Florida International University Technical Report **TR-2007-01-01**

Feasibility, Efficiency, and Effectiveness of Self-Optimizing Storage 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

Feasibility, Efficiency, and Effectiveness of Self-Optimizing Storage Systems

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

Abstract

Recent work has proposed making intelligent use of data access patterns for building self-optimizing storage systems. However, despite the continued increase in the CPU-I/O performance gap, such systems are far from wide adoption. We argue that the key reason for the lack of real systems adopting this novel idea is that current studies leave several key questions unanswered. We pinpoint the research and practical challenges that must be addressed in building effective self-optimizing storage systems. Our answers to these questions are based on practical experience while building such a system. Our prototype self-optimizing storage system offers performance improvement of 2.5X-5X, while incurring acceptable CPU and memory overheads.

1 Motivation

I/O is a well-recognized bottleneck in computer systems for a range of workloads and applications. The startup profiles of a set of Linux applications on a six-month old ext3 filesystem showed that on an average, these applications spent thrice the amount of time waiting for I/Os to complete than running on the CPU (See Table 1). Further, on an average, 40% of the data accessed required random accesses on the disk drive. This finding makes two points about application startup events: (i) I/O is the bottleneck, and (ii) there is significant room for improvement in I/O performance. Similar observations were made by Windows and Linux kernel developers regarding OS boot events [5, 9].

The primary reason for the continued increase in the performance gap between compute and storage performance is the large mechanical seek and rotational delay overhead components of the I/O access time [1]. Current file systems, that exclusively control data layout on the disk drive employ static layouts. They aim to preserve the directory structure of the file system and optimize for sequential access to entire files. These implementations lack in an important aspect, *application awareness*. Application access patterns are typically complex. Applications may access multiple files in different file system directory subtrees, often in a specific sequence. Further, rather than accessing entire files sequentially, they may access files partially and even non-sequentially.

Application	CPU (s)	I/O wait (s)	Seq I/O (%)
firefox	1.56	3.71	51.08%
gedit	0.48	7.75	57.34%
gimp	1.71	5.88	72.97%
oowriter	3.35	7.93	60.99%
xemacs	0.92	5.94	65.35%
xinit	0.57	3.55	67.42%

Table 1: **Application startup profiles for a desktop {2GHz P4, 512MB DRAM, Maxtor 6L020J1 drive}.**

Recent work has convincingly argued that the lack of intelligent use of data access patterns is a key shortcoming of current storage stack implementations [1, 2, 7, 9]. The studies mostly advocate, with some variations, reorganizing on-disk data layout based on observed access patterns.[1] FS2 [2] proposes replication within the file system to spatially co-locate temporally correlated blocks. This strategy is restricted by the distribution of free space within the file system partition. Windows XP [9] uses the defragmenter for co-locating temporally correlated data. These solutions are file system specific solutions and as a result slow to adopt. C-Miner [7] uses data mining techniques to mine correlations between block I/O requests sent to the storage device. Though this helps find interesting inter-application dependencies, this technique is likely to introduce artificial correlations due to the I/O scheduler and may lose important correlation due to absence of process-level attributes. In ALIS [1],

[1] Recent work on improved prefetching [6, 10] advocate a complementary technique. Good prefetching solutions can help reduce the number of I/O requests; good data layout can reduce or eliminate both seek as well as rotational-delay overheads. Similar arguments hold for I/O scheduling algorithms [3].

frequently occurring I/O sequences are placed sequentially on a dedicated, *reorganized area* on the disk. Since each data block can be replicated more than once and the replica closest to the current disk head position is chosen, it incurs large consistency maintenance overhead.

Despite these recent studies, key questions still remain open: (1) *what constitutes the ideal intelligence to be used in this context?*, (2) *how to efficiently obtain such intelligence within the operating system?*, and (3) *how to effectively use such intelligence to improve storage performance?* In this paper, we address these questions by articulating the key issues in building a practical solution. We propose the addition of a *self-optimizing block storage layer* to the storage stack, that reorganizes disk data on the fly while remaining completely oblivious to the file system(s) above. A prototype augmented storage stack shows I/O performance gains of 2.5X-5X with acceptable CPU and memory overheads.

2 Key Challenges

We list six important challenges that must be addressed by an effective self-optimizing storage system.

I. Information-Rich Intelligence. A variety of information about each I/O operations are critical to optimizing data layout. These include *temporal* attributes such as timestamp, *process-level* attributes such as process ID and executable issuing the I/O request, *file-level* attributes such as the file or address-space being accessed, and *block-level* attributes such as the LBA, size, and mode of access (read/write) for the I/O request. Most current approaches only leverage a subset of the above critical information. A few use incorrect or artificial attributes introduced due to unfavorable profiling points.

II. Online and Incremental Optimization. The data layout reconfiguration must occur in an online and incremental fashion, rather than demand periodic downtime for reconfiguration. This implies that the reconfiguration engine must be able to run as a background process and data layout reconfiguration must also be carried out opportunistically during I/O idle time or using techniques such as freeblock scheduling []. Some current approaches do not address this practical issue while some other require a dedicated reconfiguration phase.

III. Controllable Overhead. Almost uniformly, most of the current approaches to build self-optimizing storage systems incur CPU, memory, disk space, and disk bandwidth overheads. An ideal solution will provide for control knobs that allow trading-off one overhead for another dynamically. Current approaches do not address this challenge.

IV. File System Independence. To ensure wide adoption, the self-optimizing storage system design must be independent of the file system. Some current solutions propose modification to existing file systems; we believe that such efforts are misdirected since they are not generic solutions. An ideal solution should seamlessly handle multiple active file systems and/or mount-points. Existing file systems should require no modification. The challenge then is to seamlessly import appropriate file system level attributes.

V. Consistency. Several self-optimizing storage solutions, including the one we propose, create copies of popular data. Consistency of data across its multiple copies is an important issue when data layout is reconfigured. Such consistency must be ensure at low cost. Some solutions invalidate non up-to-date copies rendering them useless. We propose a different approach that ensures consistency across power outages with low overhead.

VI. Ease of Integration. Self-optimizing storage systems are a significant deviation from current static-layout solutions. However, developing such systems from scratch is impractical in the short term. Practical solutions must be able to fit into the existing storage stack architecture. Implementations must be minimally intrusive to ensure quick adoption. Further, they must easily integrate with current optimizations. Current solutions do not adequately address the above issues.

3 A Self-Optimizing Block Storage Layer

We present the architecture and design of a self-optimizing storage system that addresses the challenges discussed in the previous section. Figure 1 presents its architecture. The modification to the existing storage stack is in the form of a new layer which we term *self-optimizing block storage layer*. It resides between the file system and I/O scheduler layers within the operating system. As we shall explain later, this layer can designed to be dynamically and seamlessly added/removed from the storage stack. A secondary throttle-friendly user-level daemon component communicates with this layer to perform compute and memory-intensive tasks.

The following four steps govern the operation of the self-optimizing engine at the high-level. These steps conform to the autonomic loop as proposed by Kephart et al. [4] and are also either explicitly or implicitly employed by the recently proposed approaches.

1. *profiling* application block I/O accesses,
2. *analyzing* I/O accesses to derive access patterns,
3. *planning* a modification to the data layout, and
4. *executing* the plan to reconfigure the data layout.

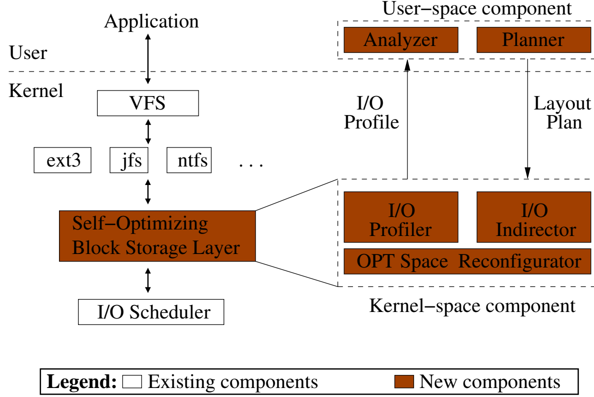

Figure 1: **System Architecture.**

Although our solution follows this well-known self-optimizing loop, there are significant differences to prior realizations of this basic idea. Each stage operates continuously in a pipelined fashion as opposed to periodic and sequential invocations of prior approaches. Further, each stage incorporates new ideas in its design and implementation. Finally the system architecture that enables the autonomic loop is novel in its ease of implementation of integration into existing storage stacks, and in its separation of kernel and user level concerns for better control on resource consumption overhead.

Reverting to the architecture, the self-optimizing block storage layer consists of three major components: *I/O Profiler, I/O Indirector,* and *OPT Space Reconfigurator*. The I/O profiler component collects block I/O access patterns while the I/O indirector directs I/O requests to the OPT partition copies (explained later on) if necessary. The OPT space reconfigurator dynamically reconfigures the contents of the OPT space. The OPT space is an optimized partition on the disk drive exclusively used by the self-optimizing block storage layer. As opposed to approaches such as FS2 [2] and C-Miner [7] wherein the self-optimizing engine operates in file system managed space, our approach uses a dedicated partition to allow a clean separation of space management responsibilities. The self-optimizing block storage layer exclusively manages the OPT partition while file systems continue to exclusively manage file system partitions.

The OPT space partition has parallels to SWAP partitions used by modern operating systems for. The SWAP partition is a reserved portion of disk to allow exceeding the limits of physical memory space by trading both average memory access performance and disk space for increased accessible virtual memory. It is a cache for memory pages. OPT space is an optimized partition of the disk space used for improving I/O performance. Here we trade disk space (as well as some CPU and memory space) for improved I/O performance. It is an optimized cache for file system blocks.

4 Implementation Outline and Key Issues

4.1 The Self-Optimizing Engine

Profiling. The *I/O profiler* component (See Figure 1) collects application I/O traces by logging each I/O request made. An application may spawn multiple processes and each process may constitute multiple threads. Each thread of execution creates correlations between the I/Os it performs. These are implicitly built into the trace (as opposed to mined correlations in [7]). The inherent intra-process 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.

I/O requests are tagged with the following temporal, process-, file-, and block- level attributes. The *timestamp* is the specific time a request was made. The *process ID* (PID) and the *executable name* help differentiate I/O requests of different processes. The address-space attribute allows the mapping of I/O requests to file or directory objects within the file system. The *LBA* and *size* attributes provide the starting logical block address and the number of blocks requested. *Mode* distinguishes read from write I/Os. The above data are collected using low-overhead kernel probing techniques at the self-optimizing block storage layer. The I/O trace is exposed to the user-space analysis component. None of the prior approaches make use of all the rich information available inside the kernel in deriving access patterns.

Analysis. The user-space *analyzer* component first splits I/O trace into multiple I/O traces, one per process or thread. The primary benefit of separating data based on process is that it eliminates I/O race conditions imposed by the multitasking nature of the operating system.

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 LBA ranges. The weight on an edge (u, v) is the frequency of accesses (reads or writes) from u to v. Applying a attenuating factor on edges for large application thinktimes and boosting those with the same address-space attribute in calculating the edge-weights further improves the weighting process.

The next step is to merge all individual process access graphs into a single *master access graph* $G(V, E)$. This merge process is non-trivial because the range nodes of individual process access graphs may overlap. If two vertices have overlapping ranges, they must be split

into multiple vertices and a directed edge added between them. Since the master graph is derived from the process access graphs, the reconfiguration operation takes into account individual access patterns of applications, a critical requirement for the reconfiguration plan. Graph merging and master access graph creation are new techniques unexplored in prior solutions.

Planning. The goal of the user-space *planner* component is to develop an OPT space reconfiguration plan that would improve the I/O performance of various applications and consequently, the overall I/O performance of system. This reconfiguration plan is built based on the master access graph from the analyzer and is specified via a reconfiguration map in which each entry indicates the LBAs of a set of blocks in file system partition(s) and their desired location in the OPT space.

The planner uses a greedy heuristic to determine the sequence in which the reconfigured data blocks must be placed in the OPT space. In brief, the greedy heuristic proceeds by first choosing the most connected (with respect to edge weights) vertex, u, in the master access graph. The next vertex chosen for placement is the vertex v most connected (in one direction only, either incoming or outgoing) to u. 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 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. The edge weighting algorithm is therefore an important aspect of improving the overall performance of the data layout reconfigurator.

Execution. The *OPT space reconfigurator* component obtains the reconfiguration map (OPT space layout plan) from the planner and executes it. This process involves copying data blocks from file system space to OPT space. Since reconfiguration is an intrusive process, it is performed in the background, either when the I/O queue is empty or when it is possible to issue such I/O requests with minimum or no overhead using optimized I/O scheduling techniques [3, 8].

The reconfiguration map is also used by the *I/O indirector* component to redirect file system block I/O requests to the OPT space if appropriate. We discuss the I/O indirection component below in more detail.

4.2 Other Components and Issues

I/O Indirection. The *I/O indirector* must efficiently direct block I/O requests to the OPT space, if the reconfiguration map contains an indirection entry for the requested block. The reconfiguration map is cached in memory and a persistent copy is stored in the beginning of the OPT space partition. Prior approaches for indirection advocate a greedy policy of choosing a copy closest to the disk head. In contrast, our approach always chooses the OPT space copy for both consistency as well as long-term I/O efficiency reasons. We address efficiency in our anticipation that subsequent I/O requests will be made to blocks correlated to the current request; the subsequent set of I/O requests will be best served from the OPT space, which is configured based on such correlation. Consistency reasons that also drive this choice appear next.

Consistency. Each time a reconfiguration of the OPT space is completed, the reconfiguration map in the OPT space is updated. Each entry in the reconfiguration map also contains a dirty bit that indicates whether the OPT space copy has been modified since the last reconfiguration. Upon writes to an OPT space block, its indirection entry in the in-memory copy of the reconfiguration map is marked as dirty, once the I/O is completed. We do not update the on-disk copy since this will substantially increase overhead for write operations.

By always choosing the OPT space copy for I/O requests, we ensure that in the presence of writes, the OPT space copies are up-to-date. However, this may render original blocks stale. The OPT space manager ensures consistency in the following manner. First, before each reconfiguration of the OPT space, dirty blocks from the OPT space are be copied to their original positions. This ensures that the file system blocks are up-to-date before a reconfiguration of the OPT space begins. Second, before the system is shut-down, the in-memory copy of the reconfiguration map is flushed to the OPT space. This is re-loaded into memory when the system is rebooted. Third, in case of power outage, upon reboot, all entries in the reconfiguration map are marked as dirty. The above techniques ensure data consistency and up-to-datedness across both reboots and power outages.

Overhead Control. Overhead in the self-optimizing storage system appears in many forms. First, the I/O profiler component buffers fragments of the I/O trace in memory before submitting them to the analyzer stage. Our un-optimized I/O trace entries average 40 bytes, which allows substantial flexibility in buffered I/O trace fragment sizes, even in kernel memory. CPU and memory overheads dominate in the analyzer and planner components. However, since these run within a user-space process, nice priority-levels and memory-pressure sensitive dynamic memory allocation can be performed to throttle resource consumption. Finally, the I/O reconfigurator can be designed to use disk idle periods and intelligent I/O scheduling so that it is almost oblivious. This

is possible since the interval between reconfiguration requests is typically large.

Ease of Integration. Ease of integration with existing technology is key to the acceptance of new ideas. The proposed architecture can be easily integrated as an independent layer in the current storage stack. Unlike prior solutions, the proposed architecture does not affect or require modification of other layers in the stack. In Linux systems, the self-optimization capability can be designed as a dynamically loadable kernel module, by introducing a new I/O scheduler type that encapsulates the actual I/O scheduler and performs the tasks of I/O profiling, I/O indirection, and OPT space reconfiguration.

5 Preliminary Results

We obtained preliminary performance and overhead numbers from our Linux-based prototype self-optimizing storage system by comparing to a vanilla kernel with an ext3 file system. These systems were six months old on an average.

Speedup of Application Startup Events. The first experiment in Figure 2 compares the improvement in I/O performance of six applications during their I/O-bound startup phase after a single reconfiguration of the OPT space. Startup events of these applications experience drastic speedup ranging from 3.5X-5X.

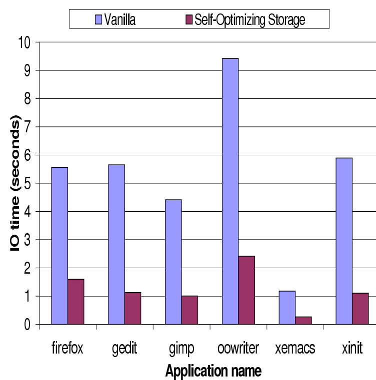

Figure 2: **I/O speedup in applications.**

Overall I/O Performance Improvement. We conducted the following experiment to estimate overall improvement in I/O subsystem performance. We trained the self-optimizing engine for a period of 12 hours on three standard development desktops. The subsequent 24 hour trace was used to evaluate the I/O performance improvement. Figure 3 shows improvement in average I/O times of 4X on an average.

Overhead. The above performance gains are drastic. However, these improvements incur overhead. Table 2 shows percentage CPU time overheads for the resource-hungry analysis and planning stages as well as the average I/O indirection time incurred in the testbed systems. These overheads, although non-trivial, can be justified easily in I/O bound systems. Further, our implementation is a highly unoptimized one. We noted several opportunities to decrease processing time through approximation as well as more efficient processing algorithms.

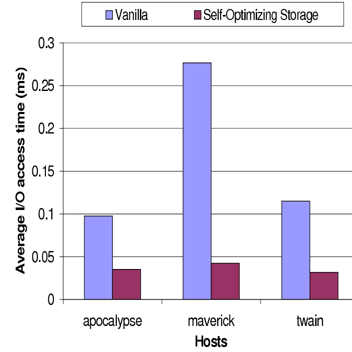

Figure 3: **Overall I/O performance improvement.**

Host m/c	Total time (hr)	Analysis time (%)	Planning time (%)	I/O Indirection (CPU-cycles/req)
apo	40	0.05%	0.06%	268
mav	37	0.27%	0.49%	297
twa	48	0.26%	0.49%	220

Table 2: **Overheads.**

6 Conclusions

Self-optimizing storage systems are a critical next step in improving storage system performance and autonomy. However, actual solutions still allude us due to several unexplored or incorrectly answered questions. This study attempts to bring out all issues facing the adoption of self-optimizing storage systems and provides answers to them based on experiences building a real system. Our preliminary prototype demonstrates that drastic I/O performance improvements are possible with acceptable overheads in other resource dimensions.

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] W. W. Hsu, A. J. Smith, and H. C. Young. The Automatic Improvement of Locality in Storage Systems.

ACM Transactions on Computer Systems, 23(4):424–473, November 2005.

[2] H. Huang, W. Hung, and K. G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. *Proceedings of the twentieth ACM Symposium On Operating Systems Principles*, pages 263–276, October 2005.

[3] 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.

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

[5] Kerneltrap.org. Linux: Boot Time Speedups Through Precaching. *http://kerneltrap.org/node/2157/*, 2004.

[6] C. Li and K. Shen. Managing Prefetch Memory for Data-Intensive Online Servers. *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 253–266, December 2005.

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

[8] 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*, October 2000.

[9] Microsoft Corporation. Fast System Startup for PCs Running Windows XP. *Windows Platform Design Notes*, December 2006.

[10] A. E. Papathanasiou and M. L. Scott. Aggressive Prefetching: An Idea Whose Time Has Come. *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems*, June 2005.