout is crucial in parallel disk architectures to enable concurrency and high performance parallel I/O. Considering the blocks of a disk I/O request, if each of the requested data blocks are stored in a different disk, then retrieval of this request would require 1 parallel access, where optimal number of parallel accesses is calculated as $\lceil \frac{b}{N} \rceil$ for b blocks and N disks. Data layout is commonly optimized by assuming static disk access pattern. For instance, *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. In order to boost the concurrency of parallel disks, an adaptive data layout optimization 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.

2.2 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 [13]. 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

• *The authors are with the Department of Computer Engineering and Computer Science, University of Louisville, Louisville, KY 40292.
E-mail: {erica.tomes, e.rush, nihat.altiparmak}@louisville.edu.*

*Manuscript received 18 Sept. 2017; revised 6 May 2018; accepted 8 May 2018. Date of publication 15 May 2018; date of current version 7 Nov. 2018.
(Corresponding author: Nihat Altiparmak.)
Recommended for acceptance by F. Douglis.
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.2018.2836426*



(a) Web Server (*wdev*) (b) Version Control (*src2*) (c) Research Proj.(*rsrch*) (d) Staging Server (*stg*) (e) HW Monitor (*hm*)

Fig. 1. Storage heat maps of enterprise servers.

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.

Block correlations are commonly encountered in storage workloads. We observed this while analyzing the storage traces of various enterprise servers from Microsoft [14]. To illustrate, Fig. 1 shows the storage heat maps of a web server, a version control server, a research project server, a staging server, and a hardware monitoring 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.

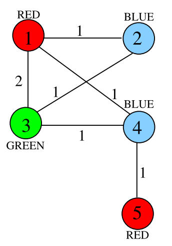
2.3 Related Work

Although static placement received more attention [3], [5], [7], [8], [9], dynamic data layout optimization techniques targeting single- and multi-disk HDDs were also proposed. Among the ones targeting single disks, block correlations were utilized in [13] for prefetching purposes and arranging correlated blocks contiguously in a single HDD for reduced seek time. A similar correlation-based prefetching strategy is also proposed for scientific applications utilizing a distributed file system, where Markov models were used to predict blocks to be served from a client cache [15]. In addition, seek-aware techniques were applied in [16] to reorganize hot data blocks sequentially on a dedicated partition of a single HDD. However, these techniques either target single HDDs for reduced seek time or aim to improve caching performance, without focusing on I/O parallelism.

Dynamic data layout optimization for HDD-based parallel disks was first investigated in [17], 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 a dynamic system. Authors in [18] 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. Most recently, authors in [19] extended the hot block

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

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



(a) I/O Monitoring (b) FIM Analysis (c) Correlation Graph

Fig. 2. I/O monitoring, analysis, and layout planning steps.

prefetching idea proposed in [13] for single disks to parallel disks and applied prefetching, where additional copies of the frequently accessed blocks are created and cached.

Existing techniques either make an assumption about data placement and I/O patterns, or ignore I/O parallelism. Especially with the popularity of *near* random access non-volatile memory (NVM) devices, I/O parallelism deserves more attention to benefit from the rich internal and external parallelism opportunity. Our main concern is proposing a general framework that can provide consistently high parallel I/O performance on both HDD- and SSD-based parallel storage systems, without damaging the sequential data access of individual HDDs and without creating additional copies of data.

3 Dynamic Data Layout Optimization Framework

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

- *Storage Workload Monitoring:* Monitors I/O requests and records block IDs that are requested together in *transactions*.
- *Storage Workload Analysis:* Analyzes the recorded transactions to detect block correlations.
- *Adaptive Data Layout Planning:* Plans a new layout adaptively based on the detected block correlations.
- *Efficient Data Migration:* Performs the planned data layout optimization by minimizing the data migration cost.

The rest of this section describes each component of the proposed framework in more detail.

3.1 Storage Workload Monitoring Module

Block level requests can be monitored using a disk I/O tracing tool such as `blktrace` [20], which can provide detailed information about each disk request including its timestamp, 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 size of the request in blocks. Using this information, the monitoring module divides the requests into transactions, where each transaction represents a set of blocks that are requested (or handled) together by the storage subsystem, and stores these transactions line by line in an output file.

A simple transaction is composed of the blocks of a single request (intra-request correlation); however, more patterns can be detected by combining requests with low inter-arrival times in a single transaction (inter-request correlation). One way to achieve this is by defining a maximum time threshold that a transaction can span based on the capabilities of the storage system in use. A sample I/O monitoring output composed of three transactions is provided in Fig. 2a, indicating that blocks 1, 2, and 3 are requested together in the first transaction, blocks 1, 3, and 4 are requested together in the second transaction, and blocks 4 and 5 are requested together in the third transaction.

Typical size of an I/O monitoring output file would change depending on the duration of the monitoring as well as the I/O intensity of the system. Our analysis on Microsoft's enterprise servers [14] shown in Fig. 1 indicated that typical sizes of raw monitoring files change between 183 MB (*wdev*) and 1.95 GB (*stg*) for a week-long monitoring period, which includes an average of 3 million requests. Since transactions generally include repeating block IDs, raw monitoring files can efficiently be compressed when they reach a threshold limit, and stored in a compressed format until they need to be analyzed. Once their analysis is completed, monitoring files can immediately be removed. Although temporary storage of monitoring files can be performed on the storage device to be optimized, their storage on a dedicated device would eliminate additional I/O and monitoring activity.

3.2 Storage Workload Analysis Module

The storage workload analysis module is responsible for analyzing the transactions 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 [21].

3.2.1 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, where we propose to place the correlated blocks into separate parallel storage components to boost concurrency and parallel I/O performance. Mining the transactions provided in Fig. 2a using the FIM techniques for $set-\ size = 2$ returns the output shown in Fig. 2b. In each row of the FIM output in Fig. 2b, 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. Using this FIM output, we can construct an undirected *correlation graph* $G(V, E)$ as shown in Fig. 2c such that each vertex $v \in V$ represents a block, each edge $(u, v) \in E$ represents a correlation between blocks u and v, and edge weights represent the correlation frequencies.

3.2.2 Multithreaded Block Correlator (MBC)

Frequent itemset mining algorithms and their data structures are designed for the generic case of searching all possible item combinations of all itemset sizes. However, finding correlated block pairs ($set\text{-}size = 2$) is sufficient for generating our target correlation graph. In this section, we propose a high-performance multithreaded block correlator, customized for our purposes, which takes the I/O monitoring result as an input (Fig. 2a) and generates the correlation graph (Fig. 2c) as an output. For a given set of m blocks (items) in a transaction, finding all possible block pairs is basically 2-combination of m, requiring $\binom{m}{2} = \frac{m \cdot (m-1)}{2}$ iterations. Once the block pairs are determined for each transaction, the correlation graph can be updated in constant time on average using adjacency lists supported by hash tables. Although even the sequential implementation of this block correlation technique would provide a significant performance improvement over the FIM approach, it is possible to improve the performance of block correlation even further by using parallelization techniques.

Our multithreaded block correlator is provided in Algorithm 1, which assigns a separate transaction to every thread and threads perform pair enumeration as well as graph insertion operations in parallel. Transactions (tasks) can be assigned to threads in two different ways: (i) Dividing the file into equal size chunks (in bytes) and assigning each thread a separate chunk, (ii) Reading the file sequentially and assigning transactions to the threads one by one. We tested both approaches and found that the second approach yields better performance in shared memory settings due to improved file I/O, caching, as well as better load balancing among the threads. There are two implementation options for the sequential task assignment: (i) Using a thread pool approach and dedicating a producer thread for performing file I/O as well as inserting the tasks into the work queue, (ii) Using a thread safe I/O library and allowing threads to perform file read in parallel using a single file handle and implicit/explicit file locking mechanisms. In Algorithm 1, we use the second implementation choice and assume an implicit file locking; however, the first approach is expected to provide a similar performance since the synchronization needs to be enforced either during the file I/O operation or during the task insertion/subtraction operations.

Algorithm 1. Multithreaded Block Correlator (MBC)

```
Input: I/O monitoring file $f$ in market basket format
Output: Correlation graph $G(V, E)$
 1: for all $transaction$ in $f$ in parallel do
 2:   blocks(1, ..., $m$) = array of block IDs (items) in $transaction$
 3:   for $i \leftarrow 1 : m - 1$ do
 4:     $v_i$ ←blocks[$i$]
 5:     $adj\_list\_v_i \leftarrow$ GetAdjList($G, v_i$)
 6:     AcquireLock ($adj\_list\_v_i$.lock)
 7:     for $j \leftarrow i + 1 : m$ do
 8:       $v_j$ ←blocks[$j$]
 9:       UpdateGraph ($G, adj\_list\_v_i, v_j$)
10:     ReleaseLock ($adj\_list\_v_i$.lock)
11:   for $j \leftarrow m : 2$ do
12:     $v_j$ ←blocks[$j$]
13:     $adj\_list\_v_j \leftarrow$ GetAdjList($G, v_j$)
14:     AcquireLock ($adj\_list\_v_j$.lock)
15:     for $i \leftarrow j - 1 : 1$ do
16:       $v_i$ ←blocks[$i$]
17:       UpdateGraph ($G, adj\_list\_v_j, v_i$)
18:     ReleaseLock ($adj\_list\_v_j$.lock)
```

At line 1 of Algorithm 1, every thread reads the transactions from the I/O monitoring file using a thread-safe I/O library guaranteeing that each thread receives a unique transaction. Next, lines 2-18 are executed by all threads in parallel. At line 2, the transaction items (block IDs) are tokenized and stored in *blocks* array. Nested loops at line 3 and 7 perform the block pair enumeration, and the correlation graph is updated at line 9. In order to protect the shared correlation graph structure, we can either utilize regular hash tables with locks or use concurrent hash tables [22] for both mapping vertices to adjacency lists (outer level) and mapping adjacency list members to the corresponding edge weights (inner level). In Algorithm 1, we use a combination of these two approaches for the best performance. We found that the concurrent hash table implementation [22] provides us the best performance at the outer level as utilized at line 5; however, locking with regular hash tables achieves a better performance at the inner level as performed at lines 6, 9, and 10. The reason behind better performance of regular hash tables at the inner level is because there is less lock contention at that level. However, at the outer level there is increased lock contention; therefore, using a concurrent hash table provides better performance.

Using an outer level hash table eliminates the naive coarse-grained locking approach, which locks the entire graph structure and prevents concurrent accesses, even if the concurrent thread operations are on non-conflicting regions of the graph. This coarse-grained locking approach generally causes an increase in lock contention. Our finer-grained locking approach, at the inner adjacency list level, alleviates the lock contention problem by allowing a high degree of concurrent graph updates. In addition, once the lock is achieved at line 6 for the adjacency list of vertex v_i, all the edges from v_i to all possible v_j enumerations can be inserted at lines 7-9 with only one acquire-lock operation, eliminating frequent locking/unlocking overhead. Since our correlation graph is undirected, the adjacency lists for the opposite end vertices are updated at lines 11-18 with the same motivation of eliminating frequent locking/unlocking issue.

Even though our scope in this paper is limited to the shared memory parallelization of the proposed custom block correlation approach, it would also be possible to parallelize this algorithm in distributed memory architectures using the MapReduce paradigm. In distributed settings, the I/O monitoring file is expected to be stored in a distributed file system, where the mappers can read the file chunks, transaction by transaction, and perform block pair enumeration for each transaction by emitting the MapReduce key-value pairs of `<key=`*block-pair*`,value=`*1*`>`. On the other hand, reducers can aggregate these key-value pairs and find the final frequency values of the block pairs as in the famous `wordcount` example of MapReduce. At the end, the correlation graph can be generated using the final output file of MapReduce.

3.3 Adaptive Data Layout Planning Module

The aim of the adaptive data layout planning module is to use block correlations produced by the storage workload analysis module and to plan a new correlation-based data layout that boosts concurrency and I/O parallelism. However, optimal placement is a challenging problem, even in simplified cases.

3.3.1 Basic Layout Planning

In this section, we formulate the correlation-based basic layout planning as an optimization problem as follows:

Problem Definition 1 (Basic Layout Optimization Problem (BLOP)): *Given N disks and a correlation graph $G(V,E)$ such that each vertex $v \in V$ represents a data block and each edge $(u,v) \in E$ represents a correlation between blocks u and v, plan a data layout so that for every correlated block pair $(u,v) \in E$, blocks u and v are stored in different disks.*

Theorem 1. *BLOP is NP-complete.*

Proof. Given a *correlation graph* $G(V,E)$, the *proper (vertex) k-coloring* [23] colors the vertices of G with a maximum of k colors such that adjacent vertices receive different colors as shown in Fig. 2c. BLOP is equivalent to the proper k-coloring problem 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 basic layout 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 instead of proposing an entirely new and unproven heuristic.

Although BLOP outlines the main purpose of our layout planning strategy, it is simplified and requires additional considerations to be applied in real world settings. First, 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*. 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 layout. 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.

3.3.2 Min-Conflict Bin Packing

Including the aforementioned real world considerations, our layout 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 [24] 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 layout 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 layout. 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 are given more importance.

P4 If there is 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 layout planning heuristic is provided in Algorithm 2. Each vertex $v \in \{1,,|V|\}$ represents a block and each color $c \in \{1,,N\}$ represents a disk, where N is the total number of disks in the system. 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 at 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 $\epsilon = 5\%$ 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 at line 19 to bound the number of iterations by a constant. The proposed layout planning heuristic has the worst-case time complexity of $O(|V| \log |V| + |E|)$.

Algorithm 2. Layout Planning Heuristic

```
Input: Uncolored correlation graph $G(V, E)$
Output: Colored correlation graph $G(V, E)$
 1: Initially color the vertices based on the original data layout
 2: caps (1, ..., $N$) = array of initial disk capacities in bytes
 3: for $v \in V$ do
 4:    $v$.tcf = 0
 5:    for $u \in v$.adj do
 6:       $v$.tcf += $(u, v)$.weight
 7: S (1, ..., $|V|$) = vertices sorted (descending) by $tcf$
 8: repeat
 9:    for $i \leftarrow 1 : |V|$ do
10:       $v \leftarrow$S[$i$]
11:       confs (1, ..., $N$) = 0
12:       for $u \in v$.adj do
13:          confs[$u$.clr] += $(u, v)$.weight
14:       for $c \leftarrow 1 : N$ 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 $\Delta conficts < \epsilon$
```

3.4 Efficient Data Migration

Property P1 of the proposed layout planning heuristic aims to eliminate unnecessary data movements during reorganization by initially mapping colors to the disks based on the original block locations. In order to test the efficiency of this technique, in this section we provide an efficient optimal algorithm for color to disk mapping guaranteeing the minimum data movement. Therefore, by using the colored graph returned by the adaptive data layout planning module, the data migration module maps each color to a separate disk, minimizing the number of block movements. A brute force solution would 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, and choose the mapping yielding the minimum amount of block movements. However, it would require unacceptably high execution time for large N due to its factorial time complexity. A polynomial time solution can be achieved by constructing the problem as a flow network and solving it using the minimum cost flow techniques [25], where an edge is drawn from source to every *color* vertex, from every *color* vertex to every *disk* vertex, and from every *disk* vertex to sink. Edge capacities are set to 1 and edge costs are set to 0 except the edges between a *color* vertex and a *disk* vertex, which is set to the amount of the data movement caused by such mapping. Running a minimum cost flow algorithm on the constructed graph will return the optimum color-to-disk mapping, yielding the minimum data movement. Readers are directed to the earlier version of this paper [12] for an in-depth description of this Section.

3.5 Additional Optimizations for HDD-Based Systems

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 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. Considering the sequentiality of correlated blocks in individual disks and 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 Fig. 2c, the layout planning module needs to group 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 this 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 as larger group sizes work against concurrency.

4 Evaluation

In this section, we share our performance evaluation and cost analysis using simulations driven by real-world application workloads.

4.1 Experimental Setup

Our simulations were run in DiskSim 4.0 using the SSD patch from Microsoft Research. We ran experiments on both HDD-based storage arrays (HSA) and SSD-based all-flash arrays. In both cases, we used the maximum number of disks supported by the simulator, 100 for HSAs and 14 for 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. DiskSim is configured in a way that it allows more than one request to be outstanding in the storage subsystem, and queuing time is included in the reported response time. Each experiment is repeated 5x with random block distributions and the results were averaged.

4.1.1 Frequent Itemset Miner

As a Frequent Itemset Mining (FIM) algorithm, we used the Borgelt's Eclat miner [21]. In order to mine for only correlated block pairs, we set both the minimum number of items parameter ($-m$) and the maximum number of items parameters ($-n$) to 2.

4.1.2 Multithreaded Block Correlator (MBC)

We implemented our multithreaded block correlator in C++ using the POSIX threads (pthreads) library, and compiled it using g++ version 5.4.0, optimization level 2 (-O2). As the concurrent hash table for mapping vertices to their corresponding adjacency lists, we used the implementation discussed in [22]. As a regular hash table for the adjacency lists themselves, we used `std::unordered_map` as part of C++11 and used pthreads' mutex locks to achieve mutually exclusive access. The machine we used for evaluation has two 14-core Intel Xeon E5-2680 V4 processors and each core is running at 2.4 GHz clock rate. The system has 128 GB of memory and it runs Ubuntu 16.04.1 LTS operating system with kernel version 4.4.0-79-generic. For comparison purposes, we have also implemented the

TABLE 1
Workload Statistics (for First 100k Requests, First 50k Training, Remaining 50k Testing)

Workload	Application	R/W (%)		Avg. R/W Size (KB)		I/O Size (MB)		Unique Data Size (MB)		Re-access (%)	InterArrival (%) $< 100\ \mu s$	
		Training	Testing	Training	Testing	Training	Testing	Training	Testing	Testing vs. Training	Training	Testing
wdev	Test Web Server	20.5 / 79.5	14.6 / 85.4	14.6 / 7.7	8.8 / 8.5	443.7	415.4	137.6	155.2	81.9	78.7	79.4
src2	Version Control	17.1 / 82.9	4.9 / 95.1	6.2 / 6.8	5.3 / 6.2	328.0	301.8	115.3	77.5	92.5	78.0	76.4
rsrch	Research Projects	18.3 / 81.7	0.6 / 99.4	7.4 / 9.4	10.5 / 8.2	440.0	401.5	139.9	93.4	88.0	81.4	78.1
stg	Staging Server	12.6 / 87.4	11.5 / 88.5	9.3 / 8.6	9.9 / 8.3	425.4	414.9	122.7	161.7	83.7	80.2	82.9
hm	Hardware Monitor	48.4 / 51.6	41.9 / 58.1	8.8 / 6.5	9.2 / 7.4	371.4	399.2	188.6	168.0	73.8	63.8	72.6
AVERAGE RESULTS:		23.4 / 76.6	14.7 / 85.3	9.3 / 7.8	8.7 / 7.7	401.7	386.5	140.8	131.2	84.0	76.4	77.9

sequential version of MBC, which we refer to as Sequential Block Correlator (SBC).

4.1.3 Initial Data Layout

Our workloads are from various HDD-based enterprise servers with a 512 B block size, where each server is configured with one or more RAID-5 arrays as data volumes. However, the only information provided in the workload traces related to the block locations is the corresponding volume numbers, which ranges between 1 to 4. Therefore, we need to assume an initial data layout for mapping block IDs to 100 disk HSAs and 14 disk AFAs. Many natural data access patterns of various applications, including 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, where the relative probability of a request for the ith most popular item is proportional to $1/i^{\alpha}$, $0 \leq \alpha \leq 1$ [10], [26]. For $\alpha = 0$, requests are evenly distributed, and for $\alpha = 1$ (Zipf's law), access distribution is skewed towards the most popular item. In other words, some data is accessed more frequently than the rest, resulting in hot spots of data locality. Based on a previous work performed on web traces, a realistic α value is found to lie between 0.64 and 0.83 [26].

In order to test our proposed framework under various skew in parallel access patterns, we performed our initial data placement by following a set of Zipf-like distributions so that the relative probability of accessing a block on the i-th most popular disk is proportional to $1/i^{\alpha}$ for $0 \leq \alpha \leq 1$. Zipf-like distributions are commonly used in storage system research to perform initial data placement [27]. Deng et. al [28] indicates that the skew of disk I/O access is often referred to as the 80/20 rule of thumb (corresponding $\alpha = 0.85$ in Zipf), or in more extreme cases, the 90/10 rule. The 80/20 rule indicates that twenty percent of storage resources receive eighty percent of I/O accesses, while the other eighty percent of resources serve the remaining twenty percent of I/O accesses. Following a similar pattern, we tested 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.

4.1.4 Evaluation Methodology and Workloads

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 workload into a training set and a testing set. The training set is the first 50 percent of the requests, and the testing set is the remaining 50 percent. After splitting the workloads, only the training set is used by our framework for detecting the patterns and determining the layout optimization plan, and only the testing set is used to evaluate the performance of our 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.*

We evaluate our framework using five publicly available real-world storage workloads provided by Microsoft [14]. These workloads include block level I/O requests of various enterprise and production servers running in Microsoft, including a web server(*wdev*), a version control server (*src2*), a research projects server (*rsrch*), a staging server (*stg*), and a hardware monitoring server (*hm*). Relevant workload statistics for utilized portions of the workloads are provided in Table 1 for both training and testing sets, including R/W percent, average R/W size, total I/O performed (R+W), total amount of unique data accessed, re-access percent (percent of testing data previously observed in training), and the percentage of requests that have interarrival time of less than 100 μs.

4.2 Experimental Results

We first briefly share the results of I/O performance evaluation for AFAs and HSAs, including the effect of additional HDD optimizations described in Section 3.5 and the cost analysis in terms of the data movement. Extended version of performance evaluation results and cost analysis can be found in the earlier version of this paper [12] including the accompanying performance graphs. In this version, we mainly focus on the performance evaluation of the proposed Multithreaded Block Correlator compared with its sequential counterpart as well as the previously proposed FIM approach.

4.2.1 SSD-Based All-Flash Arrays (AFA)

Our evaluation indicated that disk I/O latency of the static layout 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. Considering all workloads and all α values of the Zipf-like distribution, our framework achieves 111 percent Read, 52 percent Write, and 53 percent overall (R+W) performance improvement over the static layout on average.

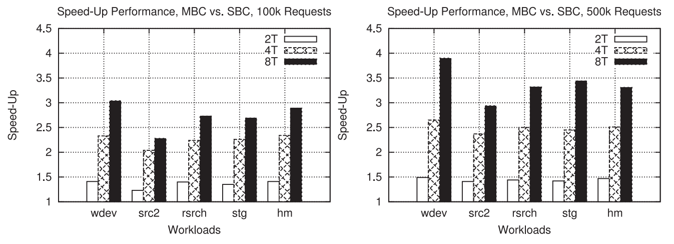
4.2.2 HDD-Based Storage Arrays (HSA)

The results for HSAs are similar to AFAs; except, the damage of not performing dynamic layout optimization is more severe since HDDs are substantially slower compared to SSDs, especially in their read performance. In HSAs, response time for the static layout 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. Considering all workloads and all α values, our framework including the additional HDD optimizations achieves 366 percent Read, 82 percent Write, and 170 percent overall (R+W) performance improvement over the static layout on average.

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

TABLE 2
Cost versus *Overall(R+W)* Performance - AFA

Workload	Threshold = 1		Threshold = 5		Threshold = 10	
	Cost(*MB*)	Per.(%)	Cost(*MB*)	Per.(%)	Cost(*MB*)	Per.(%)
wdev	82.09	16.97	9.39	9.40	1.23	6.28
src2	63.03	97.14	3.37	66.23	1.46	60.07
rsrch	76.19	89.26	4.09	41.90	0.76	32.89
stg	83.40	31.97	5.94	13.57	2.22	10.64
hm	102.63	30.46	5.70	9.86	0.95	5.66
AVG	81.46	53.15	5.69	28.19	1.32	23.10



(a) 100k Requests (b) 500k Requests

Fig. 3. Speed-up performance of MBC.

individual disks together as described in Section 3.5, we achieve an additional 216 percent Read, 24 percent Write, and 86 percent overall (R+W) performance improvement over the static layout averaged over all workloads and α values. Based on the specifications of the storage system that we used in our experiments, we set the maximum group size to 512 KB (1024 blocks). This value is calculated using the difference between the average-case and the worst-case positioning time of the HDD that we used, and the amount of data that it can transfer sequentially within this time.

4.2.3 Migration Cost Analysis

In this section, we analyze the cost of the proposed framework in terms of data migration and demonstrate how this cost can be controlled easily by using the correlation frequency values. Correlation frequency of a block pair stored in the edge weight of the correlation graph indicates the strength of the correlation. In other words, it indicates the number of times this correlation occurred in the history. Therefore, correlations with high frequency have high chance to be repeated in the future. By ignoring the correlations with less than a certain threshold, it is possible to reduce the cost of data migration. On the other hand, reducing the number of correlations considered by the framework is also expected to cause a reduced I/O performance as a side effect.

In Table 2, we illustrate the trade-off between the migration cost (data moved in MB) and the I/O performance improvement (in percent) using various threshold correlation frequency values for AFAs. The provided values are averages for all α. For the previously discussed experimental results using $threshold = 1$, our framework moves 81.5 MB of data on average, and gains an average of 53.2% performance improvement. Such an improvement was possible due to frequent re-access of the same data and low inter-arrival time between requests. Table 1 shows that on average, the testing workloads request 386.5 MB of data (I/O Size), where only 131.2 MB of this is unique. Moreover, on average 84% of data that is requested in testing is previously observed in training, clearly outlining the frequent re-access behavior. In addition, 77.9 percent of the testing requests have inter-arrival times of less than 100 μs creating a bursty I/O pattern and convoy effect.

Since the reorganization is performed on the training data, 100 percent of data that can possibly be reorganized for each workload is shown in Table 1 as "Unique Data Size - Training". For the correlation frequency threshold of 1, our framework reorganized 54.5 percent (*hm*) to 68.0 percent (*stg*) of this data (58.2 percent on average). Nevertheless, data movement amount and percentage can easily be decreased by increasing the threshold correlation frequency value and slightly losing from the performance. In Table 2, increasing the threshold from 1 to 5 reduces the data movement cost an average of 15.6x with 2.2x loss in performance (corresponds to 28.19 percent performance improvement) by reorganizing only 4.1 percent of total data. Moreover, increasing the threshold from 1 to 10 reduces the cost an average of 71.2x while decreasing the performance improvement 3.1x (corresponds to 23.10 percent performance improvement) by reorganizing only 0.97 percent of total data.

We observed a similar trade-off for HSAs; where increasing the threshold from 1 to 10 reduced the cost an average of 74x while causing only 9.8x performance loss (corresponds to 23.59 percent performance improvement) by reorganizing only 1.15 percent of total data. Extended migration cost versus performance analysis for HSAs is provided in the earlier version of this paper [12]. An alternative technique would be determining an upper limit for the amount of data to be reorganized, and performing reorganization up to this amount of blocks in decreasing total correlation frequency order.

Finally, we have also analyzed the effect of Property P1 presented as part of the proposed layout planning heuristic, which aims to minimize the amount of data movement by initially mapping colors to disks based on the original data layout, with the aim of keeping blocks in their original locations as much as possible. For the workloads that we have investigated and the block correlations existing within them, our analysis revealed that due to P1, the original mapping stays within 1-2 percent of the optimal mapping provided by the min-cost flow algorithm. Nevertheless, since this behavior may change depending on the properties of the workload in use, it is still valuable to check the the optimal mapping before performing the data movement.

4.2.4 Performance of Multithreaded Block Correlator (MBC)

In this section, we evaluate the execution time performance of the Multithreaded Block Correlator (MBC) proposed in Section 3.2.2, compared with the execution time of the Frequent Itemset Mining (FIM) algorithm, as well as the Sequential Block Correlator. In order to fairly calculate the execution times of these algorithms, we repeated every run 5x and averaged the results by clearing the page cache, directory entries, and the inodes from the memory before every run. Fig. 3a and 3b show the speed-up performance of MBC compared to SBC for 100k and 500k requests, respectively. The x-axis shows different workloads, and the y-axis shows the speed-up achieved over the sequential implementation ($\frac{SBC}{MBC}$) for 2 threads (2T), 4 threads (4T), and 8 threads (8T). For 100k requests, averaged over all workloads, MBC achieved 1.36x speed-up using 2 threads, 2.24x speed-up using 4 threads, and 2.73x speed-up using 8 threads. For 500k requests, speed-up values slightly increased to 1.45x, 2.5x, and 3.38x for 2, 4, and 8 threads, respectively. For both request sizes, we observed a stable speed-up performance after 8 threads.

We discovered that the speed-up behavior of MBC is directly related to the transaction size of the workloads. As shown in Table 1, average request size of our training workloads is around 8 KB on average, corresponding to an average of 16 blocks (items) per transaction since the block size is 512 B. In addition, *wdev* has the largest average transaction size and *src2* has the smallest average transaction size. Similarly, we observed the largest speed-up with *wdev* and the smallest speed-up with *src2*.

We also share the execution time of MBC and SBC in seconds, compared with the execution time of the FIM approach in Table 3. In this table, we are only showing the MBC for 8 threads since it provides the best performance. It is clear from the table that FIM requires considerably longer time to perform the storage workload

TABLE 3
Execution Time of FIM, SBC, and MBC

Workload	Execution Time (in seconds)	
	100k Requests	500k Requests
***wdev*- FIM**	210.53	1294.37
***wdev*- SBC**	14.30	54.67
***wdev*- MBC (8T)**	4.71	14.02
***src2*- FIM**	82.30	1817.49
***src2*- SBC**	3.15	16.06
***src2*- MBC (8T)**	1.38	5.45
***rsrch*- FIM**	175.61	1172.13
***rsrch*- SBC**	8.05	25.63
***rsrch*- MBC (8T)**	2.95	7.72
***stg*- FIM**	252.12	1343.67
***stg*- SBC**	9.00	29.80
***stg*- MBC (8T)**	3.35	8.67
***hm*- FIM**	245.14	2642.50
***hm*- SBC**	14.19	41.68
***hm*- MBC (8T)**	4.89	12.58

analysis compared to SBC and MBC. Precisely, FIM is 14.7x to 28x slower than SCB (21.6x on average), and 44.7x to 75.3x slower than MBC (57.9x on average), MBC providing up to 248.77 seconds faster storage workload analysis time than FIM for 100k. The gap between FIM and our custom block correlators increases further for larger workload sizes. For 500k requests, FIM is up to 113.2x slower than SCB (58.2x on average), and up to 333.5x slower than MBC (188.5x on average), MBC providing up to 2629.92 seconds faster storage workload analysis time than FIM for 500k requests. These results clearly outline the superiority of the multithreaded block correlator over the FIM approach.

5 Discussion

In addition to the storage arrays, the proposed framework can also be applied to various parallel storage scenarios, including key-value stores, parallel/distributed file systems, and NVMs with various levels of internal parallelism. Readers are directed to the earlier version of this paper [12] for our insights on the further applicability, and our extended discussion on the system-specific tailoring for storage heterogeneity and replication.

Although slightly increasing the correlation frequency threshold reduces the data migration cost considerably as shown in Table 2, 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.

An important system implementation includes the automatic triggering of the proposed framework. Although optimization can be triggered in fixed intervals, especially idle or low activity times, such static reorganization would be against the nature of adaptive parallel storage systems. 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) or QoS specifications are commonly used in cloud computing and enterprise data centers [3], which can be used to determine such thresholds.

Although the proposed system cannot help a workload with read-once or write-once behavior, since we propose persistent and periodic data layout optimization, the proposed system does not have the assumption or requirement for the data to be accessed in the *last* monitoring period. Since there will be multiple and continuous monitoring periods, certain data might not have been touched in the last monitoring period, but could still have been optimized based on previous monitoring periods. Finally, the proposed system currently does not consider moving quiescent data, which can also be reorganized over the disks to balance *available* disk space utilization, or could be completely archived (if allowed) in another storage system to alleviate possible capacity-based limitation in layout planning. Such optimization can easily be performed in file-level, based on the last access time of the files.

6 Conclusion and Future Work

In this paper, first we show evidence that block correlations exist in storage workloads, then introduce a framework to detect the correlated blocks efficiently using a multithreaded block correlator and reorganize the detected blocks to improve I/O parallelism. The proposed framework is generic and can be applied to various parallel storage systems to achieve dynamic adaptability. Our future work includes adapting the proposed framework to allow online continuous detection of block correlations using stream data mining techniques [29]. This will allow us to eliminate intermediate trace storage and offline trace analysis so that a faster reaction to I/O bottlenecks can be achieved.

Acknowledgments

This research was supported by the U.S. National Science Foundation (NSF) under grant CNS-1657296.

References

[1] A. Wright, “Big data meets big science,” *Commun. ACM*, vol. 57, no. 7, pp. 13–15, Jul. 2014.
[2] D. Zhao, N. Liu, D. Kimpe, et al., “Towards exploring data-intensive scientific applications at extreme scales through systems and simulations,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, Jun. 2016.
[3] N. Altiparmak and A. S. Tosun, “Replication based QoS framework for flash arrays,” in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2012, pp. 182–190.
[4] K. Salem and H. Garcia-Molina, “Disk striping,” in *Proc. IEEE 2nd Int. Conf. Data Eng.*, Feb. 1986, pp. 336–342.
[5] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (RAID),” in *Proc. 1988 ACM SIGMOD Int. Conf. Manage. Data*, 1988, pp. 109–116.
[6] M. Jung and M. Kandemir, “An evaluation of different page allocation strategies on high-speed SSDs,” in *Proc. 4th USENIX Conf. Hot Top. Storage File Syst.*, 2012, p. 9.
[7] M. J. Atallah and S. Prabhakar, “(Almost) optimal parallel block access for range queries,” in *Proc. 19th ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst.*, May 2000, pp. 205–215.
[8] A. S. Tosun,“Threshold-based declustering,” *Inf. Sci.*, vol. 177, no. 5, pp. 1309–1331, 2007.
[9] N. Altiparmak and A. S. Tosun, “Equivalent disk allocations,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 3, Mar. 2012.
[10] B. Atikoglu, Y. Xu, E. Frachtenberg, et al., “Workload analysis of a large-scale key-value store,” *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 53–64, 2012.
[11] A. Miranda and T. Cortes, “Analyzing long-term access locality to find ways to improve distributed storage systems,” in *Proc. 20th Eur. Int. Conf. Parallel, Distrib. Net.-Based Process.*, Feb. 2012, pp. 544–553.
[12] E. N. Rush, B. Harris, N. Altiparmak et al., “Dynamic data layout optimization for high performance parallel I/O,” presented at the *23rd IEEE Int. Conf. High Perform. Comput Data Analytics*, Hyderabad, India, Dec. 2016.
[13] Z. Li, Z. Chen, S. M. Srinivasan, et al., “C-miner: Mining block correlations in storage systems,” in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 173–186.
[14] D. Narayanan, A. Donnelly, and A. Rowstron, “Write off-loading: Practical power management for enterprise storage,” *ACM Trans. Storage*, vol. 4, no. 3, Nov. 2008, Art. no. 10.
[15] J. Oly and D. A. Reed, “Markov model prediction of i/o requests for scientific applications,” in *Proc. 16th Int. Conf. Supercomputing*, 2002, pp. 147–155.
[16] M. Bhadkamkar, J. Guerra, L. Useche, et al., “BORG: Block-reorganization for self-optimizing storage systems,” in *Proc. Proc. 7th Conf. File Storage Technol.*, 183–196, 2009.
[17] G. Weikum, P. Zabback, and P. Scheuermann, “Dynamic file allocation in disk arrays,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1991, pp. 406–415.
[18] R. Arnan, E. Bachmat, T. K. Lam, et al., “Dynamic data reallocation in disk arrays,” *ACM Trans. Storage*, vol. 3, no. 1, Mar. 2007.
[19] A. Miranda and T. Cortes, “Craid: Online raid upgrades using dynamic hot data reorganization,” in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 133–146.

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

[21] C. Borgelt, "Frequent item set mining," *WIREs Data Mining Knowl. Discovery*, vol. 2, pp. 437–456, Nov. 2012.

[22] X. Li, D. G. Andersen, M. Kaminsky, et al., "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proc. 9th Eur. Conf. Comput. Syst.*, pp. 27:1–27:14.

[23] T. Jensen and B. Toft, *Graph Coloring Problems*. Hoboken, NJ, USA: Wiley, 2011.

[24] A. Khanafer, F. Clautiaux, S. Hanafi, et al., "The min-conflict packing problem," *Comput. Operations Res.*, vol. 39, no. 9, 2012.

[25] L. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ, USA: Princeton Univ. Press, 1962.

[26] L. Breslau, P. Cao, L. Fan, et al., "Web caching and zipf-like distributions: evidence and implications," in *Proc. IEEE 18th Annu.Joint Conf. IEEE Comput. Commun. Soc.*, Mar 1999, vol. 1, pp. 126–134.

[27] J. C. Chou, T. Lai, J. Kim, et al., "Exploiting replication for energy-aware scheduling in disk storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 10, pp. 2734–2749, 2015.

[28] Y. Deng, L. Lu, Q. Zou, et al., "Modeling the aging process of flash storage by leveraging semantic I/O," *Future Gener. Comput. Syst.*, vol. 32, pp. 338–344, Mar. 2014.

[29] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy, "Mining data streams: A review," *ACM SIGMOD Rec.*, vol. 34, no. 2, pp. 18–26, Jun. 2005.