Florida International University Technical Report **TR-2006-09-03**

A Case for Self-Optimizing File Systems

Medha Bhadkamkar, Sam Burnett[†], Jason Liptak[‡], Raju Rangaswami, and Vagelis Hristidis
[†]Carnegie Mellon University
[‡]Syracuse University
Florida International University

medha@cs.fiu.edu srburnet@andrew.cmu.edu jjliptak@syr.edu raju@cs.fiu.edu vagelis@cs.fiu.edu

A Case for Self-Optimizing File Systems

Medha Bhadkamkar, Sam Burnett[†], Jason Liptak[‡], Raju Rangaswami, and Vagelis Hristidis
medha@cs.fiu.edu srburnet@andrew.cmu.edu jjliptak@syr.edu {raju,vagelis}@cs.fiu.edu
[†]*Carnegie Mellon University*
[‡]*Syracuse University*
Florida International University

Abstract

Efficient file systems hold one of the keys to high-performance I/O systems. Today's file systems perform a static layout of file data, aiming to preserve the directory structure of the file system and optimizing for sequential access to entire files. In this paper, we re-examine the existing state-of-the-art in file system design and find it severely lacking in an important aspect, *application awareness*. We argue that for optimal performance, file systems must self-optimize by adapting data layout to accommodate the dynamism in application access patterns. We present the design and implementation of an automated data layout reconfigurator which is at the heart of such a self-optimizing file system. Preliminary studies using file system traces indicate significant I/O performance gains when compared to a state-of-the-art ext3 file system.

1 Introduction

File systems perform a critical task within operating systems, that of providing a simple *file* abstraction to complex storage devices. To do so, they map each file or directory element to a set of *logical block addresses* (LBA), representing a portion of the disk drive space. This mapping from the file space to the drive space, as dictated by the data layout technique employed, greatly influences the performance of the file system, the applications that access the file system, and consequently the system as a whole.

The basic file and directory layout techniques in file systems have not changed significantly since the early days of the Unix Fast File System [18].

Application	CPU (s)	I/O wait (s)	Seq I/O (%)
`firefox`	1.95	5.43	13.97%
`gedit`	0.70	4.53	18.83%
`gimp`	2.67	4.37	50.49%
`oowriter`	4.83	10.82	10.47%
`xemacs`	0.74	3.59	27.03%
`xinit`	0.64	2.95	50.38%

Table 1: **Application startup profiles on a 2 GHz Intel machine with 512MB of RAM and a Maxtor 6L020J1 disk drive. The system ran Linux and an ext3 file system.**

Current file systems perform two key optimizations. First, they preserve the directory structure when mapping file system elements by allocating *cylinder groups* for storing directory sub-trees. Second, for each file, they attempt to allocate sequential space on the disk drive to optimize for sequential file access. Application access patterns, however, are typically more complex. Applications may access multiple files in different file system directory subtrees, often in a specific sequence. Further, rather than accessing an entire files sequentially, they may access files partially and even non-sequentially. These characteristics depend greatly on the specific applications running on the system, which is a dynamic set. Importantly, these characteristics put to question one of the fundamental paradigms of file system design, the *static layout* of file data.

Table 1 shows the startup profiles for six Linux-based applications using the default ext3 file system [29]. On an average, these applications spend thrice the amount of time waiting for I/Os to complete than running on the CPU when they start up. (Note that total times spent performing I/O were higher, including times when processing and I/O occurred in parallel.) Further, almost uniformly, most of the I/Os performed led to random accesses on the disk drive. This table makes two points in relation to application startup events: (i) I/O is the bottleneck, and (ii) there is significant room for improvement in I/O performance.

To improve the interactive performance of systems, such applications startup operations must be optimized by optimizing I/O systems. Further, a significant fraction of applications are also I/O bound during their lifetime, often repeating I/O access patterns. This underlines the importance of improving I/O performance. We argue that self-optimizing file systems hold the key to significantly improving the overall I/O performance in systems.

In this paper, we re-examine file system design from an application standpoint to identify performance improvement opportunities. We argue that a key reason for sub-optimal performance in existing existing file systems is ignorance of application access patterns. We make the case for a *self-optimizing file system* that dynamically adapts itself based on application access patterns. A self-optimizing file system takes into account: (i) the patterns of whole- and partial- file accesses by applications, and (ii) the frequencies of such patterns across all applications. The changing application mix over time due to dynamic running sets as well as addition of new applications to the system makes the above factors time-variant. This dynamism in file system access necessitate actively monitoring and learning from file system access patterns. A self-optimizing file system, based on this knowledge, would reconfigure file data layout as required to improve performance, in an online fashion.

Performing self-optimization within the file system requires effort along the following directions:

- *profiling* file system accesses by applications,
- *analyzing* these accesses to derive application access patterns,
- *planning* a modification to the existing data layout, and
- *executing* the plan to reconfigure the data layout.

The above steps conform to the autonomic loop as proposed by Kephart et al. [13]. In this paper, we present how each of the above stages can be realized in an actual file system with acceptable overhead. To address the important stages of analysis and planning, we use the concept of an LBA access graph, which captures both the individual and cumulative access characteristics of applications. This access graph is weighted with process-, file-, and block- level attributes. The temporal dimension can be used to further enrich the access graph, by accounting for application thinktimes. The file system data layout reconfiguration problem is thus translated to a graph layout problem. We then use a greedy heuristic that takes the access graph as input and develops a data layout reconfiguration plan. As is the case with current file systems, LBA proximity is used as an approximate measure of the access delays between blocks.

The reconfiguration plan is executed to complete the data layout reconfiguration. In an actual file system implementation, the reconfiguration of the file system metadata and data blocks could be performed during system idle time, with a frequency that is based on both system usage as well as the performance-impact of the reconfiguration. Our initial investigation reveals that on a moderately-used desktop system, although reconfiguration on a per-day basis would be feasible, using a coarser time-scale (once a week) would be adequate.

Paper contributions:

1. We re-examine file system design from an applications standpoint and provide a potential path to implementing a self-optimizing file system by presenting a possible design and a prototype implementation.

2. We model the problem of data layout reconfiguration as a graph layout problem and present a greedy heuristic for doing so.

3. We address the practical issues that arise in incorporating such a dynamic data reconfigurator in an actual file system implementation.

4. We conduct an extensive trace-drive evaluation of a prototype block layout reconfiguration system and demonstrate the benefits of the proposed approach.

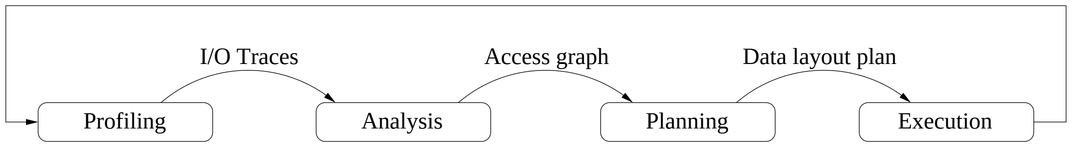

Figure 1: **The data layout reconfigurator stages.**

The rest of the paper is organized as follows. Section 2 presents the high-level approach to building a data layout reconfigurator within a self-optimizing file system. Section 3 demonstrates how the data layout reconfiguration problem can be modeled as a graph layout problem. Section 4 details a greedy heuristic for graph layout. Section 5 brings up practical issues that arise when incorporating the techniques proposed in a file system. In Section 6, we conduct a trace-based comparison study of application and system performance. Section 7 presents related work and we make concluding remarks in Section 8.

2 Overview of our Approach

Current file systems employ a a mostly static layout of file system data with the defragmentation operation being the only dynamic aspect. We argue that a static layout of file system data leads to sub-optimal performance. More specifically, we contend that file systems can improve the system I/O performance by dynamically adapting file system layout based on how the file system is used by various applications.

This section provides an overview of our approach to building a self optimizing file system. We present the data layout reconfigurator, which implements the four-stage process of *profiling, analysis, planning, and execution* and reorganizes file system data (including meta-data) on the disk drive, based on application access patterns. Please note that we have not implemented the techniques presented within a file system; instead, we have simulated it using a semi-automated process. In particular, we have built (a) a profiling tool to collect the access patterns profile, (b) an analyzing tool to analyze this profile, (c) a planning tool that, given the profile's analysis and the hard disk characteristics, generates a modification plan, and (d) a reconfiguration tool to execute the plan.

This four-stage process and the intermediate data that is created by the stages is depicted in Figure 1. The four stages are repeated continuously, thereby enabling dynamic self-optimization. We now briefly describe each of the stages.

2.1 Profiling

The profiling stage collects application I/O request patterns inside the operating system by logging each I/O request made. The I/O requests are logged with process- and block- level attributes:

- *Process-level attributes:* Timestamp, process ID, and executable name
- *Block-level attributes:* LBA, size, and mode (read/write).

The timestamp is the specific time a request was made. The *process ID* (PID) and the *executable name* help differentiate I/O requests belonging to different applications. This allows per-process I/O access patterns to be derived from a cumulative I/O trace. The *LBA* and *size* attributes provide the starting logical block address and the number of blocks to be accessed for servicing the I/O request. *Mode* distinguishes the read IO's from the write IO's.[1] The above data are collected using low-overhead kernel probing techniques.

2.2 Analysis

Once the I/O traces are obtained, we extract per-process I/O access patterns from them. We represent the access pattern for each process as a *process access graph.* An access graph $G(V, E)$, is

[1]A natural extension to this information is the file-level attribute consisting of the full-path to the file or directory accessed. Further meta-data can be marked as such and tagged with their type (e.g., inode, indirect-block, etc.). We do not discuss file-level attributes further because the current prototype of the data layout reconfigurator does not include them.

a weighted directed graph with V vertices and E edges. Every vertex represents a range of LBAs (typically representing an individual I/O request from the process) and the edge weights represent total read and writes with direction showing the transition between the LBA's[2]. A sample access graph is shown in Figure 2.

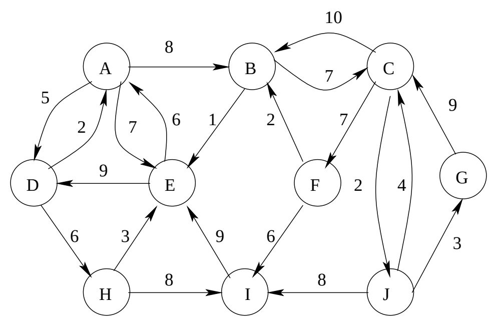

Figure 2: **A sample access graph.**

We create an such a process access graph for every process; it captures the I/O request behavior of an individual application. The *directed* and *weighted* graph representation is powerful enough to capture sequences of multi-request patterns and also the relative importance of these request patterns. If a sequence of I/O requests is repeated by the application, the corresponding edge-weights also increase.

The process access graphs obtained over a period of time (explained later) each belonging to different application invocations are then merged to form a single *master access graph*. The master graph is then used to create the data layout reconfiguration plan. Since the master graph is based on the process access graphs, the reconfiguration operation takes into account individual I/O access patterns of applications, a critical requirement for the reconfiguration operation. The algorithm for constructing the master access graph from the process graphs is elaborated in Section 3.

[2]Including application thinktimes and file-level attributes in calculating the edge-weights can further improve the weighting process. However, the current prototype does not support process thinktimes or file-level attributes, which are left as future work.

2.3 Planning

The goal of the planning stage is to develop a data layout reconfiguration plan such that the modified data layout would improve the I/O performance of various applications and consequently, the overall I/O performance of system. To do so, the planning stage takes as input the master access graph and develops a reconfiguration plan for the file system.

To develop a data layout reconfiguration plan, the planning stage first identifies contiguous portion(s) of unused disk LBA space to serve as the target location of the reconfigured data blocks. (Section 4 explains this process in more detail while also considering the possibility that such space is not available.) It then uses a greedy heuristic to determine the sequence in which the reconfigured data blocks must be placed.

In brief, the greedy heuristic proceeds by first choosing the most connected vertex, u, in the master access graph. This is the vertex with the maximum sum of incoming and outgoing edges. The next vertex chosen for placement is the vertex v most connected (in one direction only, either incoming or outgoing) to u. It is chosen to be placed either before or after u based on the direction of the edge connecting them. If v lies on the outgoing edge of u, it is placed after u and if it lies on the incoming edge it is placed before. This process is repeated until all the vertices are placed on the disk or until the edges connecting to the unplaced vertices in the master graph have weight below a certain threshold.

The above reconfiguration planner algorithm relies on the principle that edges and edge-weights in the master graph closely reflect the important I/O access patters of applications. Improving the edge weighting algorithm is therefore an important aspect of improving the overall performance of the data layout reconfigurator.

2.4 Execution

Once the new data layout plan is obtained, the final step is to execute the plan and reconfigure the existing data layout to the new layout. Such reconfiguration involves moving reconfigured data blocks to their new locations on the disk drive. Second, all file system metadata (e.g., superblock, inode, and indirect block information) must be updated to reflect the new data layout to make the file system consistent after the reconfiguration operation.

Since reconfiguration is an intrusive process, potentially slowing down the system or even requiring system down-time, it must be optimized. As we shall also explore in our evaluation (Section 6), the overhead of the reconfiguration operation is small enough that this operation could be performed on a daily basis. However, even weekly reconfiguration (or more specifically, usage-driven reconfiguration) would be a more practical approach to achieve better cost-benefit points.

3 Building the Master Access Graph

The master access graph is a key data structure used by the data layout reconfigurator. The quality of the master access graph influences greatly the quality of the overall reconfiguration operation. The *profiling* and *analysis* stages contribute towards the generation of the master access graph. In this section, we examine these two stages and the process of generating the master access graph in detail.

3.1 Profiling: Generating I/O traces

The starting point in the reconfiguration operation is understanding application I/O behavior. An application can spawn multiple processes and each process could constitute multiple threads. Each thread of execution creates correlations between the I/Os it performs. These correlations are built into the trace right from the beginning (as opposed to mined correlations in [15]). The inherent intra-process (or intra-thread) I/O correlation forms the basis of the data layout reconfiguration. If multiple processes access the same LBA, there may be correlation conflicts. This is resolved by creating a single master access graph, that captures all available correlations into a single input for the reconfiguration planner.

Figure 3 shows a fragment of an I/O trace collected from a desktop environment. Each I/O request logged has the following signature:

```
[Timestamp] [PID/TID] [Executable]
  [Starting LBA] [Size] [Mode]
```

As can be seen in Figure 3, the profiling process tags each I/O request to the file system with the corresponding executable name and its process ID (or thread ID, in case of a multi-threaded process). This allows the separation of the I/O trace collected into multiple process- or thread- specific I/O traces.

```
705423195774700 5745 screen   6914207   32  R
705423257310080 5744 bash     28511023  8   R
705423259644748 5745 utempter 24379775  8   R
705423283490088 5747 bash     7501079   40  R
705423350957048 5744 bash     7079271   8   R
705423379492524 5745 utempter 24787567  8   R
705423421266908 5753 bash     7498311   24  R
705423454005104 5745 utempter 24793415  8   R
705423493292648 5756 bash     34543375  64  R
705423565122668 5756 stty     34543439  16  R
```

Figure 3: **A sample I/O trace collected by the data layout reconfigurator.**

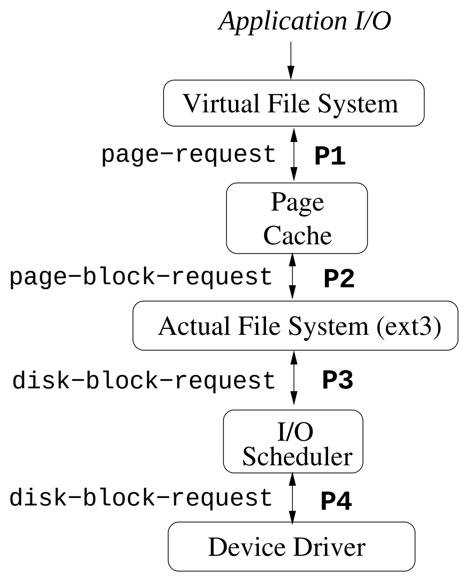

Figure 4: **I/O trace profiling points in Linux.**

3.1.1 I/O Profiling Points

To collect I/O traces, it is important to determine the ideal trace collection point inside an OS kernel. Figure 4 depicts the various points inside the Linux operating system where I/O traces could be obtained. These are labeled as `P1` through `P4`. The quality of the I/O traces collected depends strongly on the profiling point used.

`P1` is an ideal profiling point that preserves the integrity of the application I/O sequence with process- and file- level attributes. However, this profiling point cannot be used with the stock Linux kernel because block-level attributes are missing at this level. Modifying the kernel to retain this mapping would make this an ideal profile point. `P2` and `P3` provide similar traces which also include page cache effects, except that `P2`, once again, lacks block-level attributes. Finally, `P4` is a post I/O scheduler profiling point that also includes scheduler introduced effects, apart from losing process-level attributes. The I/O scheduler adds artificial correlation between I/O requests as well as eliminates process-level correlations due to re-ordering of I/O requests. It is noteworthy that several studies [15, 2, 31, 22] use `P4` as the profiling point.

The current prototype of the data layout reconfigurator uses `P3` as its profiling point. In a future implementation, `P1` would be the ideal profiling point if it can be enriched with block-level attributes.

3.2 Analysis: Generating the Master Access Graph

The I/O trace obtained from the profiling stage is split into multiple I/O traces, one per each process or thread. In particular, an I/O trace is created for each pair of (`pid`, `program`) from the trace file (`program` resolves duplicates in case of recycled PIDs). The primary benefit of separating data based on process is that it eliminates race conditions imposed by the multitasking nature of the operating system; an individual application will almost always access the same data when starting up and is likely to access similar data over its lifetime during successive invocations. However, if two applications run at the same time, it is unlikely that the sequence of requests will ever be reproduced exactly ever again.

For each process or thread, we use its I/O trace to build a directed *process access graph* $G_i(V_i, E_i)$, where each vertex represents an LBA range and each edge a transition between two LBAs. The weight on an edge (u, v) is the frequency of accesses (reads or writes) from u to v.[3]

[3]This weighting scheme reflects the current prototype we have developed. Richer weighting schemes can be developed that employ, among other information, file- and thinktime- attributes. For example, a higher weight can be associated on an edge that connects LBAs representing data of a single file. A larger process thinktime would reduce the weight, reflecting that perhaps there is a reduced dependency between the corresponding I/O requests by the process.

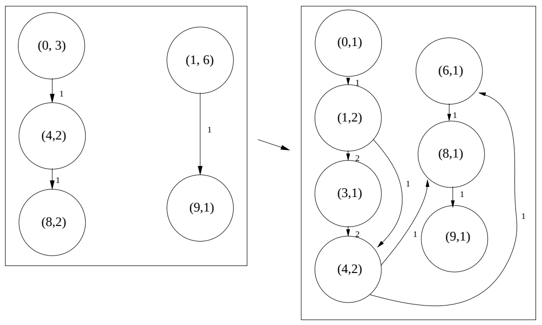

Figure 5: **Merging graphs to form a master graph.**

The next step is to merge all individual process access graphs into a single *master access graph* $G(V, E)$. Since every vertex contains a range of LBA's, it complicates the merge process. The complexity arises because the range nodes of individual process access graphs may overlap. If two vertices have overlapping ranges, they must be split into multiple vertices so that each LBA is represented in at most one range vertex. For instance, let u, v be two overlapping range nodes with LBAs $u_a, \ldots, u_b, \ldots, u_c$ and $u_b, \ldots, u_c, \ldots, u_d$ respectively. Then we split them to nodes $u' = u_a, \ldots, u_b$, $u'' = u_b, \ldots, u_c$ and $u''' = u_c, \ldots, u_d$, and rearrange edges accordingly. Note that we must check for node overlaps across process access graphs and within a process access graph, while building the process access graphs. Figure 5 shows how two range-based process access graphs are merged into the master access graph, where each vertex is represented by a pair *(start LBA, end LBA)*. The algorithm to perform this splitting procedure is as follows:

For each vertex v of the graph:

1. Examine all other vertices u in the graph. If u overlaps v, let V be the LBAs represented by v and U be the LBAs represented by u. Create new vertices for:

   (a) $I = U \cup V$

   (b) $J = U - V$

   (c) $K = V - U$

2. Move the edges coming into u to the new vertex that has the same starting LBA as u.
3. Move the edges coming out of u to the new vertex that has the same ending LBA as u.
4. Move the edges coming into v to the new vertex that has the same starting LBA as v.
5. Move the edges coming out of v to the new vertex that has the same ending LBA as v.
6. Wherever u was split, add an edge with the weight being the maximum of the incoming and outgoing weights on u.
7. Do the same for v.

Note that the above algorithm is simplistic and is for illustration purposes only. The complexity of the algorithm can be reduced by keeping the vertices sorted by their initial LBA. In any case, once this operation is completed, the master access graph is obtained. This master access graph serves as input to the *planning* stage which comes up with the data layout reconfiguration plan.

4 On-Disk Layout of Master Access Graph

In this section we explain how a master access graph G is placed on a hard disk. For clarity of presentation, in this section we assume G is stored in a fresh contiguous portion S of the disk. We elaborate on the implications of this assumption and alternative approaches in Section 5. Note that every block in G is moved from its original LBA to a new LBA in S. Also note that if a vertex of G corresponds to a block range, then multiple consecutive LBA's in S are occupied. For simplicity of presentation we assume that each vertex in G corresponds to a single block.

We start placing blocks at the middle LBA_m of S. We first place the most connected vertex u at LBA_m. The most connected vertex is the one with the highest sum of (incoming and outgoing) edge weights, i.e., with $max(w_i(u)+w_o(u))$, where $w_i(u)$ and $w_o(u)$ are the sums of incoming and outgoing edge weights of u respectively.

Next, we pick vertex $v \in G$, adjacent to u, with highest weight on the edge between v and u. If the direction of this edge is $v \to u$, we place v on LBA_v right before LBA_m. Thus, $LBA_v = LBA_m - 1$. Similarly, if the edge is $u \to v$, v is placed after u and $LBA_v = LBA_m + 1$. The next vertex x to be placed is adjacent to the ones already placed and has the highest sum of incoming or outgoing weights to the *list* of vertices already placed, i.e. $[u, v]$ or $[v, u]$. x is placed either before L, at $min(LBA_v, LBA_m) - 1$ or after L, at $max(LBA_v, LBA_m) + 1$ depending on the direction of the maximizing sum of weights. That is, if the sum of outgoing (resp. incoming) edges from x to L is maximum then x is placed before (resp. after) L. This process is repeated until all the vertices of G have been placed.

If G is not connected then once a connected component is placed we follow the previous process for

Auxiliary Methods:
```
int PlaceInLBA(LBA lba,vertex z)
/* places z at LBA lba.*/
int MaxDegreeNode(Graph g);
/* returns vertex u with max(wi(u)+wo(u))in g.*/
<v,direction> GetNextNode(LBAList L, Graph g)
/* returns the next vertex v in g with max sum
   of incoming or outgoing edge weights to L;
   direction=incoming or outgoing accordingly*/
```

Algorithm:
```
Place(Graph G, contiguous LBA space S)
    Graph g <- G;
    List L <- NULL;
    lba <- LBAl <- LBAr <- LBAm;
    /*LBAm is middle LBA of S*/
    v <- MaxDegreeNode(g);
    PlaceInLBA(lba,v);
    L.Add(v);
    while (more vertices in g) {
       <v,direction> == GetNextNode(L,g)
       if (v==NULL) /*in connected component*/
         if (direction == outgoing)
            LBAl <- LBAl-1;
            PlaceInLBA(LBAl,v);
            L.AddToFront(v);
            Remove v from g;
         else
            LBAr <- LBAr+1;
            PlaceInLBA(LBAr, v);
            L.AddToEnd(v);
            Remove v from g;
       else /*go to other connected component*/
         L <- NULL;
         LBA <- LBAl <- LBAr <- LBAm';
         /*LBAm' is middle LBA of largest
           contiguous portion of S*/
         v <- MaxDegreeNode(g);
         PlaceInLBA(LBA,v);
    }
```

Figure 6: **Algorithm for placement of vertices of G.**

each of the other connected components, that is, we start from the most connected vertex of the component.

Figure 6 presents the algorithm for this strategy. First, the vertex in G with the highest sum of edge weights is placed on the middle LBA of S. The vertices that are placed on the disk are added to the list L. `GetNextNode()` returns the subsequent vertices to be placed, which are adjacent to the vertices in L. If no such vertices are found, and if the graph still contains vertices which are not placed on the disk (vertices in another connected component of G), we repeat the above process starting at a fresh LBA in S. The algorithm terminates when all the vertices have been placed.

Example Consider the master access graph in Figure 2. C is the most connected vertex and is placed in the middle LBA of the contiguous space S. Next vertex B is placed $after$ vertex C since it is connected by an outgoing edge and has the highest weight of all the edges connected to C. Next, vertex G is placed $before$ vertex C since it is connected with the highest weight on the incoming edge to the group of vertices placed i.e. C and B. At this point $L = [G, C, B]$. All the remaining vertices are placed in a similar manner. The final sequence of vertices placed on the disk from the lowest LBA to the highest is: $L = [F, H, J, A, G, C, B, E, D]$.

In the above algorithm, for choosing the next vertex to place, we use the maximum sum of weights in any one direction from any unplaced vertex to all the placed vertices. This may introduce An improvement to this would be choose instead the left-most (resp. right-most) k vertices of the placed vertices for outgoing (resp. incoming) edges from the unplaced vertex. This would reduce the importance to dependencies on far-away LBAs.

Once the new layout is determined, the *execution phase* performs the actual task of moving block data to their new locations and performing book-keeping of file system meta-data.

5 Practical Implementation Issues and Limitations

So far, we have presented methods and procedures that can be employed in building a self-optimizing file system. We now look at various issues of feasibility, practicality, and performance, that arise when incorporating these into an actual implementation.

5.1 Profi ling phase

The first significant issue is the overhead incurred in profiling application access patterns. The primary overhead in this stage is the management of the profile data collected, before it is incorporated into the access graph. If online graph construction (elaborated below) is employed, a temporary store in kernel memory would be sufficient before the data is processed. However, in case on offline graph construction, the volume of data to be managed requires secondary storage. In such a case, writing of this profile data to a an *access log* must be performed non-intrusively. Considering that each entry in the access log is significantly smaller than the size of the original I/O being logged, this overhead is negligible. Back of the envelope calculations suggest that a moderate amount of temporary kernel memory, say 10 MB, can store as many as 200,000 access log entries before requiring to be flushed to the disk drive. This provides ample flexibility in timing the log flushes, so that they are non-intrusive.

5.2 Analysis phase

The master access graph construction algorithm for the access graph can either be online or offline. The online version constructs the access graph incrementally as access patterns are obtained, typically operating at intervals of either minutes or hours, while the offline version constructs the graph all at once after obtaining access patterns over a longer duration of time such as a day or week. While the online version has the advantage of reducing the turnaround time for the layout reconfiguration as access patterns change, the offline version has the advantages of reducing file system runtime overhead. The current prototype of the layout reconfigurator uses the latter version.

5.3 Planning phase

The data layout reconfiguration planner requires additional space on the disk for effectively planning the data layout reconfiguration operation for the execution phase. The planner assumes that there is a contiguous portion of the disk LBA space that is available to move reconfigured data into. The abundance of disk space (in comparison with the more scarce disk bandwidth) in most systems makes the use of additional space for layout reconfiguration planning practical. To execute the reconfiguration plan, the file system would read in a set of data blocks form the first location into memory and then write them back to their respective new locations on the drive.

Each time a new data layout is planned, the execution phase requires a contiguous space on the drive. This can be made available by freeing the contiguous space from the previous run, by restoring the blocks that were moved in the previous run to their original locations first. The restoring operation would require maintaining the original mappings of the file system blocks which incurs an overhead proportional to the data moved in the execution phase. After restoration, the space freed up can be re-used for the current run of the execution phase.

5.4 Execution phase

The execution of the reconfiguration plan is the most resource-consuming of the four phases. During this phase the file system must perform several operations including the actual movement of the data blocks, calculating the metadata updates that need to be performed (free-block bitmaps, superblock, inodes, indirect blocks), and performing the metadata updates. As mentioned ealier, we anticipate that this operation will be performed relatively infrequently, thereby reducing the impact of the overhead of executing the reconfiguration plan.

Although we have presented layout algorithms for entire graphs, an actual file system implementation could also perform incremental layout of graphs that are updated over time. In either version of the problem, an optimization objective that we have so far neglected is reducing the total data movement required to realize the new data layout. To reduce the amount of data movement required, we suggest the possibility of not having to reconfigure all data represented by the graph. A subset of the graph data which correspond to edge weights above a certain threshold could be chosen for reconfiguration, thereby reducing the data movement overhead.

5.5 Other issues

The self-optimizing file system reconfigures data layout so that repetitive application access patterns incur reduced I/O times and consequently better performance over time. To this end, it builds intelligence about the access patterns of each application. However, in a general time-shared system, I/Os from multiple processes get interleaved, thereby destroying the accesses patterns of individual applications. To address this issue, we bring to attention the recent success of *anticipatory scheduling* [12], which eliminates deceptive idleness in applications issuing synchronous I/O, thereby reducing the effects of interleaved I/Os. Further, readahead techniques employed by most modern file systems also reduce the effects of interleaved I/Os. With individual application accesses being more co-located in a self-optimizing file system, we believe that anticipatory scheduling will be able to reduce such effects to a greater degree.

6 System Evaluation

In this section we evaluate the performance of the Data Layout Reconfigurator (DLR) against the default layout of the ext3 file system. The results demonstrate the performance gains of using DLR. We also calculate the overhead for running the DLR and find it within acceptable bounds.

6.1 Experimental setup

To conduct the experiments, we start with a Linux system (2.6 kernel) that uses the default ext3 data layout. The DRL collects I/O traces (both application-specific and system-wide) and suggests a data layout reconfiguration plan. Next, the I/O traces corresponding to the original layout and the new layout are played out and total I/O times compared.

To evaluate the difference in performance of applications, we collect traces from six different IO bound applications mentioned in Table 1. We also collect traces from three different machines over

Drive	Capacity	Make	Model
`Twain`	17GB	Maxtor	6L120J1
`Apocalypse`	23GB	Quantum	fireballp KX27.3
`Mark`	17GB	Maxtor	6L120J1
`Maverick`	34GB	West. Dig.	WD800BB-75FJA1
`Fallout-1`	34GB	Seagate	ST336754LW
`Fallout-2`	68GB	Hitachi	HU10307073FL3600

Table 2: **Experimental set-up details.**

several days to obtain an access pattern. We then reconfigure the layout on these machines and observe the increase in performance both in the specific applications as well as the overall system I/O performance. The details of the disk drives on several machines used in the evaluation are presented in Table 2. The last two drives Fallout-1 and Fallout-2 are SCSI drives. All machines use a 2GHz Intel P4 processor and 512MB of DRAM.

We evaluate two placement strategies. The first, which we call *sequential layout*, assumes a contiguous portion is available on the disk in the mid-portion of the LBA-range of the I/O requests in the trace. Since this may not be true in a realistic environment, we also compare the performance with a *fragmented layout*.[4] In the fragmented version, we use the same set of blocks that were accessed by the application to perform the reconfiguration (i.e. no additional contiguous space is assumed), instead of the sequential set of blocks used in the sequential layout. The order in which the blocks are placed remains the same. The performance of the fragmented layout can be considered as a lower bound on the improvement.

6.2 Evaluation

For the various applications, Figure 7 shows the time required to read all initialization data from disk as specified by the six IO bound Linux applications. The first bar, Ext3, in the graph corresponds to time required to access the blocks based on the default ext3 file system layout. The second bar, DLR-Seq, corresponds to the time required to access the blocks using the sequential placement of the access graphs by the DLR and the third, DLR-Frag, corresponds to that of accessing blocks using the fragmented layout of the access graphs by the DLR. The fourth bar corresponds to the results we obtained from our implementation of C-Miner [15], which is an algorithm which produces frequent sequences of blocks from a set of short sequences. For C-Miner, we experimented with different values of *support*, ranging from two to five, and *gap* between 20 and 100, for six different sequences of the applications. The most optimal results are presented here, with support = 2 and gap = 100.

[4]Notice that the sequential layout may not provide an upper bound on the performance; a true upper bound can be obtained when a contiguous space on the outer cylinders (with considerably higher bandwidth) is allocated for the reconfigured blocks.

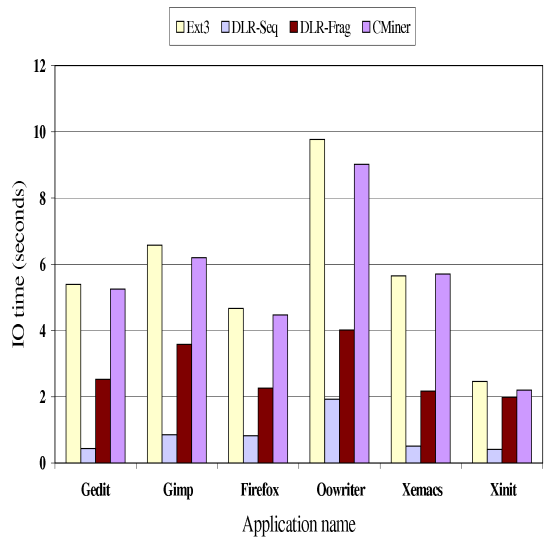

Figure 7: **Disk access times for different applications.**

On an average, sequential placement yields an improvement in performance by 57 percent and even the fragmented placement yields a 23 percent performance increase.

To perform multi-day experiments, without focusing on any specific set of applications but on the system as whole, disk activity was monitored over a span of several days on three different machines. DLR was used to plan a new data layout for all the blocks that were accessed in that duration. Figure 8 shows the performance improvement with DLR-Seq and DLR-Frag. The average performance improvement for the DLR-Seq in this case is about 70 percent.

Host	Requests	Profile time(s)	Processing time(s)	Overhead(%)	sec./req.
`twain`	43472	51336	789	1.54%	0.0181
`apocalypse`	68752	262615	1492	0.57%	0.0217
`mark`	181120	156531	12480	7.97%	0.0689

Table 3: **Overhead for analysis and planning for multi-day traces.**

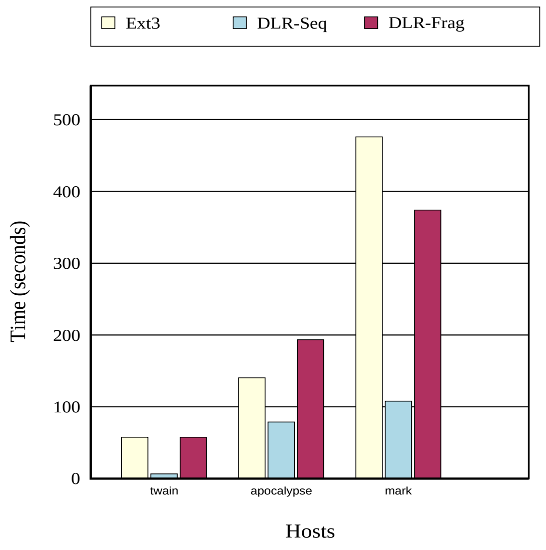

Figure 8: **Disk access times for multi-day I/O traces.**

6.3 Overhead

A self-optimizing file system, being an active system, would have some impact on the overall system performance. Table 3 shows the overhead for processing data with different number of requests. As is apparent, the percentage of time spent processing increases with the number of records. In a real-world system the amount of time spent in processing would probably be even higher. However, these three phases can be done offline, during system idle times. Additionally, the number of requests processed can be restricted. This, and other practical issues discussed in Section 5, must be considered while implementing a self optimizing file system. Furthermore, the implementation we have evaluated is an unoptimized one. An actual implementation of a self-optimizing file system would likely employ techniques to significantly decrease processing time, perhaps through approximation or more efficient processing algorithms.

6.4 Sensitivity to drive characteristics

To evaluate the effect of drive characteristics on the performance of the DLR, we conducted a sensitivity study over a range of drives mentioned in Table 2. Figure 9 shows the average IO time required for the startup of the applications mentioned in Table 1 on each of the drives. The increase in performance with the DLR using the sequential layout averages to 82 percent with a minimum of 75 percent and the maximum of 88 percent. Using the fragmented layout also results in a performance increase with an average of 44 percent; the minimum being 39 percent and the maximum as much as 52 percent.

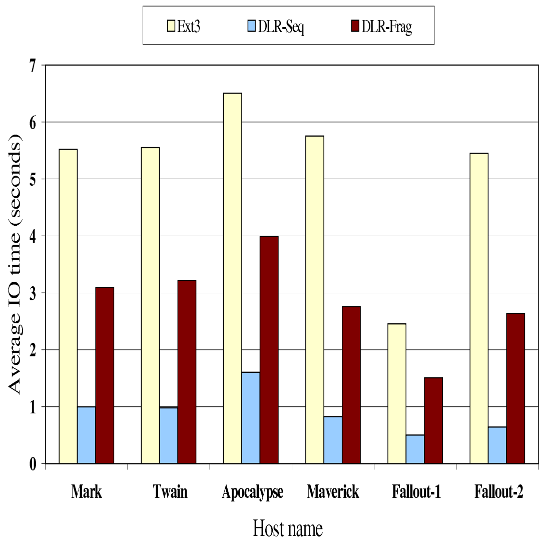

Figure 9: **Sensitivity of IO access times to various drive characteristics.**

7 Related Work

Significant research has been done in dynamically adapting storage systems to improve performance. This section briefly discusses the closely related work in this area.

Early work [10, 34] has shown that for optimal performance, data should be placed on the disk drive using the *organ-pipe* which places the frequently accessed data in the center of the disk, and the lesser accessed data on the inner and outer tracks. Several strategies have been proposed to reorganize the frequently accessed blocks on the disk. We classify these into two broad categories as block-level approaches and file-level approaches, reflecting the abstraction at which solutions are suggested.

Block-level approaches: Block-level approaches work at the disk block abstraction level, obtaining access patterns at the block level, typically at the storage system.

Ruemmler and Wilkes [22] and Vongsathorn et al. [31] use *Cylinder Shuffling* to lower the access time by minimizing the mean seek time. These techniques keep a count of the number of requests to a cylinder and move the most frequently accessed data segments to the center of the disk. In [2], the frequent blocks are copied to a reserved space in the center of the disk. These strategies emphasize on process-agnostic block counts to perform the data reorganization.

Kuenning et. al. [14] and Griffoen et. al. [9] suggest inferring frequently accessed data by mining I/O traffic. Li et. al. [15] and Gunawi et. al. [11] propose black box data mining techniques for learning block correlations and deconstructing commodity storage clusters to infer I/O traffic semantics respectively. Although used for different purposes, the above approaches nevertheless suggest infer access patterns at the block-level.

More recently, CMiner [15] modifies the frequent sequence mining algorithm of [1] to find the frequent sequences from a set of short sequences which in turn infer the correlations between blocks. The correlated sequence of blocks are then placed contiguously on the disk. This work is closely related and uses a *black-box* approach as opposed to our *white box* approach.

Salmon et al. [23] describe a generic two-tiered architecture that provides the framework for combining multiple heuristics for data reorganization. It generates constraints based on these heuristics from which a new disk layout is obtained.

For throughput improvement, Ganger et al. have proposed free-block scheduling and track-aligned extents [17, 20, 16, 24], orthogonal techniques for I/O performance improvement with block-level solutions.

Wilkes et. al. proposed HP AutoRAID [33] that automatically and transparently adapts to workload changes by using a two-level storage hierarchy; the upper level provides complete data redundancy for popular data while the lower level provides RAID 5 parity protection for inactive data.

File-level approaches: Several strategies monitor the file usage to obtain a pattern of the data accessed. In CLUMP [7], the file system monitors the file accesses and dynamically reorganizes files in clusters and/or prefetches clusters of files. The CLUMP solution operates at the granularity of whole files which does not account for intra-file block dependencies. Further, they do not consider the process-context of the file access. [27] monitors the file accesses and moves the frequently accessed data to the center of the disk during system idle periods. This approach has shortcomings similar to CLUMP.

PROFS [32] improves the IO performance Log structured File Systems(LFS) [21] by reducing the transfer time. It uses timestamp information to maintain the active ratio of the files accessed and to take advantage of the higher bandwidth in the outer tracks, it places the active files starting from the outermost track. It uses the frequency of file accesses and not the association at the file or the block level.

PLACE [19], a gray-box technique, exposes the underlying layout structure to applications, so that the applications can place the necessary files based on their access patterns to improve the I/O performance.

File access patterns can be used to predict future file accesses. Amer et al. [4] and Yeh et al. [35] use such predictions for purposes of file caching and energy conservation respectively. Similarly, in [36], file predictions are used to conserve energy in mobile computers.

In a converse approach to all of the above (including ours), Sivathanu et al. [26] propose semantically-smart disk systems (SDS), an enhanced disk system that adapts itself by observing file system I/O traffic. SDS also targets the same goal as ours, i.e. I/O performance improvement, but from a different perspective.

Other approaches: Several researchers propose generic solutions to self-optimizing I/O systems that cannot be clearly categorized into one of the above. Carnegie Mellon University's Self-* Storage project [8] addresses performance and management within storage systems by borrowing ideas from AI, control systems, and human organization. The central idea of Self-* is the concept of building large storage systems using smaller storage bricks that are self-tuning and self-adapting.

Recent work by Thereska et. al. [28] proposes to convert complex optimization and tuning problems into simpler search problems by asking the question *"What $\cdots$ if"*. Similarly, Polus [30] addresses the problem of transforming a small set of high-level QoS goals into low-level system actions by providing a reasoning and learning-based framework. Approaches for automated resource provisioning and dynamic storage reconfiguration such as Minerva [3] and Hippodrome [5, 6] require declarative input for performance requirements, workload and storage device characteristics for generating storage designs by solving multi-constraint optimization problems.

Our approach: We propose a framework for building for building self-optimizing file systems and propose a specific design of a data layout reconfigurator that will form the core of such a file system. The ideas we have presented are different from the above mentioned related work in one or more of the following aspects. First, to maintain the sequence of requests, the data layout reconfigurator collects per-process I/O profiles within the OS. Most of the above strategies use the traces which are collected when the request is sent to the disk driver to retrieve the blocks. Second, we propose using process- as well as block- level attributes to enrich the associations between I/O requests observed. From this information, the application specific requests can be filtered out which makes the access patterns in our approach application aware. The importance of application awareness in data layout reconfiguration also concurs with the findings of [25] that application-specific benchmarking is a better measure of the systems performance. In the future, we intend to incorporate file- and thinktime- attributes as well. The generic directed, weighted graph representation is powerful enough to capture all the various dimension of I/O correlations. Third, once the access patterns are obtained, the data layout reconfigurator uses weights and directions on the master access graph to determine the sequence in which the blocks should be placed. In most of the above mentioned strategies, once the frequently accessed *quantum* of data is obtained, there is no specific order in which the data is placed on the disk.

8 Conclusions and Future Work

We have argued in favor of self-optimizing file systems that continuously adapt their data layout based on application I/O access patterns to deliver optimal performance. We have presented techniques for building an automatic data layout reconfigurator, which is the core self-optimizing engine within such a file system. We discussed practical issues that arise when incorporating these techniques in an actual file system implementation. We presented a preliminary performance evaluation of the proposed approach with very promising results, and observed the potential for significant I/O performance gains.

Based on this preliminary study, we argue that self-optimizing file systems offer a critical next step in improving storage system performance and autonomy. However, an actual realization of a self-optimizing file system is a lengthy and arduous task. Further, we realize that there may be several practical considerations when building a file system based on the ideas presented. This study surely does not address all such concerns, although we did touch on a few that stood out. We believe that the potential gains to be had justify endeavors in realizing self-optimizing versions of existing file systems. We have started work on a self-optimizing version of the ext3 file system.

Acknowledgements

This work was supported in part by the National Science Foundation under grants IIS-0534530 and IIS-0552555 and the Department of Energy under grant DE-FG02-06ER25739.

References

[1] R. Agrawal and R. Srikant. Mining Sequential Patterns. In P. S. Yu and A. S. P. Chen, editors, *Eleventh International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995. IEEE Computer Society Press.

[2] S. Akyurek and K. Salem. Adaptive Block Rearrangement. *Computer Systems*, 13(2):89–121, 1995.

[3] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[4] A. Amer, D. Long, J. Paris, and R. Burns". File Access Prediction with Adjustable Accuracy. *International Performance Conference on Computers and Communication*, 2002.

[5] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. *Usenix Conference on File and Storage Technologies*, January 2002.

[6] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: Quickly Finding Near-Optimal Storage System Designs. *HP Labs Technical Report HPL-SSP-2001-05*, 2001.

[7] P. Eaton, D. Geels, and G. Mori. Clump: Improving File System Performance Through Adaptive Optimizations. December 1999.

[8] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. Self-* Storage: Brick-based Storage with Automated Administration. *Carnegie Mellon University Technical Report, CMU-CS-03-178*, August 2003.

[9] J. Griffoen and R. Appleton. Reducing File System Latency using a Predictive Apporach. *Proceedings of the Summer USENIX Conference*, pages 197–207, June 1994.

[10] D. D. Grossman and H. F. Silverman. Placement of Records on a Secondary Storage Device to Minimize Access Time. *Journal of the ACM*, 20(3):429–438, 1973.

[11] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing Commodity Storage Clusters. *Proceedings of the International Symposium on Computer Architecture*, June 2005.

[12] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. *18th Symposium on Operating Systems Principles*, September 2001.

[13] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003.

[14] G. H. Kuenning and G. J. Popek. Automated Hoarding for Mobile Computers. *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 264–275, October 1997.

[15] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 173–186, April 2004.

[16] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock Scheduling Outside of Disk Firmware. *Usenix Conference on File and Storage Technologies*, January 2002.

[17] C. R. Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle. Towards Higher Disk Head Utilization: Extracting Free Bandwith From Busy Disk Drives. *Proceedings of the OSDI*, 2000.

[18] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX*. *ACM Transactions on Computer Systems 2*, 3:181–197, August 1984.

[19] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. *Proceedings of the USENIX Annual Technical Conference*, pages 311–324, June 2003.

[20] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle. Data mining on an OLTP system (nearly) for free. *Proceedings of the ACM SIGMOD*, May 2000.

[21] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. 1991.

[22] C. Ruemmler and J. Wilkes. Disk Shuffling. *Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories*, October 1991.

[23] B. Salmon, E. Thereska, C. Soules, and G. Ganger. A Two-tiered Software Architecture for Automated Tuning of Disk Layouts. *Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.

[24] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *FAST*, 2002.

[25] M. I. Seltzer, D. Krinsky, K. A. Smith, and X. Zhang. The Case for Application-Specific

Benchmarking. In *Workshop on Hot Topics in Operating Systems*, 1999.

[26] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. *Proceedings of the USENIX Symposium on File and Storage Technologies*, pages 73–88, March 2003.

[27] C. Staelin and H. Garcia-Molina. Smart Filesystems. In *USENIX Winter Conference*, pages 45–52, 1991.

[28] E. Thereska, D. Narayanan, and G. R. Ganger. Towards self-predicting systems: What if you could ask "what-if"? *3rd International Workshop on Self-adaptive and Autonomic Computing Systems*, August 2005.

[29] S. C. Tweedie. Journaling the Linux ext2fs File System. *The Fourth Annual Linux Expo*, May 1998.

[30] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease. Polus : Growing Storage QoS Management Beyond a "Four-year Old Kid". *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 31–44, March 2004.

[31] P. Vongsathorn and S. D. Carson. A System for Adaptive Disk Rearrangement. *Softw. Pract. Exper.*, 20(3):225–242, 1990.

[32] J. Wang and Y. Hu. PROFS-Performance-Oriented Data Reorganization for Log-Structured File System on Multi-Zone Disks. In *MASCOTS '01: Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*, page 285, Washington, DC, USA, 2001. IEEE Computer Society.

[33] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *Proceedings of the Symposium on Operating System Principles*, 1995.

[34] C. K. Wong. Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *ACM Computing Surveys*, 12(2):167–178, 1980.

[35] T. Yeh, D. Long, and S. Brandt. Caching Files with a Program-based Last N Successors. *Workshop on Caching, Coherency and Consistency (WC3 '01)*, June 17, 2001.

[36] T. Yeh, D. Long, and S. Brandt. Conserving Battery Energy through Making Fewer Incorrect File Predictions. *IEEE Workshop on Power Management for Real-Time and Embedded Systems at the IEEE Real-Time Technology and Applications Symposium*, pages 30–36, May 29, 2001.