BORG: Block-reORGanization for Self-optimizing Storage Systems

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

Abstract

This paper presents the design, implementation, and evaluation of BORG, a self-optimizing storage system that performs *automatic block reorganization* based on the observed I/O workload. BORG is motivated by three characteristics of I/O workloads: non-uniform access frequency distribution, temporal locality, and partial determinism in non-sequential accesses. To achieve its objective, BORG manages a small, dedicated partition on the disk drive, with the goal of servicing a majority of the I/O requests from within this partition with significantly reduced seek and rotational delays. BORG is transparent to the rest of the storage stack, including applications, file system(s), and I/O schedulers, thereby requiring no or minimal modification to storage stack implementations. We evaluated a Linux implementation of BORG using several real-world workloads, including individual user desktop environments, a web-server, a virtual machine monitor, and an SVN server. These experiments comprehensively demonstrate BORG's effectiveness in improving I/O performance and its incurred resource overhead.

1 Introduction

There is a continual increase in the gap between CPU performance and disk drive performance. While the steady increase in main memory sizes attempts to bridge this gap, the impact is relatively small; Patterson *et al.* [25] have pointed out that disk drive capacities and workload working-set sizes tend to grow at a faster rate than memory sizes. Present day file systems, which control space allocation on the disk drive, employ static data layouts [5, 8, 15, 20, 22, 37]. Mostly, they aim to preserve the directory structure of the file system and optimize for sequential access to entire files. No file system today takes into account the dynamic characteristics of I/O workload within its data management mechanisms.

We conducted experiments to reconcile past observations about the nature of I/O workloads [7, 9, 30] in the context of current-day systems including end-user and server-class systems. Our key observations that motivate BORG are: *(i)* on-disk data exhibit a *non-uniform access frequency distribution*; the "frequently accessed" data is usually a small fraction of the total data stored when considering a coarse-granularity time-frame, *(ii)* considering a fine-granularity time-frame, the "on-disk working-set" of typical I/O workloads is dynamic; nevertheless, workloads exhibit *temporal locality* in the data that they access, and *(iii)* I/O workloads exhibit *partial determinism* in their disk access patterns; besides sequential accesses to portions of files, fragments of the block access sequence that lead to non-sequential disk accesses also repeat. We elaborate on these observations in § 2.

While the above observations mostly validate the prior studies, and may even appear largely intuitive, surprisingly, there is a lack of commodity storage systems that utilize these observations to reduce I/O times. We believe that such systems do not exist because *(i)* key design and implementation issues related to the feasibility of such systems have not been resolved, and *(ii)* the scope of effectiveness of such systems has not been determined.

We built BORG, an online *Block-reORGanizing* storage system to comprehensively address the above issues. BORG correlates disk blocks based on block access patterns to capture the I/O workload characteristics. It manages a dedicated, *BORG OPtimized Target (BOPT)* partition and dynamically copies working-set data blocks (possibly spread over the entire disk) in their relative access sequence contiguously within this partition, thus simultaneously reducing seek and rotational delays. In addition, it assimilates all *write requests* into the BOPT partition's write buffer. Since BORG operates in the background it presents little interference to foreground applications. Also, BORG provides strong block-layer data consistency to upper layers, by maintaining a persistent page-level *indirection map*.

We evaluated a Linux implementation of BORG for a variety of workloads including a development workstation, an SVN server, a web server, a virtual machine monitor, as well as several individual desktop applications. The evaluation shows both the benefits and shortcomings of BORG as well as its resource overheads. Particularly, BORG can degrade performance when a non-sequential read workload suddenly shifts its on-disk working-set. For most workloads, however, BORG decreased disk busy times in the range 6% to 50%, offering the greatest benefit in the case of non-sequential write-mostly workloads without tuning BORG parameters for optimality. A sensitivity study with various parameters of BORG demonstrates the importance of careful parameter choice which can lead to even greater improve-

Workload type	File System size [GB]	Memory size [GB]	Reads [GB]		Writes [GB]		File System accessed	Top 20% data access	Partial determinism
			Total	Unique	Total	Unique			
office	8.29	1.5	6.49	1.63	0.32	0.22	22.22 %	51.40 %	65.42 %
developer	45.59	2.0	3.82	2.57	10.46	3.96	14.32 %	60.27 %	61.56 %
SVN server	2.39	0.5	0.29	0.17	0.62	0.18	14.60 %	45.79 %	50.73 %
web server	169.54	0.5	21.07	7.32	2.24	0.33	4.51 %	59.50 %	15.55 %

Table 1: **Summary statistics of week-long traces obtained from four different systems.**

ments or degrade performance in the worst case; a self-configuring BORG is certainly a logical and feasible direction. Memory overheads of BORG are bound within 0.25% of BOPT, but CPU overheads are higher. Fortunately, most processing can be done in the background and there is ample room for improvement.

This paper makes the following contributions: *(i)* we study the characteristics of I/O workloads and show how the findings motivate BORG (§ 2) , *(ii)* we motivate and present the detailed design and the first implementation of a disk data re-organizing system that adapts itself to changes in the I/O workload (§ 3 and § 4), *(iii)* we present the challenges faced in building such a system and our solutions to it (§ 5), and *(iv)* we evaluate the system to quantify its merits and weaknesses (§ 6).

2 Characteristics of I/O Workloads

In this section, we investigate the characteristics of modern I/O workloads, specifically elaborating on those that directly motivate BORG. We collected I/O traces, downstream of an active page cache, over a one-week period from four different machines. These machines have different I/O workloads, including *office* and *developer* desktop workloads, a version control *SVN (Subversion) server*, and a *web-server*. The office and developer workloads are single-user workloads. The former workload was composed mostly of web-browsing, graph plotting with gnuplot, and several open-office applications, while the latter consisted of extensive development using emacs, gcc, and gdb, document preparation using LaTeX, email, web-browsing, and updates of the operating system. The SVN server hosted document and project code-base repositories for our 6-person research group. Finally, the web-server workload mirrored the web-requests made to our department's production web-server on one of our lab machines and served 1.1 million web requests during the trace period. Key statistics for these workloads are summarized in Table 1. We define the *on-disk working-set* (henceforth also referred to simply as "working-set") of an I/O workload as the set of all unique blocks accessed in a given interval.

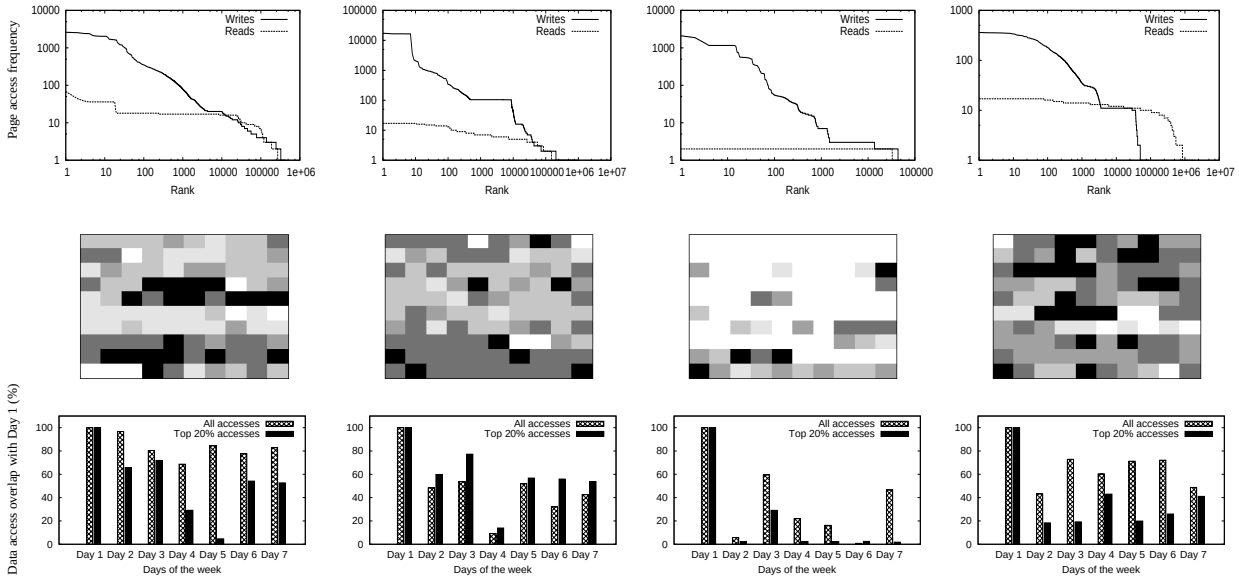
2.1 Non-uniform Access Frequency Distribution

Researchers have pointed out that file system data have non-uniform access frequency distribution [2, 29, 39]. This was confirmed in the traces that we collected where less than 4.5-22.3% of the file system data were accessed over the duration of an entire week (shown in Table 1). We observe that the office and web server workloads are read mostly, while the developer and SVN server are write mostly. Figure 1 (top row) shows page access rank-frequency plots for the workloads; file system pages were 4KB in size, composed of 8 contiguous blocks. A uniform trend to be observed across the various workloads is that the really high frequency accesses are due write requests. However, and especially in the case of the read-mostly office and web server workloads, there are a large number of read requests that occur repeatedly. In either case (read or write), the access frequencies are highly skewed. Figure 1 (middle row) depicts disk *heatmaps* created by partitioning the disk into regions and measuring accesses to each region. The heatmaps indicate that accesses, both high and low frequency ones, in most cases are spread over the entire disk area. Skewed data access frequency is further illustrated in Table 1 – the top 20% most frequently accessed blocks contributed to a substantially large ($\sim$45-66%) percentage of the total accesses across the workloads, which are within the ranges reported by Gómez and Santonja (Figure 2(a) in [7]) for the Cello traces they examined.

Based on the above observations, it is reasonable to expect that co-locating frequently accessed data in a small area of the disk would help reduce seek times when compared to the same data being spread throughout the entire disk area. Akyurek and Salem [2] have demonstrated the performance benefits of such an optimization via a simulation study. This observation also motivates reorganizing copies of popular blocks in BORG.

2.2 Temporal Locality

Temporal locality in I/O workloads is observed when the on-disk working-sets remain mostly static over short durations. Here, we refer to a locality of hours, days, or weeks, rather than seconds or minutes (typical of main memory accesses). For instance, a developer may work on a few projects over a period of a few weeks or months, typically resulting in her daily or weekly working sets being substantially smaller than her entire disk size. In servers, popularity of client requests result in temporal locality. A web server's top-level links tend to be accessed more frequently than content that is embedded much deeper in the web-site; an important new revision



(a) office (b) developer (c) SVN server (d) web server

Figure 1: **Rank-frequency, heatmap, and working-set plots for week-long traces from four different systems.** *The heatmaps (middle row) depict frequency of accesses in various physical regions of the disk, each cell representing a region. Six normalized, exponentially-increasing heat levels are used in each heatmap where darker cells represent higher frequency of accesses to the region. Disk regions are mapped to cells in row-major order.*

of a specific repository on an SVN server is likely to be accessed repeatedly over the initial weeks.

Figure 1 (bottom row) depicts the changes in the per-day working-sets of the I/O workload. The two end-user I/O workloads and the web server workload exhibit large overlaps in the data accessed across successive days of the week-long trace with the first day of the trace. There is substantial overlap even among the top 20% most accessed data across successive days. Interestingly, these workloads do not necessarily exhibit a gradual decay in working-set overlap with day 1 as one might expect, indicating that popularity is consistent across multi-day periods. The SVN server exhibits anomalous behavior because periods of high *commit* activity degrade temporal locality (new data gets created), while periods of high *update* activity improve temporal locality.

These observations indicate that optimizing layout based on past I/O activity can improve future I/O performance for some workloads and motivates planning block reorganization based on past activity in BORG.

2.3 Partial Determinism

Partial determinism in I/O workload occurs when certain non-sequential accesses in the block access sequence are found to repeat. A *non-sequential access* is defined by a sequence of two I/O operations that are addressed non-contiguous block addresses. It manifests in both end-user systems and servers. For instance, I/O during application start-up is largely deterministic, both in terms of the set of I/O requests and the sequence in which they are requested. Reading files related to a repeatable task such as setting up a project in an integrated development environment, compilation, linking, word-processing, etc. result in a deterministic I/O pattern. In a web-server, accessing a web-page involves accessing associated sub-pages, images, scripts, etc., in deterministic order.

In Table 1, we present the partial determinism for each workload calculated as the percentage of non-sequential accesses that repeat at least once during the week. The partial determinism percentages are high for the two end-user and the SVN server workloads. Further, for each of these workloads, there were a non-trivial amount of non-sequential accesses that repeated as many as 100 times. These findings suggest that there is ample scope for optimizing the repeated non-sequential access patterns.

3 Overview and Architecture

BORG is motivated by the simple question: *What storage system optimizations based on workload characteristics can allow applications to utilize the disk drive more efficiently than current systems do?* This section presents the rationale behind the design decisions in BORG and its system architecture.

3.1 BORG Design Decisions

A Disk-based Cache.

The operating system uses main memory to cache frequently and recently accessed file system data to reduce

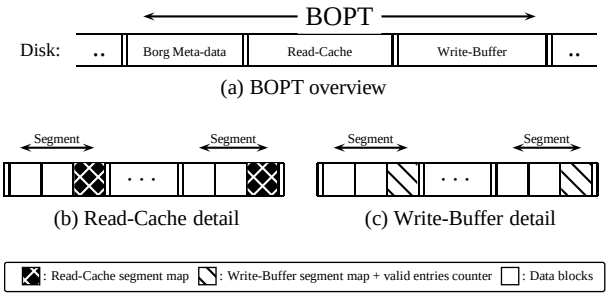


Figure 2: **Format of the BOPT partition.** *Each entry in the Write-Buffer and Read-Cache map tables is a 3-tuple of the form (FS LBA, BOPT LBA, valid bit).*

the number of disk accesses incurred. In any given duration of time, the effectiveness of the cache is largely dependent on the on-disk working-set of the I/O workload, and can degrade when this working-set increases beyond the size of the page cache. Storage optimizations such as prefetching [16, 24, 33] and I/O scheduling [13, 26, 27, 32] help improve disk I/O performance in such situations.

Using a disk-based cache as an extension of the main memory cache offers three complementary advantages in comparison to main memory caching alone, prefetching, and I/O scheduling. First, it is more effective as a cache (than main memory) because it offers a less expensive (and thus larger) as well as reliable caching solution, thus allowing data to be cache-resident for long periods of time. Second, the size of the disk-based cache can easily be configured by the system administrator without changing any hardware. And finally, dynamically optimizing data layout based on access patterns within a disk-based cache provides the unique ability to make originally non-sequential data accesses more sequential.

A Block Layer Solution.
A self-optimizing storage solution can be built at any layer in the storage stack (shown in Figure 3). Block level attributes of disk I/O operations are not easily obtained at the VFS or the page cache layer. While file system layer solutions can benefit from semantic knowledge of blocks, they incur a significant disadvantage in being tied to a specific file system (and perhaps even version). Device driver encapsulations (interface at P4) are incapable of capturing upper layer attributes, such as process ID and request time-stamp due to I/O scheduler reordering and loss of process context.

We contend that the block layer (interface at P3) is ideal for introducing block reorganization for several reasons. First, key temporal, block- and process- level attributes about disk accesses are available. Second, operating at the block layer makes the solution independent of the file system layer above, allowing it the flexibility to support multiple heterogeneous file systems simultaneously. Finally, new abstractions due to virtualization trends (e.g., virtual block device abstraction) as well as network-attached storage environments (SAN and NAS) can be supported in a straightforward way. In the case of SAN, BORG can reside on the client where all context for I/O operations are readily available with the underlying assumption that the SAN device's logical block address space is optimized for sequential access. In the case of NAS, the BORG layer can reside within the NAS device where I/O context is readily available. Modifying the NAS interface to include process associations within file I/O requests can complete the profile information.

Using an Independent BOPT partition.
The file system optimizes for sequential accesses to entire files, a common form of file access. However, certain workloads, including application start-up, content indexing and web-page requests, exhibit a more non-sequential, but deterministic, access behavior. It is thus possible that the same set of data can be accessed sequentially by some applications and non-sequentially by others. Further, some deterministic non-sequential accesses may only be temporary phenomenon.

Based on this observation, Akyurek and Salem [2] have argued in favor of *copying* rather than *shuffling* [29, 39] of data. Copying retains original sequential layouts so a choice of location based on the observed access pattern may be possible. Reverting back to the original layout is straightforward. Similarly, rather than permanently disturbing the sequential layout of files, BORG operates on copies of blocks placed temporarily in an independent BOPT partition, optimizing for the current common case of access for each data block.

3.2 BORG Architecture

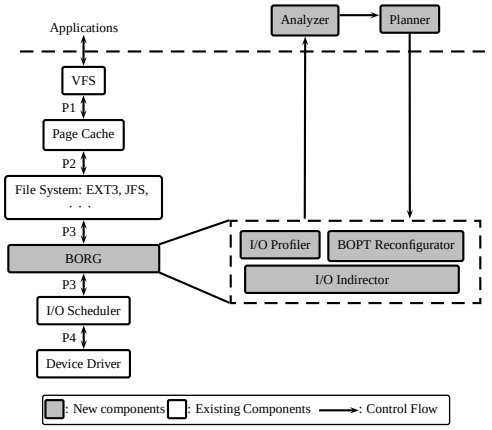


Figure 3: **BORG System Architecture.**

Abstractly, BORG follows a four-stage process:
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.

In addition, an I/O indirection mechanism runs continuously, re-directing requests to the partition that it optimizes as required. Figure 3 presents the architecture of BORG in relation to the storage stack within the operating system. The modification to the existing storage stack is in the form of a new layer, which we term *BORG layer*, that implements three major components: the *I/O profiler*, the *BOPT reconfigurator* and the *I/O Indirector*. A secondary throttle-friendly user-space component implements the *analyzer* and the *planner* stages of BORG and performs computation and memory-intensive tasks. While profiling and indirection are both continuous processes, the other stages run periodically and in succession culminating in a reconfiguration operation.

For the I/O profiler, we use a low-overhead kernel tool called `blktrace` [3]. The analyzer reads the I/O trace collected by the profiler and derives data access patterns. Subsequently, the planner uses these data access patterns and generates a new reconfiguration plan for the BOPT partition, which it communicates to the BOPT reconfigurator component. The user-space analyzer and planner components run as a low-priority process, utilizing only otherwise free system resources. Under heavy system load, the only impact to BORG is that generating the new reconfiguration plan would be delayed.

The BOPT reconfigurator is responsible for the periodic reconfiguration of the BOPT partition, per the *layout plan* specified by the planner. The reconfigurator issues low-priority disk I/Os to accomplish its task, minimizing the interference to foreground disk accesses. Finally, the I/O indirector continuously directs I/O requests either to the FS partition or the BOPT partition, based on the specifics of the request and the contents of the BOPT.

3.3 BOPT Space Management

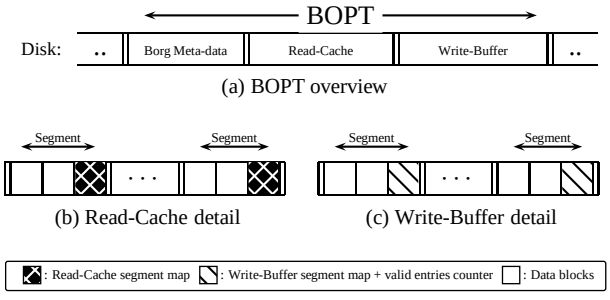


Figure 4: **Format of the BOPT partition.** *Each entry in the Write-Buffer and Read-Cache map tables is a 3-tuple of the form (FS LBA, BOPT LBA, valid bit).*

The OPtimized Target partition (BOPT) as managed by BORG is shown in Figure 4. To reduce head movement, we suggest that the BOPT partition be created adjoining the *swap* partition if virtual memory is used. BORG partitions the BOPT into three fragments: *BORG Meta-data*, *Read-cache*, and *Write-buffer*. The Read-cache and Write-buffer are further sub-divided into fixed-length segments which store both data and (valid/invalid) map entries for the segment. The in-memory indirection map (elaborated in § 4.5) maintained by BORG is a union of all the segment map entries in the BOPT. The BOPT map entries are synchronously updated each time the in-memory map information changes. Additionally, the segment map in the write-buffer contains a "valid entries counter" to track space usage in the write buffer.

Magic number	BORG BOPTpartition identifier.
`BORG_REQUIRE` bit	BOPT contains dirty data.
BOPT size	BOPT partition size.
Read-cache info	Offset and size of the Read-cache.
Write-buffer info	Offset and size of the Write-buffer.
Segment size	Fixed size of segments in the BOPT.

Table 2: **Borg meta-data.**

Table 2 depicts the BOPT meta-data fragment. It stores key persistent information that aid in the operation of BORG. The `BORG_REQUIRE` bit is *set* when the BOPT contains data that requires to be copied back to the FS. If set, the operating system initiates BORG at boot time to ensure consistent data accesses. The remaining meta-data information is used to correctly populate the in-memory indirection map structure during BORG initialization.

4 Detailed Design

In this section, we present the design details of BORG by elaborating on its individual components.

4.1 I/O Profiler

The *I/O profiler* is a data collection component that is responsible for comprehensively capturing all disk I/O activity. The I/O profiler generates an *I/O trace* that includes the temporal (timestamp of the request), process (process ID and executable) and the block-level (address range and read/write mode) attributes. We use the **Q** events reported by blktrace [3], which capture the I/O requests queued at the block layer. These include all requests as issued by the file system(s), including any journaling and/or page destageing mechanisms. We defer further details to the blktrace work [3].

4.2 Analyzer

The *analyzer* is responsible for summarizing the disk I/O workload. It first splits the I/O trace obtained from the profiler into multiple I/O traces, one per process. Each process trace is used to build a directed *process access graph* $G_i(V_i, E_i)$, where vertices represent LBA ranges and edges a temporal dependency (correlation) between two LBA ranges. The weight on an edge between vertices (u, v) represents the frequency of accesses (reads

or writes) from u to v. The *directed* and *weighted* graph representation is powerful enough to identify repeated sequences of multiple non-sequential requests.

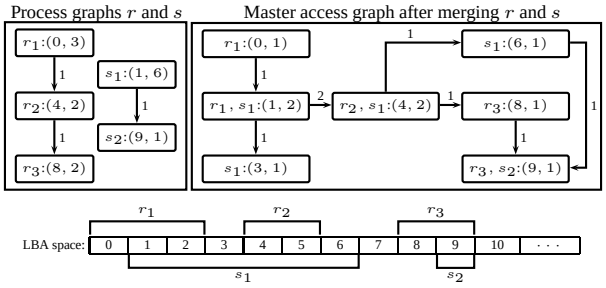


Figure 5: **Building the master access graph.** *Vertices are defined by (start LBA, size of request). Since vertices r_1 and s_1 have overlapping LBAs, r_1 is split into two vertices in the master access graph, one with size 1 and the other with the overlapping s_1 blocks, starting at LBA 1 with size 2.*

Since multiple processes may access the same LBA, a single *master access graph* $G(V, E)$, that captures all available correlations into a single input for the reconfiguration planner is created (illustrated in Figure 5). The complexity of the merge process increases if two vertices (either within the same graph or across graphs) have overlapping ranges. This is resolved by creating multiple vertices so that each LBA is represented in at most one range vertex. While we omit the detailed algorithm for vertex splitting and graph merging due to space constraints, we point out that we reduce the complexity of the merge algorithm by keeping the vertices sorted by their initial LBA. The total time complexity for the analyzer stage is given by $O(n \times l)$, where n is the number of vertices and l is the size (in LBA) of the largest vertex in the graph. Once the merge operation is completed, the master access graph, G, is obtained.

4.3 Planner

The *planner* takes the master access graph, G, as input and determines a reconfiguration plan for the BOPT partition. It uses a greedy heuristic that starts by choosing for placement the most connected vertex, u, i.e., with the maximum sum of incoming and outgoing edges (Figure 6). Next it chooses 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. The next vertex to be placed is the one most connected to the *group* $u \cup v$. This process is repeated until either all the vertices in G are placed, or the read cache in the BOPT is fully occupied, or the edges connecting to the unplaced vertices in the master graph have weight below a chosen threshold. If the graph contains disconnected components, each of these are placed as separate groups. The time complexity for the planner is $O(n \times lg(m) + n^2)$ where n is the number of vertices and m is the number of edges; finding the most connected vertex takes $O(n \times lg(m))$ time and finding the next vertex takes $O(n)$ time .

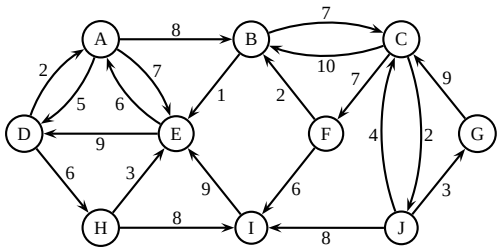


Figure 6: **Placing the master access graph.** *C is the most connected vertex and is chosen for placement first. Next, vertex B is placed $after$ vertex C since it is connected by an outgoing edge and has the highest weight of all the edges connected to C. Next, vertex G is placed $before$ vertex group $C \cup B$. The final sequence of vertices from the lowest LBA to the highest is: $L = [F, H, J, A, G, C, B, E, D]$.*

4.4 BOPT Reconfigurator

The *BOPT reconfigurator* implements the plan created by the planner component by performing the actual data movement to realize the new configuration of the BOPT. This task is complicated primarily because of consistency and overhead concerns. Overhead is partially addressed by issuing low-priority I/O requests for data layout reconfiguration, making the use of a priority scheduler a prerequisite. BORG ensures block data consistency between the FS and BOPT copies of data blocks by maintaining a persistent indirection map, termed the `borg_map`, that continuously tracks the most up-to-date location of a data block. This map is updated each time a block location changes.

The reconfigurator copies blocks in three stages; *outgoing*, where it copies all the dirty blocks that are no longer in the new plan back to the original file system (FS) location, *relocate*, where it copies blocks that have to be relocated within the BOPT, and *incoming* where it copies all the new blocks that have to be copied from the FS to the BOPT. A single data movement operation and the corresponding update on `borg_map` entry can be considered 'atomic' since any application *write* request to the *source* LBA during data movement is put on hold until after the movement is complete and the `borg_map` entry is updated. This ensures that an up-to-date version of data is always maintained by the file system.

4.5 I/O Indirector

The *I/O indirector* operates continuously, redirecting file system I/O requests as required. An I/O request may be composed of an arbitrary number of pages. Each page

request is handled separately based on (i) number of blocks that can be satisfied from the BOPT as per the `borg_map` entry, (ii) type of operation (read or write) and (iii) presence of a free page in the BOPT.

For each I/O request larger than one page, the indirector splits it into multiple per-page requests. If a mapping exists for all the pages of the I/O request in the `borg_map`, the request is indirected to the BOPT. If no mapping exists, and the request is a read request, it is issued unchanged to the file system. If only some pages of a read I/O request are mapped and the mapped entries are clean, the entire I/O is indirected to the file system; this optimization reduces the seek overhead incurred to serve the request partially from the BOPT and the rest from the FS. For a write request, when no mapping exists for any of the pages, the blocks are written to a *write-buffer* portion of the BOPT reserved for assimilating write requests (if space permits) along with an additional request for updating corresponding mapping entries in the `borg_map`. For partially-mapped writes, the mapped blocks are indirected to their BOPT locations; the unmapped pages are also absorbed in the write-buffer, space permitting, otherwise these are issued to the FS.

4.6 Kernel Data Structures

The persistent data structure `borg_map` is implemented as a radix tree such that given an FS LBA, the BOPT LBA can be retrieved efficiently and vice-versa. It also maintains the *dirty* information for the BOPT LBAs. For every page of 4KB, BORG stores 4 bytes each for the forward and the reverse mapping and one dirty bit. If all the pages in the BOPT of size S GB are occupied, the worst case memory requirement is $2 \times S$ MB (S MB for forward and reverse mapping each), and $\frac{S}{2^5}$ MB for the dirty information. Thus, in the worst case, `borg_map` requires memory of 0.25% of the size of the BOPT partition, an acceptable requirement for kernel-space memory.

5 Implementation Issues

In this section, we discuss the particularly challenging aspects of the BORG implementation that help address data consistency and overhead.

5.1 Persistent Indirection Map

Since BORG replicates popular data in the BOPT space, the system must ensure that reads are always up-to-date versions of data, including after a clean shutdown or a system crash. BORG implements a persistent `borg_map`, which is distributed within read-cache and write-buffer segments of the BOPT. Map entries on-disk are updated (along with their in-memory version) each time the BOPT partition is reconfigured or when a new map entry is added to accommodate a new write absorbed into the BOPT. Upon writes to an existing BOPT mapped block, its indirection entry in the in-memory copy of the reconfiguration map is marked as dirty, once the I/O is completed. To minimize overhead for BOPT writes, we chose not to maintain dirty information in the on-disk copy. Upon reboot after an unclean shut down, all entries in the persistent map are marked as dirty and future IOs to these blocks are directed to the BOPT.

5.2 Optimizing Reconfiguration

Consider a set L of n LBAs, $L_1, \cdots, L_n$, sequentially located in the BOPT space. L forms a *chain* if $\forall L_i \in L$, where $L_i \neq L_n$, L_i has to be relocated to location $L_i + 1$ and L_n is an outgoing block. If L_n, has to be relocated to L_1 within the BOPT, L forms a *cycle*. Information about chains and cycles, that occur exclusively for the relocated blocks, can be used to further optimize data movement during the reconfiguration operation. If a cycle exists, it is broken by copying the last block L_n back to the FS (if dirty) and then deleting the plan entry for that block; an additional plan entry is then created to mark this as `incoming` block to L_o. Next, all remaining blocks belonging to the same chain/cycle are copied to their new locations in the BOPT. To do so, the reconfigurator issues all reads to the source locations in parallel; once all reads have been completed, it issues all the writes in parallel, in each case allowing the I/O scheduler to optimize the request schedule.

5.3 Other Data Consistency Issues

BORG maintains metadata at the granularity of a *page* (rather than *block*) to reduce metadata memory requirement (by 8X for Linux file systems). Consequently, the indirector must carefully handle I/O requests whose sizes are not multiples of the page-size and/or which are not page-aligned to the beginning of the target partition. We address this issue via I/O request splitting and page-wise indirection, techniques borrowed from our earlier work on EXCES [38], a block-layer extension that manages a persistent cache for reducing disk power consumption.

BORG is dynamically included in the I/O stack by substituting the `make_request` function of the device targeted for performance optimization. While module insertion is straightforward, module removal/unload must ensure that all the data from the BOPT has been copied back to their original locations in the file system and handle foreground I/O correctly. Once again, BORG uses techniques from EXCES [38] and flushes dirty BOPT blocks to their original locations in the file system upon removal. To address race conditions caused when an application issues an I/O request to a page that is being flushed to disk, BORG stalls (via `sleep`) the foreground I/O operation until the specific page(s) being flushed are written to the disk.

Host	Make	Model	RAM (MB)	Capacity (GB)		
				Total	FS	BOPT
01	WD	2500AAKS	1024	250	46	1
02	WD	360GD	1024	39	24	2
03	Maxtor	6L020L1	1024	20	15	2
04	WD	2500AAKS	1024	250	180	8
05	Maxtor	6L020J1	1536	20	8	1

Table 3: **Experimental test-bed details.**

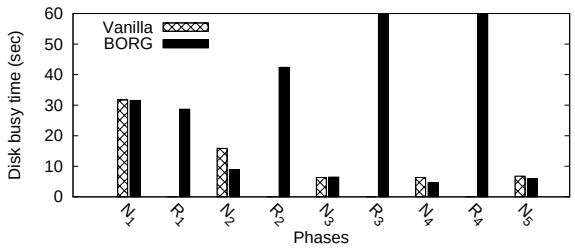


Figure 7: **Disk busy times in various phases of the SVN server trace replay.** *R_i and N_j correspond to reconfiguration phase i and non-reconfiguration phase j respectively. R_3 and R_4 are beyond the y-axis range with values of 272 and 564 seconds respectively.*

6 Evaluation

In this section, we compare the performance of BORG with a *vanilla* system in which all the blocks are located in their original FS space under various workloads to answer the following questions.

(i) How well does BORG perform? We use the total disk busy time (i.e., excluding all idle periods) as the primary metric of performance. Due to BORG's optimizations, apart from the potentially improved head positioning times, the degree of merging of requests may also be increased when compared with the vanilla configuration, thus changing the request pattern itself. Thus, the more common I/O response time metric is an ill-suited choice. The total disk busy time (henceforth simply referred to as disk busy time) is also robust against the trace-replay speedups we employ in some of our experiments.

(ii) Why is BORG effective? We would like to know if BORG performance gains are because of the sequentiality or the proximity of data (or both) in the BOPT. We use two metrics, *average seek distance* and *non-sequential accesses percentage* for this purpose. The latter is measured as $\frac{\#\text{ Seeks}}{\#\text{ BlocksRead}}$. Since non-sequential accesses are at least an order of magnitude less efficient than sequential accesses, even a small reduction in this metric may lead to substantial performance benefit.

(iii) When is BORG not effective? BORG can degrade the system performance for certain workloads. We evaluate BORG for varying workloads to determine in which cases it could perform worse than the vanilla system.

(iv) How much CPU resource overhead does BORG incur? While the upper bound on memory overhead was examined in § 4.6, the CPU resources consumed by BORG should also be within acceptable limits. We use the execution times for various stages of BORG as an approximate measure of CPU resource utilization.

(v) How is BORG affected by its parameters? We perform a sensitivity analysis of BORG to its parameters - reconfiguration interval, BOPT size, and BOPT write buffer fraction - to evaluate their impact on performance.

Experimental Setup. All experiments were performed on machines running the Linux 2.6.22 kernels. We used host machines, `01` through `05`, with differing hardware configurations and disk drives (Table 3). We used `reiserfs` for `01` and `03`, and `ext3` for the rest. No additional hardware was required to implement BORG.

We conducted four different sets of experiments. The first set uses week-long traces of a developer's system and a Subversion control server (SVN). The second experiment is an actual deployment of a web server that mirrors our CS department's web server. The third experiment evaluates BORG performance in a virtual machine environment. The fourth experiment evaluates the performance improvement due to BORG for application start-up events.

In each experiment, we performed 4 reconfigurations equally spaced in time; this gave us a reasonable number of phases for detailed analysis. By not choosing more favorable times such as idle disk periods based on well-known diurnal workload cycles, we would only overestimate the overhead of BORG during reconfiguration. We further discuss the selection of this parameter in § 6.5 and § 7. Finally, we use the notation R_i and N_j in various graphs to denote reconfiguration phase i and non-reconfiguration phase j respectively.

6.1 Trace Replay

To evaluate BORG under realistic workloads, we conducted trace replay experiments using SVN server and developer workloads described in Table 1. For the traces and the replay, we used *blktrace* and *btreplay* respectively [3]. We used an acceleration factor of 168X that reduces the experimentation time from one week to a manageable one hour after verifying that the resultant block access sequence was unaffected. The trace-playback acceleration factor was reverted to 1X during each reconfiguration operation to accurately estimate reconfiguration overhead. Since we only measure disk busy times, the comparison between normal and reconfigurations phases remains valid despite the varying acceleration factors.

6.1.1 SVN Server

For the SVN server trace replay, we used the host `02` (Table 3). The write buffer size was set to 20% of the BOPT

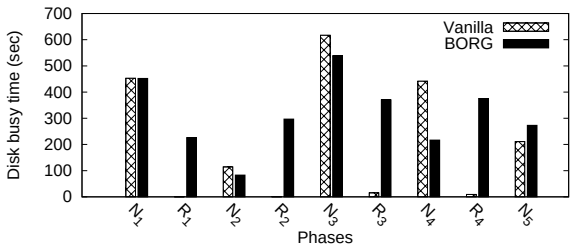


Figure 8: **Disk busy time in various phases of the developer trace replay.**

size. Figure 7 shows the disk busy times during different phases of the experiment. In all the reconfiguration phases the busy time with BORG is notably higher than the vanilla case. This is due to substantial head movement during reconfiguration for relocating blocks. The longest reconfiguration phase lasted approximately 10 minutes. R_3 and R_4 have substantially higher busy time than the previous two reconfigurations. After trace analysis, we found that while the amount of data movement was similar across the four reconfiguration instances, in the latter two phases, the I/O scheduler merge ratio and the sequential disk accesses dropped dramatically; this can be attributed to the blocks relocated within the BOPT being spread out more than in the previous reconfigurations. However, As is evident by the vanilla busy times, the foreground activity during these intervals are negligible and thus the increased reconfiguration durations have little impact to foreground I/O.

In all the non-reconfiguration phases, each of which lasted 1.75 days approximately, BORG offers better performance for foreground I/O than the vanilla configuration. In the best case (range N_2), BORG decreases the disk busy time by approximately 45%. This is a surprising result, since as per Figure 1(c), the working-set for this workload undergoes rapid shifts. The explanation lies in the fact that the SVN server is a write-intensive workload and the BOPT write-buffer is successful in sequentializing a rapidly changing, possibly non-sequential, write workload. Analysis of the block level traces revealed that with BORG, the non-sequential access percentage reduced from 1.70% to 1.15%, and the average seek distance reduced from 704 to 201 cylinders during the non-reconfiguration phases.

6.1.2 Developer

For the developer trace replay, we used the host `01` (Table 3) with the BOPT write buffer set to 40% of the BOPT size. Figure 8 shows the disk busy time for this experiment in various phases. With this workload, the longest measured reconfiguration phases were R_3 and R_4 which lasted approximately 7 minutes each. We observe reduced disk busy times (13% to 50% reductions) across the non-reconfiguration periods, except for N_5 which shows an increase of 25%. Overall, the developer workload is a write-mostly workload and thus, largely conducive to BORG optimizations. Analysis of the block level traces revealed that overall, the non-sequential access percentage reduced from 3.93% to 3.30%, and the average seek distance reduced from 1203 to 782 cylinders when using BORG.

6.2 Web Server

To evaluate BORG in a production server environment, we made a copy of the our Computer Science department web server on the `04` machine (see Table 3), and replayed all the web requests for a week. During this week a total of 1137234 requests to 256017 distinct files were serviced. We set BORG to reconfigure four times during this period, using an BOPT of 8GB ($< 5\%$ of the 180GB web server file system). To measure the influence of the I/O history, we conducted two sets of experiments. In the first experiment, we used all the traces gathered from the beginning of the experiment as input to the reconfigurator (*cumulative*). For the second, we only used the portion of the trace corresponding to the period since the last reconfiguration (*partial*).

Figure 9: **Disk busy time for the week long web log replay.** Borg-C *and* Borg-P *correspond to using cumulative and partial traces respectively.*

Figure 9 shows the improvements in disk busy time across various non-reconfiguration and reconfiguration phases during the experiment. For both the cumulative and partial experiments, BORG reduces disk busy time in all non-reconfiguration phases with reductions ranging from 14% to 35% for cumulative and 5% to 39% for the partial configuration, except N_5 for cumulative which reported a 6% increase for cumulative due to drastic change in the last interval's workload. Disk busy times in reconfiguration phases are typically higher due to the overhead of copying data to the BOPT. Nev-

ertheless, BORG was able to obtain overall reductions of 14% and 18% for cumulative and partial configuration. It is interesting to note that short term training yielded better results in this case, perhaps due to greater influence of short term locality.

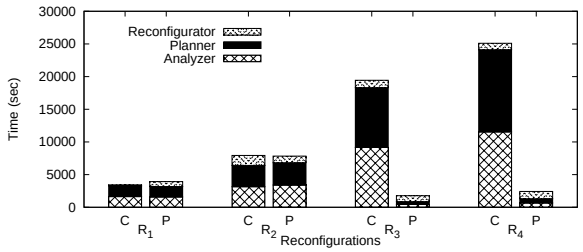


Figure 10: **BORG overhead.** *Bars* C *and* P *represent the cumulative and partial traces experiments respectively.* R_i *indicates the* i*th reconfiguration.*

Next we examine operational overhead of BORG. Figure 10 shows the amount of time spent in each phase of the reconfiguration. With cumulative traces, the time required for the analyzer and planner phases increases linearly. While the planner and analyzer stages can run as low-priority tasks in the background, we must point out that the current implementation of BORG analyzer and planner stages are highly unoptimized and there is substantial room for improvement. We discuss possible improvements for both subsystems in §7. With partial traces, the time increases until the second reconfiguration, but then decreases and stays almost constant for the following ones, indicating a gradually stabilizing working-set.

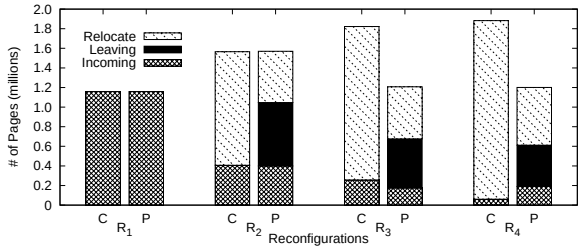


Figure 11: **Differences in the reconfiguration plans.**

To explain this further, we examined the reconfiguration plan divided by the type of operation (refer to § 4.4), presented in Figure 11. We note that the size of the plan consistently increases when using cumulative traces and most of the movements correspond to page relocates, which are page movements within the BOPT itself. The story is quite different for partial traces, where we see pages not accessed in the past interval leaving the BOPT, resulting in a smaller working set in the BOPT and thereby reducing the amount of work done by the analyzer, planner, and reconfigurator stages.

6.3 Virtual Machines

BORG has the potential to significantly improve the performance of virtualized environments, by co-locating multiple virtual machine (VM) localities spread across a physical volume. We evaluated the impact on the per-VM boot time and the overall performance of virtual machines by deploying BORG in a Xen [4] virtual machine monitor. We created four VMs, each with 64MB memory and 4GB physical partition on the host `05` (refer to Table 3). For evaluating boot-time improvement, we trained BORG with the boot-time events of all the virtual machines. BORG showed an almost 3X average improvement in VM boot-times - 167 seconds with vanilla and 65 seconds with BORG.

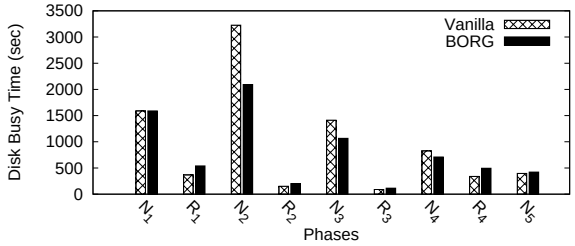


Figure 12: **BORG with a VMM.**

To measure normal execution performance improvement for the VMs, we ran the Postmark benchmark which emulates an e-mail server and creates and updates small files. We set the number of files to be 2000 in 500 directories and performed 200,000 transactions. We reconfigured BORG after every 20% of the benchmark was executed with the training set including I/O operations from the start of the execution of the benchmark. The results for the I/O performance are shown in Figure 12. As before, the reconfiguration phases see a increased disk busy times with BORG. For the normal operation, as the training set increases, the disk busy times with BORG starts decreasing. Overall, there is an average decrease of 6% in busy time during the non-reconfiguration phases. However, this improvement is not consistent; performance degrades substantially even during normal operation in the early stages of the benchmark. The *loss of process context* inside the VMM is a key problem that tends to convert sequentially laid out files into non-sequential upon reconfiguration. We believe that making BORG aware of process context inside the VMM [14] can substantially improve the BOPT layout, resulting in much greater performance benefit.

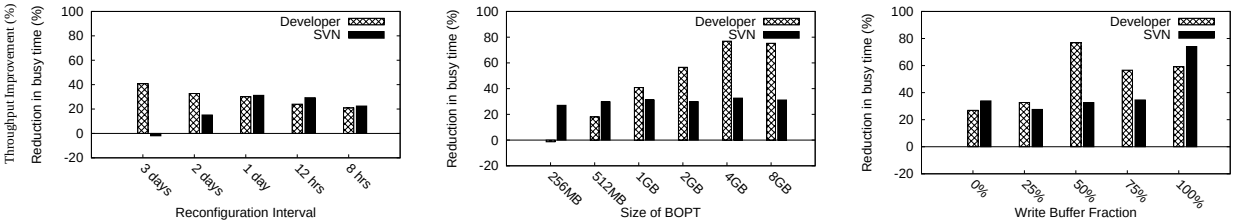


Figure 13: A sensitivity analysis of BORG performance to its configurable parameters.

App	Start-up time		Rand. I/O %		Avg seek (#cyl)	
	V	B	V	B	V	B
firefox	3.71	2.32	2.7	1.2	132	37
oowriter	5.30	2.74	3.8	0.2	193	20
xemacs	7.26	2.72	2.1	0.3	87	9
acroread	6.20	2.65	4.6	0.1	39	9
eclipse	4.12	1.52	2.5	0.3	198	29
gimp	3.62	3.66	2.5	2.1	102	63
ooimpress	5.18	1.97	2.7	0.3	61	39

Table 4: **Application start-up time improvement.** *V: vanilla, B: BORG.*

6.4 Application Start-up

We evaluated the impact of BORG on I/O-bound start-up phase for common desktop applications using host `03`. We first trained the system for a duration of approximately four hours, during which we invoked a subset of the applications listed in Table 4 (but specifically excluding `gimp` and `ooimpress`) multiple times for performing common office tasks. We invalidated the page cache periodically to artificially dilate time and simulate system reboots. Table 4 shows the difference in application start-up times, the percentage of sequential accesses and average seek overhead. For the applications that were used in training, it can be observed that there is a noticeable improvement in the I/O time with BORG - at least 43% for `oowriter` and up to 67% for `eclipse`. Further, it is interesting to observe that although the percentage of sequential I/Os decreases for `oowriter` and `acroread` with BORG, there is an overall improvement in I/O performance, possibly due to a reduction in the rotational overhead . There is barely any difference in the performance for untrained application `gimp`. However, although `ooimpress` was not used in the training, its start-up user-time shows an improvement of 62% in the average I/O time; this can be attributed to large shared libraries also used by the `oowriter` which was included in training.

6.5 Sensitivity Analysis

To gain maximum performance improvement with BORG its configurable parameters – the reconfiguration interval, the BOPT size, and the BOPT write buffer fraction — must be carefully tuned for a given workload. To better understand the effects of these parameters, we replayed the developer and the SVN workload traces on host `01` varying each of these parameters over a range of values. In all the experiments, the trace replay begins at the same starting point, that is after a *base reconfiguration*, which uses the first six hours of the trace as the training period. We measure the relative efficiency of disk I/O using BORG averaged across the non-reconfiguration intervals by reporting the *improvement in disk busy time throughput* (referred to henceforth as "throughput improvement") when compared to a vanilla system.

6.5.1 Reconfiguration Interval

Figure 13 (left) shows the percentage improvement over the vanilla system. The reconfiguration interval is varied from 8 hours (18 reconfigurations) to 3 days (1 reconfiguration). To bootstrap the sensitivity analysis, the BOPT size is fixed to 1GB, with 50% reserved for write buffering in this experiment. For the developer workload, as the reconfiguration interval increases the throughput increases, the training set becomes larger, and BORG can more effectively capture the working-set. For the SVN workload, the performance decreases for higher intervals. This is because the SVN working-set changes quite frequently (elaboration in § 2 and Figure 1(c)).

6.5.2 BOPT size

We use the best-case reconfiguration intervals of 3 days for the developer and a day for the SVN workload from the previous experiment. We vary the BOPT size from 256MB to 8GB, of which the write buffer is always chosen as 50% of the BOPT size. Figure 13 (middle) shows that as the BOPT size increases, BORG's performance with the developer workload increases since the developer workload has a larger working set. When most of the blocks in the working set can be accommodated in the BOPT, the performance improvement stabilizes. Since the working set size for the SVN workload is relatively smaller, the performance improvement is almost same for the BOPT sizes >256MB.

6.5.3 Write Buffer Variation

From our previous results, we pick an interval of 3 days and 1 day and BOPT size of 2GB and 4GB for the developer and the SVN workloads respectively. We vary the write buffer from 0-100%. Figure 13 (right) shows that for the developer workload, not having a write buffer re-

sults in the lowest throughput. There is a steady increase in performance, peaking at 50% write buffer. Thereafter, it starts falling since read performance begins to degrade due to lesser available read cache. For the write-intensive SVN workload, the performance increases with increase in the write buffer size, since all the writes can be co-located in the BOPT partition.

Configuring BORG parameters The above experiments indicate that configuring parameters incorrectly can lead to sub-optimal performance improvements with BORG. Fortunately, iterative algorithms can be easily employed to identify better parameter combinations in a straightforward way. Exploring such iterative algorithms more formally is one aspect of our future work.

7 Discussion

While our experiences with BORG have been mostly positive, there are several directions in which the current version can be either improved or extended. We now discuss some of the significant directions that can serve as subjects of future investigation.

Analyzer and Planner optimization. The current versions of the analyzer (§ 4.2) and the planner (§ 4.3) components of BORG do not use the results of past executions and therefore incur higher overheads for every subsequent reconfiguration when using cumulative traces for training. Each of these components can be substantially optimized by making them more intelligent. The analyzer can build the master access graph incrementally rather than from scratch; likewise, the planner can incrementally create the new plan for BOPT reconfiguration during each iteration.

Alternate BOPT layout strategies. The current version of BORG uses a simple BOPT layout strategy starting from the most-connected vertex – the vertex with the highest sum of its edge-weights – in the master access graph, and then choosing the vertex most connected to it, and so on. Alternate layout strategies can be envisioned that potentially yield greater benefit. For instance, the placement can begin with the nodes connected to the highest weight edge, and then resorting to the same incremental addition of vertices. Alternatively, a distributed layout algorithm can be designed which uses many starting points for building the layout.

Timely reconfiguration. The current reconfiguration trigger in BORG is based on a fixed interval. However, opportune times for performing reconfiguration are during periods of no or low foreground I/O activity, especially for workloads that exhibit obvious idle or peak periods of activity. More sophisticated triggers can use alternate metrics to identify "unwanted" or "much needed" reconfiguration, such as the BOPT hit rate or the percentage of sequential accesses pre- and post- indirection to evaluate the effectiveness of the current BOPT layout. The above techniques can help substantially reduce the impact of reconfiguration to foreground I/O and increase the effectiveness of each reconfiguration operation.

Avoiding performance degradation. BORG can degrade performance for certain workloads, for instance, a read-intensive workload that has a very large or unstable working-set (§ 6.2). Future versions of BORG can be made intelligent to measure the impact of reconfiguration on such workloads by comparing the percentage sequentiality and the spatial locality for the accesses before (vanilla) and after (BORG) the indirection operation. If these metrics degrade post-BORG, BORG can be disabled. Such a mechanism will allow system performance to degrade gracefully in the event that the workload is not conducive to benefit from block reorganization.

8 Related Work

We examine related work by organizing the literature into block and file based approaches.

8.1 Block level approaches

Early work [41] on optimized data layout argued for placing the frequently accessed data in the center of the disk. Vongsathorn *et al.* [39] and Ruemmler and Wilkes [29] both propose Cylinder Shuffling. Ruemmler and Wilkes specifically demonstrated that performing relatively infrequent shuffling led to greater improvement in I/O performance. In Akyurek and Salem's work [2], the authors demonstrated the advantages of copying over shuffling and the importance of reorganization at the block (rather than cylinder) level. These early data clustering approaches emphasized on process- and access-pattern- agnostic block counts to perform the data reorganization and reported simulation-based results.

Researchers have also investigate self-optimizing RAID systems. Wilkes *et al.* proposed HP AutoRAID [40], a controller-based solution, that transparently adapts to workload changes by using a two-level storage hierarchy; the upper level provides data redundancy for popular data while the lower level provides RAID 5 parity protection for inactive data. Work on eager writing [42] and distorted mirrors [35] address mirrored/striped RAID configurations primarily for database OLTP workload (which are characterized by little locality or sequentiality) that choose to write to a free sector closest to the head position on one more disk drives. While we are yet to explore BORG's use in multi-disk systems, the optimizations used in BORG are different and mostly complementary to the above proposals, whereby BORG attempts to capture longer-term on-disk working-sets within a dedicated volume.

Hu *et al.*'s work on Disk Caching Disk [10] uses an additional logging disk (or disk partition) to perform writes

sequentially and subsequently, destage to their original locations. Write buffering in BORG is slightly different in that writes to data already in the BOPT partition are written in place. The DCD work does not optimize for data read operations; BORG optimizes reads as well so head movement is substantially restricted.

Among recent work on block reorganization, C-Miner [17] uses advanced data mining techniques to mine correlations between block I/O requests. These techniques can be utilized in BORG to infer complex disk access patterns. The Intel Application Launch Accelerator [12] reorganizes blocks used during application start-up to be more sequential, but does not provide a generic solution to improve overall disk I/O performance of the system.

For throughput improvement, Schindler *et al.* have proposed free-block scheduling [18] and track-aligned extents [31] which use intelligent I/O scheduling rather than block reorganization. These are complementary techniques that can be used in conjunction with BORG.

Among block level approaches, our work is closest to ALIS [9], wherein frequently accessed blocks as well as block sequences are placed sequentially on a dedicated, reorganized area on the disk. There are key differences in design and implementation, though. First, BORG incurs reduced space, maintenance, and metadata overhead since it maintains at most one copy of each data block. The multiple replicas in ALIS can become stale quickly in write-intensive workloads. Further, unlike BORG, ALIS does not optimize write traffic. Finally, the evaluation of ALIS techniques is performed using a disk simulator with trace playback. On the other hand, we implement and evaluate an actual system, thereby having the opportunity to address a greater detail of system implementation issues.

8.2 File level approaches

In one of the early file oriented approaches, Staelin *et al.* [36] proposed monitoring file accesses and moving frequently accessed files (entirely) to the center of the disk. Log-structured file systems (LFS [28]) offer superior performance for workloads with large number of small writes by batching disk writes to the end of a disk-sequential *log*. BORG writes all data to the BOPT partition to achieve a similar effect, but also attempts to co-locate a majority of read operations with the writes. Matthews *et al.* [19] proposed an optimization to LFS by incorporating data layout reorganization to improve read performance. Their use of block access graphs is similar to the process access graphs used in BORG. Their LFS-specific solution moves blocks within the LFS partition storing exactly one copy of each block at any time. Since BORG stores two copies, it can optimize for sequential and application-driven deterministic, non-sequential accesses simultaneously.

Researchers have also explored data- and application-specific layout mechanisms. Ganger and Kaashoek [6] advocate co-locating inodes and file blocks for small files. Conversely, PLACE [23], exposes the underlying layout structure to applications, so they can perform custom data placement. Sivathanu *et al.* [34] propose semantically-smart disk systems (SDS) that infer file system semantic associations for blocks, subsequently used for aligning files with track boundaries. Windows XP [21] uses the defragmenter for co-locating temporally correlated file data for speeding up application start-up events. BORG is a generic solution in comparison to the above approaches, since it creates a block reorganization mechanism that can adapt to an arbitrary workload.

Mac OS's HFS Plus [1] uses adaptive hot file clustering to migrate and sequentially store hot files of small sizes near the volume's metadata. In contrast, BORG operates at the block layer and sequentializes by copying (rather than migrating) hot block sequences, which may span either partial or multiple files.

Among file level approaches, BORG is closest to the FS2 [11]. FS2 proposes replication of frequently accessed blocks based on disk access patterns in file system free space. This strategy, unfortunately, also restricts the degree of seek and rotational-delay optimization due to the distribution of free space. Since FS2 may create multiple copies of a block simultaneously, staleness, and consequently, space and I/O bandwidth wastage, become important concerns (similar to those in ALIS); BORG maintains at most one extra copy of each block and its strength is in being a non-intrusive, storage-stack friendly, and file system independent (portable) solution.

9 Conclusions and Future Work

We presented BORG, a self-optimizing layer in the storage stack that automatically reorganizes disk data layout to adapt to the workload's disk access patterns. BORG was designed to optimize both read and write traffic dynamically by making reads and writes more sequential and restricting majority of head movement within a small optimized disk partition. A Linux implementation of BORG was evaluated and shown to offer performance gains in the average case for varied workloads including office and developer class end-user systems, a web server, an SVN server, and a virtual machine monitor. Disk busy time reductions with BORG across these workloads during non-reconfiguration intervals range from 6% (for the VM workload) to 50% (for the developer server workload), with even greater improvements possible with careful parameter selection within BORG.

BORG performs occasionally worse than a vanilla system, specifically when a read-mostly workload dras-

tically shifts its working set. BORG is able to easily address changing working-sets with a (possibly non-sequential) write workload, since it has the ability to absorb and sequentialize writes inside the BOPT. A sensitivity analysis revealed the importance of choosing the right configuration parameters for reconfiguration interval, BOPT size, and the write-buffer fraction. Fortunately, simple iterative algorithms can be quite effective in identifying the right parameter combination; a formal investigation of such an approach is an avenue for future work. The memory and CPU overheads incurred by BORG are modest, and with ample scope for further optimization. In summary, we believe that BORG offers a novel and practical approach to building self-optimizing storage systems that can offer large I/O performance improvements in commodity environments.

Acknowledgments

We would like to thank the reviewers of this paper and especially our shepherd Ken Salem for insightful feedback that helped improve the content and presentation of this paper substantially. This work was supported in part by the NSF grants CNS-0747038 and IIS-0534530 and by DoE grant DE-FG02-06ER25739.

References

[1] HFS Plus Volume Format. *http://developer.apple.com/technotes/tn/tn1150.html*.

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

[3] J. Axboe. blktrace user guide, February 2007.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the ACM SOSP*, October 2003.

[5] H. Custer. Inside the Windows NT File System. *Microsoft Press*, August 1994.

[6] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. *Proc. of the USENIX Technical Conference*, 1997.

[7] M. Gómez and V. Santonja. Characterizing Temporal Locality in I/O Workload. *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2002.

[8] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit filesystem with Guaranteed Rate IO. *SGI Technical Report*, 1996.

[9] 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, Nov 2005.

[10] Y. Hu and Q. Yang. DCD – Disk Caching Disk: A New Approach for Boosting I/O Performance. *Proc. of the International Symposium on Computer Architecture*, 1995.

[11] H. Huang, W. Hung, and K. G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. *Proc. of the ACM SOSP*, October 2005.

[12] Intel Corporation. Intel application launch accelerator. *http://support.intel.com/support/chipsets/iaa/*, 1998.

[13] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. *Proc. of the ACM SOSP*, Sept 2001.

[14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. *Proc. of the USENIX Technical Conference*, May 2006.

[15] D. Kleikam, D. Blaschke, S. Best, and B. Arndt. JFS for Linux. *http://jfs.sourceforge.net/*.

[16] C. Li and K. Shen. Managing Prefetch Memory for Data-Intensive Online Servers. *Proc. of the USENIX FAST*, December 2005.

[17] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. *Proc. of the USENIX FAST*, April 2004.

[18] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock Scheduling Outside of Disk Firmware. *Proc. of USENIX FAST*, January 2002.

[19] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. *Proc. of the ACM SOSP*, 1997.

[20] 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.

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

[22] Namesys, Inc. The ReiserFS File System. *http://www.namesys.com/*.

[23] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. *Proc. of the USENIX Technical Conference*, June 2003.

[24] A. E. Papathanasiou and M. L. Scott. Aggressive Prefetching: An Idea Whose Time Has Come. *Proc. of the Workshop on HotOS*, June 2005.

[25] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM SOSP*, December 1995.

[26] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. *Proc. of the USENIX Technical Conference*, June 2003.

[27] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). *Proc. of the IEEE RTSS*, December 2003.

[28] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proc. of the ACM SOSP*, October 1991.

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

[30] C. Ruemmler and J. Wilkes. UNIX disk access patterns. *Proc. of the Winter USENIX Conference*, 1993.

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

[32] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. *Proc. of the Winter USENIX Technical Conference*, 1990.

[33] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. *Proc. of the Workshop on HotOS*, May 1997.

[34] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. *Proc. of the USENIX FAST*, March 2003.

[35] J. A. Solworth and C. U. Orji. Distorted Mirrors. *Proc. of PDIS*, 1991.

[36] C. Staelin and H. Garcia-Molina. Smart Filesystems. In *USENIX Winter Conference*, 1991.

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

[38] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon, and R. Rangaswami. EXCES: External caching in energy saving storage systems. *IEEE HPCA*, 2008.

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

[40] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *Proc. of the ACM SOSP*, 1995.

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

[42] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. *Proc. of USENIX FAST*, January 2002.