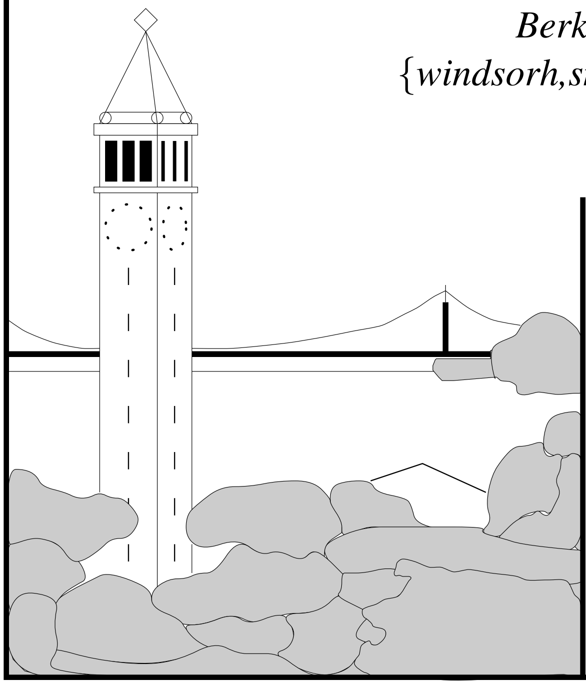
The Automatic Improvement of Locality in Storage Systems

Windsor W. Hsu[†‡]
Alan Jay Smith[‡]
Honesty C. Young[†]

[†]*Storage Systems Department*
Almaden Research Center
IBM Research Division
San Jose, CA 95120
{*windsor,young*}@*almaden.ibm.com*

[‡]*Computer Science Division*
EECS Department
University of California
Berkeley, CA 94720
{*windsorh,smith*}@*cs.berkeley.edu*



Report No. UCB/CSD-03-1264

July 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The Automatic Improvement of Locality in Storage Systems

Windsor W. Hsu[†‡] Alan Jay Smith[‡] Honesty C. Young[†]

[†]Storage Systems Department
Almaden Research Center
IBM Research Division
San Jose, CA 95120
{windsor,young}@almaden.ibm.com

[‡]Computer Science Division
EECS Department
University of California
Berkeley, CA 94720
{windsorh,smith}@cs.berkeley.edu

Abstract

Disk I/O is increasingly the performance bottleneck in computer systems despite rapidly increasing disk data transfer rates. In this paper, we propose Automatic Locality-Improving Storage (ALIS), an introspective storage system that automatically reorganizes selected disk blocks based on the dynamic reference stream to increase effective storage performance. ALIS is based on the observations that sequential data fetch is far more efficient than random access, that improving seek distances produces only marginal performance improvements, and that the increasingly powerful processors and large memories in storage systems have ample capacity to reorganize the data layout and redirect the accesses so as to take advantage of rapid sequential data transfer. Using trace-driven simulation with a large set of real workloads, we demonstrate that ALIS considerably outperforms prior techniques, improving the average read performance by up to 50% for server workloads and by about 15% for personal computer workloads. We also show that the performance improvement persists as disk technology evolves. Since disk performance in practice is increasing by only about 8% per year [18], the benefit of ALIS may correspond to as much as several years of technological progress.

1 Introduction

Processor performance has been increasing by more than 50% per year [16] while disk access time, being limited by mechanical delays, has improved by only about 8% per year [14, 18]. As the performance gap between the processor and the disk continues to widen, disk-based storage systems are increasingly the bottleneck in computer systems, even in personal computers (PCs) where I/O delays have been found to highly frustrate users [25]. To make matters worse, disk recording density has recently been rising by more than 50% per year [18], far exceeding the rate of decrease in access density (I/Os per second per gigabyte of data), which has been estimated to be only about 10% per year in mainframe environments [35]. The result is that although the disk arm is only slightly faster in each new generation of the disk, each arm is responsible for serving a lot more data. For example, Figure 1 shows that the time to read an entire disk using random I/O has increased from just over an hour for a 1992 disk to almost 80 hours for a disk introduced in 2002.

Funding for this research has been provided by the State of California under the MICRO program, and by AT&T Laboratories, Cisco Corporation, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sun Microsystems, Toshiba Corporation and Veritas Software Corporation.

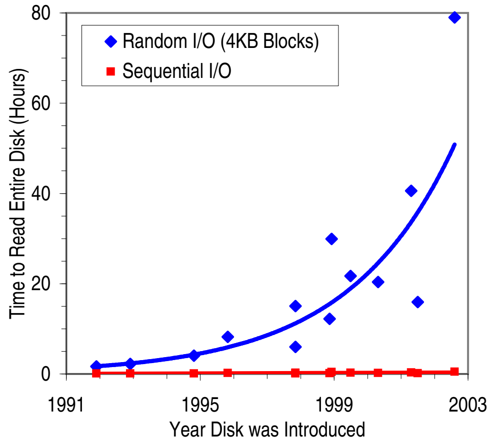


Figure 1: Time Needed to Read an Entire Disk as a Function of the Year the Disk was Introduced.

Although the disk access time has been relatively stable, disk transfer rates have risen by as much as 40% per year due to the increase in rotational speed and linear density [14, 18]. Given the technology and industry trends, such improvement in the transfer rate is likely to continue, as is the almost annual doubling in storage capacity. Therefore, a promising approach to increasing effective disk performance is to replicate and reorganize selected disk blocks so that the physical layout mirrors the logically sequential access. As more computing resources become available or can be added relatively easily to the storage system [19], sophisticated techniques that accomplish this transparently, without human intervention, are increasingly possible. In this paper, we propose an autonomic [23] storage system that adapts itself to a given workload by automatically reorganizing selected disk blocks to improve the spatial locality of reference. We refer to such a system as Automatic Locality-Improving Storage (ALIS).

The ALIS approach is in contrast to simply clustering related data items close together to reduce the seek distance (*e.g.,* [1, 3, 43]); such clustering is not very effective at improving performance since it does not lessen the rotational latency, which constitutes about 40% of the read response time [18]. Moreover, because of inertia and head settling time, there is only a relatively small time difference between a short seek and a long seek, especially with newer disks. Therefore, for ALIS, reducing the seek distance is only a secondary effect. Instead, ALIS focuses on reducing the number of physical I/Os by transforming the request stream to exhibit more sequentiality, an effect that is not likely to diminish over time with disk technology trends.

ALIS currently optimizes disk block layout based on the observation that only a portion of the stored data is in active use [17] and that workloads tend to have long repeated sequences of reads. ALIS exploits the former by clustering frequently accessed blocks together while largely preserving the original block sequence, unlike previous techniques (*e.g.,* [1, 3, 43]) which fail to recognize that spatial locality exists and end up rendering sequential prefetch ineffective. For the latter, ALIS analyzes the reference stream to discover the repeated sequences from among the intermingled requests and then lays the sequences out physically sequentially so that they can be effectively prefetched. By operating at the level of the storage system, rather than in the operating system, ALIS transparently improves the performance of all I/Os, including system generated I/O (*e.g.,* memory-mapped I/O, paging I/O, file system metadata I/O) which may constitute well over 60% of the I/O activity in a system [43]. Trace-driven simulations using a large collection of real server and PC workloads show that ALIS considerably outperforms previous techniques to improve read performance by up to 50% and write performance by as much as 22%.

The rest of this paper is organized as follows. Section 2 contains an overview of related work. In Section 3, we present the architecture of ALIS. This is followed in Section 4 by a discussion of the methodology used to evaluate the effectiveness of ALIS. Details of some of the algorithms are presented in Section 5 and are followed in Section 6 by the results of our performance analysis. We present our conclusions in Section 7 and discuss some possible extensions of this work in Section 8. To keep the chapter focused, we highlight only portions of our results in the main text. More detailed graphs and data are presented in the Appendix.

2 Background and Related Work

Various heuristics have been used to lay out data on disk so that items (*e.g.,* files) that are expected to be used together are located close to one another (*e.g.,* [10, 34, 36, 40]. The shortcoming of these *a priori* techniques is that they are based on static information such as the name space relationships of files, which may not reflect the actual reference behavior. Furthermore, files become fragmented over time. The blocks belonging to individual files can be gathered and laid out contiguously in a process known as defragmentation [8, 33]. But defragmentation does not handle inter-file access patterns and its effectiveness is limited by the file size which tends to be small [2, 42]. Moreover, defragmentation assumes that blocks belonging to the same file tend to be accessed together which may not be true for large files [42] or database tables, and during application launch when many seeks remain even after defragmentation [25].

The *posteriori* approach utilizes information about the dynamic reference behavior to arrange items. An example is to identify data items – blocks [1, 3, 43], cylinders [53], or files [48, 49, 54] – that are referenced frequently and to relocate them to be close together. Rearranging small pieces of data was found to be particularly advantageous [1] but in doing so, contiguous data that used to be accessed together could be split up. There were some early efforts to identify dependent data and to place them together [4, 43], but for the most part, the previous work assumed that references are independent, which has been shown to be invalid for real workloads (*e.g.,* [20, 21, 45, 47]). Furthermore, the previous work did not consider the aggressive sequential prefetch common today, and was focused primarily on reducing only the seek time.

The idea of co-locating items that tend to be accessed together has been investigated in several different domains – virtual memory (*e.g.,* [9]), processor cache (*e.g.,* [15]), object database (*e.g.,* [50]) *etc.* The basic approach is to pack items that are likely to be used contemporaneously into a superunit, *i.e.,* a larger unit of data that is transferred and cached in its entirety. Such clustering is designed mainly to reduce internal fragmentation of the superunit. Thus the superunits are not ordered nor are the items within each superunit. The same approach has been tried to pack disk blocks into segments in the log-structured file system (LFS) [32]. However, storage systems in general have no convenient superunit. Therefore such clustering merely moves related items close together to reduce the seek distance. A superunit could be introduced but concern about the response time of requests in the queue behind will limit its size so that ordering these superunits will still be necessary for effective sequential prefetch.

Some researchers have also considered laying out blocks in the sequence that they are likely to be used. However, the idea has been limited to recognizing very specific patterns such as sequential, stepped sequential and reverse sequential in block address [5], and to the special case of application starts [25] where the reference patterns likely to be repeated are identified with the help of external knowledge.

There has also been some recent work on identifying blocks or files that tend to be used together or in a particular sequence so that the next time a context is recognized, the files and blocks can be prefetched accordingly (*e.g.,* [12, 13, 28, 29, 39]). The effectiveness of this approach is constrained by the amount of locality that is present in the reference stream, by the fact that it may not improve fetch efficiency since disk head movement is not reduced but only brought forward in time, and by the burstiness in the I/O traffic which makes it difficult to prefetch the data before it is needed.

3 Architecture of ALIS

ALIS consists of four major components. These are depicted in the block diagram in Figure 2. First, a *workload monitor* collects a trace of the disk addresses referenced as requests are serviced. This is a low overhead operation and involves logging four to eight bytes worth of data per request. Since the ratio of I/O traffic to storage capacity tends to be small [17], collecting a reference trace is not expected to impose a significant overhead. For instance, [17] reports that logging eight bytes of data per request for the Enterprise Resource Planning (ERP) workload at one of the nation's largest health insurers will create only 12 MB of data on the busiest day.

Periodically, typically when the storage system is relatively idle, a *workload analyzer* examines the reference data collected to determine which blocks should be reorganized and how they should be laid out. Because workloads tend to be bursty, there should generally be enough lulls in the storage system for the workload analysis to be performed daily [17]. The analysis can also be offloaded to a separate machine if necessary. The workload analyzer uses two strategies, each targeted at ex-

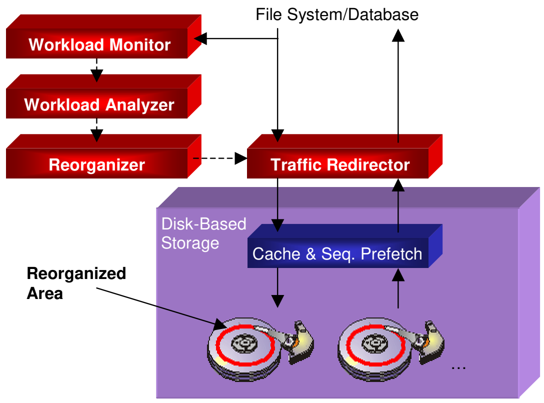


Figure 2: Block Diagram of ALIS.

ploiting a different workload behavior. The first strategy attempts to localize *hot*, *i.e.,* frequently accessed, data in a process that we refer to as *heat clustering*. Unlike previously proposed techniques, ALIS localizes hot data while preserving and sometimes even enhancing spatial locality. The second strategy that ALIS uses is based on the observation that there are often long read sequences or *runs* that are repeated. Thus it tries to discover these runs to lay them out sequentially so that they can be effectively prefetched. We call this approach *run clustering*. The various clustering strategies will be discussed in detail in Section 5.

Based on the results of the workload analysis, a *reorganizer* module makes copies of the selected blocks and lays them out in the determined order in a preallocated region of the storage space known as the *reorganized area (RA)*. This reorganization process can proceed in the background while the storage system is servicing incoming requests. The use of a specially set aside reorganization area as in [1] is motivated by the fact that only a relatively small portion of the data stored is in active use [17] so that reorganizing a small subset of the data is likely to achieve most of the potential benefit. Furthermore, with disk capacities growing very rapidly [14], more storage is available for disk system optimization. For the workloads that we examined, a reorganized area 15% the size of the storage used is sufficient to realize nearly all the benefit.

In general, when data is relocated, some form of directory is needed to forward requests to the new location. Because ALIS moves only a subset of the data, the directory can be simply a lookaside table mapping only the data in the reorganized area. Assuming 8 bytes are needed to map 8 KB of data and the reorganized area is 15% of the storage space, the directory size works out to be equivalent to only about 0.01% of the storage space (15%*8/8192 $\approx$ 0.01%). The storage required for the directory can be further reduced by using well-known techniques such as increasing the granularity of the mapping or restricting the possible locations that a block can be mapped to. The directory can also be paged. Such actions may, however, affect performance.

Note that there may be multiple copies of a block in the reorganized area because a given block may occur in the heat-clustered region and also in multiple runs. The decision of which copy to fetch, either original or one of the duplicates in the reorganized area is determined by the *traffic redirector* which sits on the I/O path. For every read request, the traffic redirector looks up the directory to determine if there are any up-to-date copies of the requested data in the reorganized area. If there is more than one up-to-date copy of a block in the system, the traffic redirector can select the copy to fetch based on the estimated proximity of the disk head to each of the copies and the expected prefetch benefit. A simple strategy that works well in practice is to give priority to fetching from the runs. If no run matches, we proceed to the heat-clustered data, and if that fails, the original copy of the data is fetched. We will discuss the policy of deciding which copy to select in greater detail in Section 5.

For reliability, the directory is stored and duplicated in known, fixed locations on disk. The on-disk directory is updated only during the process of block reorganization. When writes to data that have been replicated and laid out in the reorganized area occur, one or more copies of the data have to be updated. Any remaining copies are invalidated. We will discuss which copy or copies to update later in Section 6.3. It suffices here to say that such update and invalidate information is maintained in addition to the directory. At the beginning of the block reorganization process, any updated blocks in the reorganized region are copied back to the home or original area. Since there is always a copy of the data in the home area, it is possible to make the reorganization process resilient to power failures by using an intentions list [52]. With care, block reorganization can be performed while access to data continues.

The on-disk directory is read on power-up and kept static during normal operation. The update or invalidate information is, however, dynamic. Losing the memory copy of the directory is thus not catastrophic, but having non-volatile storage (NVS) would make things simpler for maintaining the update/invalidate information. Without NVS, a straightforward approach is to periodically write the update/invalidate information to disk. When the system is first powered up, it checks to see if it was shut down cleanly the previous time. If not, some of the update/invalidate information may have been lost. The update/invalidate information in essence tracks the blocks in the reorganized area that have been updated or invalidated since the last reorganization. Therefore, if the policy of deciding which blocks to update and which to invalidate is based on regions in the reorganized area, copying all the blocks in the update region back to the home area and copying all the blocks from the home area to the invalidate region effectively resets the update/invalidate information.

ALIS can be implemented at different levels in the storage hierarchy, including the disk itself, especially if predictions about embedding intelligence in disks [11, 26, 41] come true. We are particularly interested in the storage adaptor and the outboard controller, which can be attached to a storage area network (SAN) or an internet protocol network (NAS), because they provide a convenient platform to host significant resources for ALIS, and the added cost can be justified, especially for high performance controllers that are targeted at the server market and which are relatively price-insensitive [19]. For the personal systems, a viable alternative is to implement ALIS in the disk device driver.

More generally, ALIS can be thought of as a layer that can be interposed somewhere in the storage stack. ALIS does not require a lot of knowledge about the underlying storage system, although its effectiveness could be increased if detailed knowledge of disk geometry and angular position are available. So far, we have simply assumed that the storage system downstream is able to service requests that exhibit sequentiality much more efficiently than random requests. This is a characteristic true of all disk-based storage systems and it turns out to be extremely powerful, especially in view of the disk technology trends.

Note that some storage systems such as RAID (Redundant Array of Inexpensive Disks) [6] adaptors and outboard controllers implement a virtualization layer or a virtual to physical block mapping so that they can aggregate multiple storage pools to present a flat storage space to the host. ALIS assumes that the flat storage space presented by these storage systems performs largely like a disk so that virtually sequential blocks can be accessed much more efficiently than random blocks. This assumption tends to be valid because, for practical reasons such as to reduce overhead, any virtual to physical block mapping is typically done at the granularity of large extents so that virtually sequential blocks are likely to be also physically sequential. Moreover, file systems and applications have the same expectation of the storage space so it behooves the storage system to ensure that the expectation is met.

4 Performance Evaluation Methodology

In this paper, we use trace-driven simulation [46, 51] to evaluate the effectiveness of ALIS. In trace-driven simulation, relevant information about a system is col-

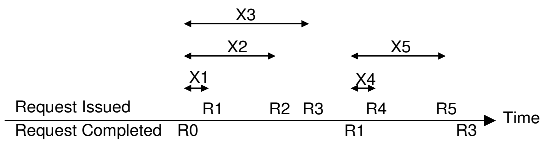


Figure 3: Intervals between Issuance of I/O Requests and Most Recent Request Completion.

lected while the system is handling the workload of interest. This is referred to as tracing the system and is usually achieved by using hardware probes or by instrumenting the software. In the second phase, the resulting trace of the system is played back to drive a model of the system under study. Trace-driven simulation is thus a form of event-driven simulation where the events are taken from a real system operating under conditions similar to the ones being simulated.

4.1 Modeling Timing Effects

A common difficulty in using trace-driven simulation to study I/O systems is to realistically model timing effects, specifically to account for events that occur faster or slower in the simulated system than in the original system. This difficulty arises because information about how the arrival of subsequent I/Os depend upon the completion of previous requests cannot be easily extracted from a system and recorded in the traces. See [18] for a brief survey of how timing effects are typically handled in trace-driven simulations and how the various methods fail to realistically model real workloads. In this paper, we use an improved methodology that is designed to account for both the feedback effect between request completion and subsequent I/O arrivals, and the burstiness in the I/O traffic. This methodology is developed in [18] and is outlined below. We could have simply played back the trace maintaining the original timing as in [43] but that would result in optimistic performance results for ALIS because it would mean more free time for prefetching and fewer opportunities for request scheduling than in reality.

Results presented in [17] show that there is effectively little multiprogramming in PC workloads and that most of the I/Os are synchronous. Such predominantly single-process workloads can be modeled by assuming that after completing an I/O, the system has to do some processing and the user, some "thinking", before the next set of I/Os can be issued. For instance, in the timeline in Figure 3, after request R0 is completed, there are delays during which the system is processing and the user is thinking before requests R1, R2 and R3 are issued. Because R1, R2 and R3 are issued after R0 has been completed, we consider them to be dependent on R0. Similarly, R4 and R5 are deemed to be dependent on R1. Presumably, if R0 is completed earlier, R1, R2 and R3 will be dragged forward and issued earlier. If this in turn causes R1 to be finished earlier, R4 and R5 will be similarly moved forward in time. The "think" time between the completion of a request and the issuance of its dependent requests can be adjusted to speed up or slow down the workload.

In short, we consider a request to be dependent on the *last* completed request, and we issue a request only after its parent request has completed. The results in this paper are based on an issue window of 64 requests. A request within this window is issued when the request on which it is dependent completes, and the think time has elapsed. Inferring the dependencies based on the last completed request is the best we can do given the block level traces we have. The interested reader is referred to [18] for alternatives in special cases where the workloads can be described and replayed at a more logical level. For multiprocessing workloads, the dependence relationship should be maintained on a per process basis but unfortunately process information is typically not available in I/O traces. To try to account for such workloads, multiple traces can be merged to form a workload with several independent streams of I/O, each obeying the dependence relationship described above.

4.2 Workloads and Traces

The traces analyzed in this study were collected from both server and PC systems running real user workloads on three different platforms – Windows NT, IBM AIX and HP-UX. All of them were collected downstream of the database buffer pool and the file system cache, *i.e.,* these are physical address traces, and all hits to the I/O caches in the host system have been removed. The PC traces were collected with VTrace [30], a software tracing tool for Intel x86 PCs running Windows NT/2000. In this study, we are primarily interested in the disk activities, which are collected by VTrace through the use of device filters. We have verified the disk activity collected by VTrace with the raw traffic observed by a bus (SCSI) analyzer. Both the IBM AIX and HP-UX traces were collected using kernel-level trace facilities built into the respective operating systems. Most of the traces were gathered over periods of several months but to keep the simulation time manageable, we use only the first 45 days of the traces of which the first 20 days are used to warm up the simulator.

The PC traces were collected on the primary computer systems of a wide-variety of users, including engineers, graduate students, a secretary and several people in senior managerial positions. By having a wide variety of users in our sample, we believe that our traces

Design-ation	User Type	System Configuration					Trace Characteristics				
		System	Memory (MB)	File Systems	Storage Used[i] (GB)	# Disks	Duration	Footprint[ii] (GB)	Traffic (GB)	Requests (10^6)	R/W Ratio
P1	Engineer	333MHz P6	64	1GB FAT[i] 5GB NTFS[i]	6	1	45 days (7/26/99 - 9/8/99)	0.945	17.1	1.88	2.51
P2	Engineer	200MHz P6	64	1.2, 2.4, 1.2GB FAT	4.8	2	39 days (7/26/99 - 9/2/99)	0.509	9.45	1.15	1.37
P3	Engineer	450MHz P6	128	4, 2GB NTFS	6	1	45 days (7/26/99 - 9/8/99)	0.708	5.01	0.679	0.429
P4	Engineer	450MHz P6	128	3, 3GB NTFS	6	1	29 days (7/27/99 - 8/24/99)	4.72	26.6	2.56	0.606
P5	Engineer	450MHz P6	128	3.9, 2.1GB NTFS	6	1	45 days (7/26/99 - 9/8/99)	2.66	31.5	4.04	0.338
P6	Manager	166MHz P6	128	3, 2GB NTFS	5	2	45 days (7/23/99 - 9/5/99)	0.513	2.43	0.324	0.147
P7	Engineer	266MHz P6	192	4GB NTFS	4	1	45 days (7/26/99 - 9/8/99)	1.84	20.1	2.27	0.288
P8	Secretary	300MHz P5	64	1, 3GB NTFS	4	1	45 days (7/27/99 - 9/9/99)	0.519	9.52	1.15	1.23
P9	Engineer	166MHz P5	80	1.5, 1.5GB NTFS	3	2	32 days (7/23/99 - 8/23/99)	0.848	9.93	1.42	0.925
P10	CTO	266MHz P6	96	4.2GB NTFS	4.2	1	45 days (1/20/00 – 3/4/00)	2.58	16.3	1.75	0.937
P11	Director	350MHz P6	64	2, 2GB NTFS	4	1	45 days (8/25/99 – 10/8/99)	0.73	11.4	1.58	0.831
P12	Director	400MHz P6	128	2, 4GB NTFS	6	1	45 days (9/10/99 – 10/24/99)	1.36	6.2	0.514	0.758
P13	Grad. Student	200MHz P6	128	1, 1, 2GB NTFS	4	2	45 days (10/22/99 – 12/5/99)	0.442	6.62	1.13	0.566
P14	Grad. Student	450MHz P6	128	2, 2, 2, 2GB NTFS	8	3	45 days (8/30/99 – 10/13/99)	3.92	22.3	2.9	0.481
P-Avg.	-	318MHz	109	-	5.07	1.43	41.2 days	1.59	13.9	1.67	0.816

(a) Personal Systems.

Design-ation	Primary Function	System Configuration					Trace Characteristics				
		System	Memory (MB)	File Systems	Storage Used[i] (GB)	# Disks	Duration	Footprint[ii] (GB)	Traffic (GB)	Requests (10^6)	R/W Ratio
FS1	File Server (NFS[iii])	HP 9000/720 (50MHz)	32	3 BSD[iii] FFS[iii] (3 GB)	3	3	45 days (4/25/92 - 6/8/92)	1.39	63	9.78	0.718
TS1	Time-Sharing System	HP 9000/877 (64MHz)	96	12 BSD FFS (10.4GB)	10.4	8	45 days (4/18/92 - 6/1/92)	4.75	123	20	0.794
DS1	Database Server (ERP[iii])	IBM RS/6000 R30 SMP[iii] (4X 75MHz)	768	8 AIX JFS (9GB), 3 paging (1.4GB), 30 raw database partitions (42GB)	52.4	13	7 days (8/13/96 – 8/19/96)	6.52	37.7	6.64	0.607
S-Avg.	-	-	299	-	18.5	8	32.3 days	4.22	74.6	12.1	0.706

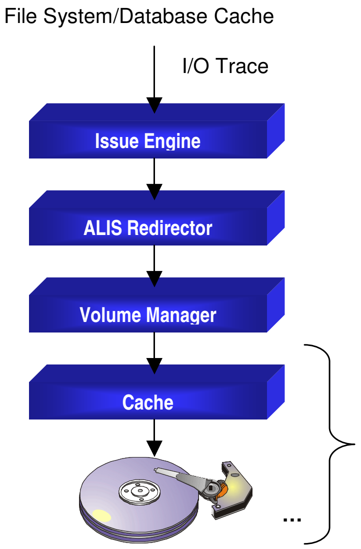
[i] Sum of all the file systems and allocated volumes.
[ii] Amount of data referenced at least once
[iii] AFS – Andrew Filesystem, AIX – Advanced Interactive Executive (IBM's flavor of UNIX), BSD – Berkeley System Development Unix, ERP – Enterprise Resource Planning, FFS – Fast Filesystem, JFS – Journal Filesystem, NFS – Network Filesystem, NTFS – NT Filesystem, SMP – Symmetric Multiprocessor

(b) Servers.

Table 1: Trace Description.

are illustrative of the PC workloads in many offices, especially those involved in research and development. Note, however, that the traces should not be taken as typical or representative of any other system or environment. Despite this disclaimer, the fact that many of their characteristics correspond to those obtained previously (see [17]), albeit in somewhat different environments, suggest that our findings are to a large extent generalizable. Table 1(a) summarizes the characteristics of these traces. We denote the PC traces as P1, P2, ..., P14 and the arithmetic mean of their results as P-Avg. As detailed in [17], the PC traces contain only I/Os that occur when the user is actively interacting with the system. Specifically, we consider the system to be idle from ten minutes after the last user keyboard or mouse activity until the next such user action, and we assume that there is no I/O activity during the idle periods. We believe that this is a reasonable approximation in the PC environment, although it is possible that we are ignoring some level of activity due to periodic system tasks such as daemons. This latter type of activity should have a negligible effect on the I/O load, and is not likely to be noticed by the user.

The servers traced include a file server, a time-sharing system and a database server. The characteristics of these traces are summarized in Table 1(b). Throughout this paper, we use the term S-Avg. to denote the arithmetic mean of the results for the server workloads. The file server trace (FS1) was taken off a file server for nine clients at the University of California, Berkeley. This system was primarily used for compilation and editing. It is referred to as Snake in [44]. The



	Resource-Poor	Resource-Rich
Read Caching	8MB per disk, Least-Recently-Used (LRU) replacement.	1% of storage used, Least-Recently-Used (LRU) replacement.
Prefetching	32KB read-ahead. Preemptible read-ahead up to maximum prefetch of 128KB, read any free blocks.	Conditional sequential prefetch, 16KB segments for PC workloads, 8KB segments for server workloads, prefetch trigger of 1, prefetch factor of 2. Preemptible read-ahead up to maximum prefetch of 128KB, read any free blocks, 8MB per disk opportunistic prefetch buffer.
Write Buffering	4MB per disk, lowMark = 0.2, highMark = 0.8, Least-Recently-Written (LRW) replacement, 30s age limit.	0.1% of storage used, lowMark = 0.2, highMark = 0.8, Least-Recently-Written (LRW) replacement, 1 hour age limit.
Request Scheduling	Shortest Access Time First with age factor = 0.01 (ASATF(0.01)), queue depth of 8.	Shortest Access Time First with age factor = 0.01 (ASATF(0.01)), queue depth of 8.
Striping	Stripe unit of 2MB.	Stripe unit of 2MB.

Figure 4: Block Diagram of Simulation Model Showing the Optimized Parameters for the Underlying Storage System.

trace denoted TS1 was gathered on a time-sharing system at an industrial research laboratory. It was mainly used for news, mail, text editing, simulation and compilation. It is referred to as Cello in [44]. The database server trace (DS1) was collected at one of the largest health insurers nationwide. The system traced was running an Enterprise Resource Planning (ERP) application on top of a commercial database system. This trace is only seven days long and the first three days are used to warm up the simulator. More details about the traces and how they were collected can be found in [17].

In addition to these base workloads, we scale up the traces to obtain workloads that are more intense. Results reported in [17] show that for the PC workloads, the processor utilization (in %) during the intervals between the issuance of an I/O and the last I/O completion is related to the length of the interval by a function of the form $f(x) = 1/(ax + b)$ where $a = 0.0857$ and $b = 0.0105$. To model a processor that is n times faster than was in the traced system, we would scale only the system processing time by n, leaving the user portion of the think time unchanged. Specifically, we would replace an interval of length x by one of length $x[1 - f(x) + f(x)/n]$. In this paper, we run each workload preserving the original think time. For the PC workloads, we also evaluate what happens in the limit when systems are infinitely fast, *i.e.,* we replace an interval of length x by one of $x[1 - f(x)]$. We denote these sped-up PC workloads as P1s, P2s, ..., P14s and the arithmetic mean of their results as Ps-Avg.

We also merge ten of the longest PC traces to obtain a workload with ten independent streams of I/O, each of which obeys the dependence relationship discussed earlier. We refer to this merged trace as Pm. The volume of I/O traffic in this merged PC workload is similar to that of a server supporting multiple PCs. Its locality characteristics are, however, different because there is no sharing of data among the different users so that if two users are both using the same application, they end up using different copies of the application. Pm might be construed as the workload of a system on which multiple independent PC workloads are consolidated. For the server workloads, we merge the FS1 and TS1 traces to obtain Sm. In this paper, we often use the term *PC workloads* to refer collectively to the base PC workloads, the sped-up PC workloads and the merged PC workload. The term *server workloads* likewise refers to the base server workloads and the merged server workload. Note that neither speeding up the system processing time nor merging multiple traces are perfect methods for scaling up the workloads but we believe that they are more realistic than simply scaling the I/O inter-arrival time, as is commonly done.

4.3 Simulation Model

The major components of our simulation model are presented in Figure 4. Our simulator is written in C++ using the CSIM simulation library [37]. It is based upon a detailed model of the mechanical components of the IBM Ultrastar 73LZX [24] family of disks that is used in disk development and that has been validated against test measurements obtained on several batches of the disk. More details about our simulator are available in [18]. The IBM Ultrastar 73LZX [24] family of 10K RPM disks consists of four members with storage capacities of 9.1 GB, 18.3 GB, 36.7 GB and 73.4 GB. The average seek time is specified to be 4.9 ms and the data rate varies from 29.2 to 57.0 MB/s. When we evaluate the effectiveness of ALIS as disk technology improves, we scale the characteristics of the disk according to technology trends which we derive by analyzing the specifications of disks introduced over the last ten years [18].

A wide range of techniques such as caching, prefetching, write buffering, request scheduling and striping have been invented for optimizing I/O performance. Each of these optimizations can be configured with different policies and parameters, resulting in a huge design space for the storage system. In our earlier work [18], we systematically explore the entire design space to establish the effectiveness of the various techniques for real workloads, and to determine the best practices for each technique. Here, we leverage our previous results and use the optimal settings we derived to set up the baseline storage system for evaluating the effectiveness of ALIS. As in [18], we consider two reasonable configurations the parameters of which are summarized in Figure 4.

As its name suggests, the resource-rich configuration represents an environment in which resources in the storage system are plentiful, as may be the case when there is a large outboard controller. In the resource-rich environment, the storage system has a very large cache that is sized at 1% of the storage used. It performs sequential prefetch into the cache by conditioning on the length of the sequential pattern already observed. In addition, the disks perform opportunistic prefetch (preemptible read-ahead and read any free blocks). The resource-poor environment mimics a situation where the storage system consists of only disks and low-end disk adaptors. Each disk has an 8 MB cache and performs sequential read-ahead and opportunistic prefetch. The parameter settings for these techniques are in Figure 4 and are the optimal values found in [18]. The interested reader is referred to [18] for more details about these configurations.

For workloads with multiple disk volumes, we concatenate the volumes to create a single address space. Each workload is fitted to the smallest disk from the IBM Ultrastar 73LZX [24] family that is bigger than the total volume size, leaving a headroom of 20% for the reorganized area. When we scale the capacity of the disk and require more than one disk to hold the data, we stripe the data using the previously determined stripe size of 2 MB [18]. We do not simulate RAID error correction since it is for the most part orthogonal to ALIS.

Later in Section 6.4, we will show that infrequent (daily to weekly) block reorganization is sufficient to realize most of the benefit of ALIS. Given our prior result that there is a lot of time during which the storage system is relatively idle [17], we make the simplifying assumption in this study that the block reorganization can be performed instantaneously. In [22], we validate this assumption by showing that the process of physically copying blocks into the reorganized region takes up only a small fraction of the idle time available between reorganizations.

4.4 Performance Metrics

I/O performance can be measured at different levels in the storage hierarchy. In order to fully quantify the effect of ALIS, we measure performance from when requests are issued to the storage system, before they are potentially broken up by the ALIS redirector or the volume manager for requests that span multiple disks. The two important metrics in I/O performance are *response time* and *throughput*. Response time includes both the time needed to service the request and the time spent waiting or queueing for service. Throughput is the maximum number of I/Os that can be handled per second by the system. Quantifying the throughput is generally difficult with trace-driven simulation because the workload, as recorded in the trace, is constant. We can try to scale or speed up the workload to determine the maximum workload the system can sustain but this is difficult to achieve in a realistic manner.

In this study, we estimate the throughput by considering the amount of critical resource each I/O consumes. Specifically, we look at the average amount of time the disk arm is busy per request, deeming the disk arm to be busy both when it is being moved into position to service a request and when it has to be kept in position to transfer data. We refer to this metric as the *service time*. Throughput can be approximated by taking the reciprocal of the average service time. One thing to bear in mind is that there are opportunistic techniques, especially for reads (*e.g.,* preemptible read-ahead), that can be used to improve performance. The service time does not include the otherwise idle time that the opportunistic techniques exploit. Thus the service time of a lightly loaded disk will tend to be optimistic of its maximum throughput.

Recall that the main benefit of ALIS is to transform the request stream so as to increase the effectiveness of sequential prefetch performed by the storage system downstream. To gain insight into this effect, we also examine the read *miss ratio* of the cache (and prefetch buffer) in the storage system. The read miss ratio is defined as the fraction of read requests that are not satisfied by the cache (or prefetch buffer), or in other words, the fraction of requests that requires physical I/O. It should take on a lower value when sequential prefetch is more effective.

	Resource-Poor				Resource-Rich			
	Average Read Response Time (ms)	Average Read Service Time (ms)	Read Miss Ratio	Average Write Service Time (ms)	Average Read Response Time (ms)	Average Read Service Time (ms)	Read Miss Ratio	Average Write Service Time (ms)
P-Avg.	3.34	2.22	0.431	1.41	2.66	1.79	0.313	0.700
S-Avg.	2.67	1.91	0.349	1.32	1.20	0.886	0.167	0.535
Ps-Avg.	3.83	2.18	0.450	1.05	3.15	1.75	0.319	0.681
Pm	3.14	2.23	0.447	1.30	1.65	1.24	0.226	0.474
Sm	3.37	2.67	0.468	1.57	1.38	1.10	0.204	0.855

Table 2: Baseline Performance Figures.

Note that we tend to care more about read response time and less about write response time because write latency can often be effectively hidden by write buffering [18]. Write buffering can also dramatically improve write throughput by eliminating repeated writes [18], and because workloads tend to be bursty [17], the physical writes can generally be deferred until the system is relatively idle. Moreover, despite predictions to the contrary (*e.g.,* [38]), both measurements of real systems (*e.g.,* [2]) and simulation studies (*e.g.,* [7, 18, 21]) show that large caches, while effective, have not eliminated read traffic. For instance, having a very large cache sized at 1% of the storage used only reduces the read-to-write ratio from about 0.82 to 0.57 for our PC workloads, and from about 0.62 to 0.43 for our server workloads. Therefore in this paper, we focus primarily on the read response time, read service time, read miss ratio, and to a lesser extent, on the write service time. In particular, we look at how these metrics are improved with ALIS where improvement is defined as $(value_{old} - value_{new})/value_{old}$ if a smaller value is better and $(value_{new} - value_{old})/value_{old}$ otherwise. We use the performance figures obtained previously in [18] as the baseline numbers. These are summarized in Table 2.

5 Clustering Strategies

In this section, we present in detail various techniques for deciding which blocks to reorganize and how to lay these blocks out relative to one another. We refer to these techniques collectively as clustering strategies. We will use empirical performance data to motivate the various strategies and to substantiate our design and parameter choices. General performance analysis of the system will appear in the next section.

5.1 Heat Clustering

In our earlier work, we observed that only a relatively small fraction of the data stored on disk is in active use [17]. The rest of the data are simply there, presumably because disk storage is the first stable or non-volatile level in the memory hierarchy, and the only stable level that offers online convenience. Given the exponential increase in disk capacity, the tendency is to be less careful about how disk space is used so that data will be increasingly stored on disk just in case they will be needed. Figure 5 depicts the typical situation in a storage system. Each square in the figure represents a block of data and the darkness of the square reflects the frequency of access to that block of data. The squares are arranged in the sequence of the corresponding block addresses, from left to right and top to bottom. There are some hot or frequently accessed blocks and these are distributed throughout the storage system. Accessing such active data requires the disk arm to seek across a lot of inactive or cold data, which is clearly not the most efficient arrangement.

This observation suggests that we should try to determine the active blocks and cluster them together so that they can be accessed more efficiently with less physical movement. As discussed in Section 2, over the last two decades, there have been several attempts to improve spatial locality in storage systems by clustering together hot data [1, 3, 43, 48, 49, 53, 54]. We refer to these schemes collectively as *heat clustering*. The basic approach is to count the number of requests directed to each unit of reorganization over a period of time, and to use the counts to identify the frequently accessed data and to rearrange them using a variety of block layouts. For the most part however, the previously proposed block layouts fail to effectively account for the fact that real workloads have predictable (and often sequential) access patterns, and that there is a dramatic and increasing difference between random and sequential disk performance.

5.1.1 Organ Pipe and Heat Placement

For instance, previous work relied almost exclusively on the *organ pipe block layout* [27] in which the most frequently accessed data is placed at the center

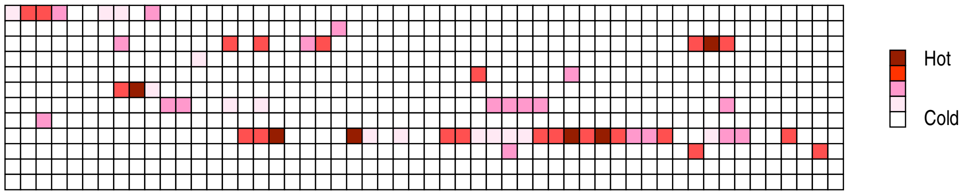


Figure 5: Typical Unoptimized Block Layout.

of the reorganized area, the next most frequently accessed data is placed on either side of the center, and the process continues until the least-accessed data has been placed at the edges of the reorganized region. If we visualize the block access frequencies in the resulting arrangement, we get an image of organ pipes, hence the name.

Considering the disk as a 1-dimensional space, the organ pipe heuristic minimizes disk head movement under the conditions that data is demand fetched, and that the references are derived from an independent random process with time invariant distribution and are handled on a first-come-first served basis [27]. However, disks are really 2-dimensional in nature, and the cost of moving the head is not an increasing function of the distance moved. A small backward movement, for example, requires almost an entire disk revolution. Furthermore, storage systems today perform aggressive prefetch and request scheduling, and in practice data references are not independent nor are they drawn from a fixed distribution. See for instance [20, 21, 45, 47] where real workloads are shown to clearly generate dependent references. In other words, the organ pipe arrangement is not optimal in practice.

In Figures 6(a) and A-1(a), we present the performance effect of heat clustering with the organ pipe layout on our various workloads. These figures assume that reorganization is performed daily and that the reorganized area is 10% the size of the total volume and is located at a byte offset 30% into the volume with the volume laid out inwards from the outer edge of the disk (sensitivity to these parameters is evaluated in Section 6). Observe that the organ pipe heuristic performs very poorly. As can be seen by the degradation in read miss ratio in Figures A-2(a) and A-3(a), the organ pipe block layout ends up splitting related data, thereby rendering sequential prefetch ineffective and causing some requests to require multiple I/Os. This is especially the case when the unit of reorganization is small, as was recommended previously (*e.g.,* [1]) in order to cluster hot data as closely as possible. With larger reorganization units, the performance degradation is reduced because spatial locality is preserved within the reorganization unit. In the limit, the organ pipe layout converges to the original block layout and achieves performance parity with the unreorganized case.

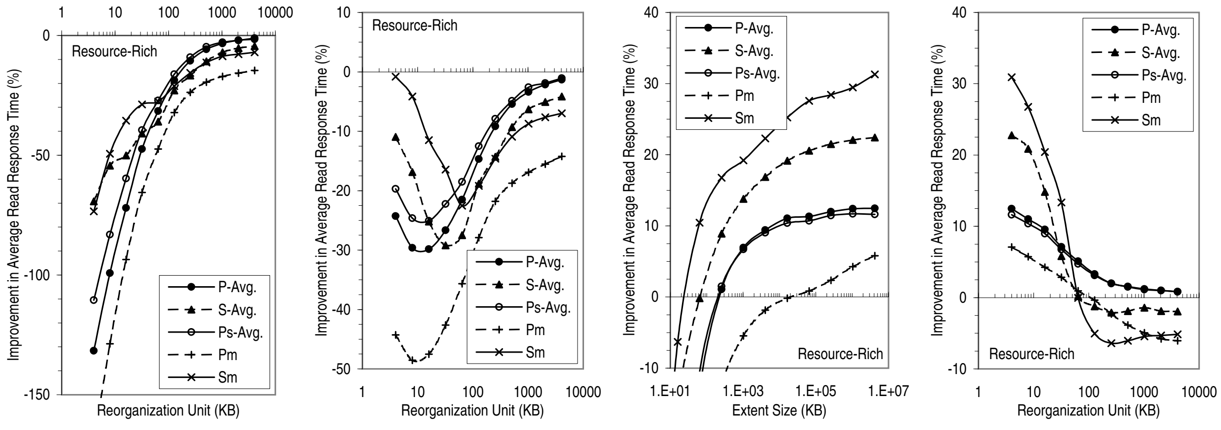
To prevent the disk arm from having to seek back and forth across the center of the organ pipe arrangement, an alternative is to arrange the hot reorganization units in decreasing order of their heat or frequency counts. In such an arrangement which we call the *heat layout*, the most frequently accessed reorganization unit is placed first, followed in order by the next most frequently accessed unit, and so on. As shown in Figure A-6, the heat layout, while better than the organ pipe layout, still degrades performance substantially for all but the largest reorganization units.

5.1.2 Link Closure Placement

An early study did identify the problem that when sequentially accessed data is split on either side of the organ pipe arrangement, the disk arm has to seek back and forth across the disk resulting in decreased performance [43]. A technique of maintaining forward and backward pointers from each reorganization unit to the reorganization unit most likely to precede and succeed it was proposed. This scheme is similar to the heat layout except that when a reorganization unit is placed, the *link closure* formed by following the forward and backward pointers is placed at the same time. As shown in Figures 6(b) and A-1(b), this scheme, though better than the pure organ pipe heuristic, still suffers from the same problems because it is not accurate enough at identifying data that tend to be accessed in sequence.

5.1.3 Packed Extents and Sequential Layout

Since only a portion of the stored data is in active use, clustering the hot data should yield a substantial improvement in performance. The key to realizing this potential improvement is to recognize that there is some existing spatial locality in the reference stream, especially since the original block layout is the result of many optimizations (see Section 2). When clustering the hot data, we should therefore attempt to preserve the original block sequence, particularly when there is aggressive read-ahead.



(a) Organ Pipe. (b) Link Closure. (c) Packed Extents. (d) Sequential.

Figure 6: Effectiveness of Various Heat Clustering Block Layouts at Improving Average Read Response Time (Resource-Rich).

With this insight, we develop a block layout strategy called *packed extents*. As before, we keep a count of the number of requests directed to each unit of reorganization over a period of time. During reorganization, we first identify the n "reorganization units" with the highest frequency count, where n is the number of reorganization units that can fit in the reorganized area. These are the target units, *i.e.,* the units that should be reorganized. Next, the storage space is divided into extents each the size of many reorganization units. These extents are sorted based on the highest access frequency of any reorganization unit within each extent. If there is a tie, the next highest access frequency is compared. Finally, the target reorganization units are arranged in the reorganized region in ascending order of their extent rank in the sorted extent list, and their offset within the extent. The packed extents layout essentially packs hot data together while preserving the sequence of the data within each extent, hence its name.

The main effect of the packed extents layout is to reduce seek distance without decreasing prefetch effectiveness. By moving data that are seldom read out of the way, it actually also improves the effectiveness of sequential prefetch, as can be seen by the reduction in the read miss ratio in Figures A-2(c) and A-3(c). In Figures 6(c) and A-1(c), we present the corresponding improvement in read response time. These figures assume a 4 KB reorganization unit. Observe that the packed extents layout performs very well for large extents, improving average read response time in the resource-rich environment by up to 12% and 31% for the PC and server workloads respectively. That the performance increases with extent sizes up to the gigabyte range implies that for laying out the hot reorganization units, preserving existing spatial locality is more important than concentrating the heat.

This observation suggests that simply arranging the target reorganization units in increasing order of their original block address should work well. Such a *sequential layout* is the special case of packed extents with a single extent. While straightforward, the sequential layout tends to be sensitive to the original block layout, especially to user/administrator actions such as the order in which workloads are migrated or loaded onto the storage system. But for our workloads, the sequential layout works well. Observe from Figures 6(d) and A-1(d) that with a reorganization unit of 4 KB, the average read response time is improved by up to 29% in the resource-poor environment and 31% in the resource-rich environment. We will therefore use the sequential layout for heat clustering in the rest of this paper. It turns out that the sequential layout was considered in [1] where it was reported to perform worse than the organ pipe layout. The reason was that the earlier work did not take into account the aggressive caching and prefetching common today, and was focused primarily on reducing the seek time.

To increase stability in the effectiveness of heat clustering, the reference counts can be aged such that

$$Count_{new} = \alpha Count_{current} + (1 - \alpha) Count_{old} \quad (1)$$

where $Count_{new}$ is the reference count used to drive the reorganization, $Count_{current}$ is the reference count collected since the last reorganization and $Count_{old}$ is the previous value of $Count_{new}$. The parameter α controls the relative weight placed on the current reference

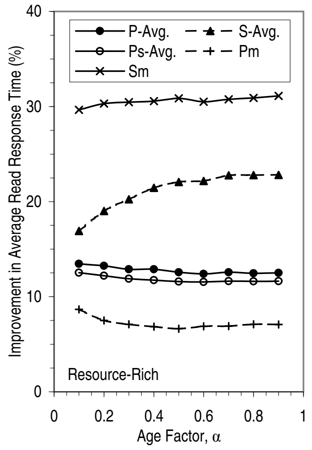


Figure 7: Sensitivity of Heat Clustering to Age Factor, α (Resource-Rich).

counts and those obtained in the past. For example, with an α value of 1, only the most recent reference counts are considered. In Figures 7 and A-7, we study how sensitive performance with the sequential layout is to the value of the parameter α. The PC workloads tend to perform slightly better for smaller values of α, meaning when more history is taken into account. The opposite is true, however, of the server workloads on average but both effects are small. As in [43], all the results in this paper assume a value of 0.8 for α unless otherwise indicated.

5.2 Run Clustering

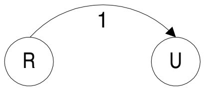
Our analysis of the various workloads also reveals that the reference stream contains long read sequences that are often repeated. The presence of such repeated read sequences or *runs* should not be surprising since computers are frequently asked to perform the same tasks over and over again. For instance, PC users tend to use a core set of applications, and each time the same application is launched, the same set of files [55] and blocks [25] are read. The existence of runs suggest a clustering strategy that seeks to identify these runs so as to lay them out sequentially (in the order they are referenced) in the reorganized area. We refer to this strategy as *run clustering*.

5.2.1 Representing Access Patterns

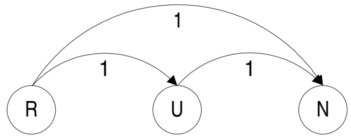
The reference stream contains a wealth of information. The first step in run clustering is to extract relevant details from the reference stream and to represent the extracted information compactly and in a manner that facilitates analysis. This is accomplished by building an access graph where each node or vertex represents a unit of reorganization and the weight of an edge from vertex i to vertex j represents the desirability for reorganization unit j to be located close to and after reorganization unit i. For example, a straightforward method for building the access graph is to set the weight of edge $i \to j$ equal to the number of times reorganization unit j is referenced *immediately after* unit i. But this method represents only pair-wise patterns. Moreover, at the storage level, any repeated pattern is likely to be interspersed by other requests because the reference stream is the aggregation of many independent streams of I/O, especially in multi-tasking and multi-user systems. Furthermore, the I/Os may not arrive at the storage system in the order they were issued because of request scheduling or prefetch. Therefore, the access graph should represent not only sequences that are exactly repeated but also those that are *largely* or *pseudo* repeated.

Such an access graph can be constructed by setting the weight of edge $i \to j$ equal to the number of times reorganization unit j is referenced *shortly after* accessing unit i, or more specifically the number of times unit j is referenced within some number of references, τ, of accessing unit i [31]. We refer to τ as the *context size*. As an example, Figure 8(a) illustrates the graph that represents a sequence of two requests where the first request is for reorganization unit R and the second is for unit U. In Figure 8(b), we show the graph when an additional request for unit N is received. The figures assume a context size of two. Therefore, edges are added from both unit R and unit U to unit N. Figure 8(c) further illustrates the graph after an additional four requests for data are received in the sequence R, X, U, N. The resulting graph has three edges of weight two among the other edges of weight one. These edges of higher weight highlight the largely repeated sequence R, U, N. This example shows that by choosing an appropriate value for the context size or τ, we can effectively avoid having intermingled references obscure the runs. We will investigate good values for τ for our workloads later.

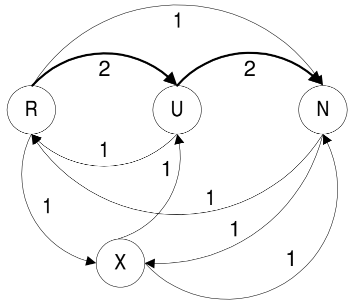
An undesirable consequence of having the edge weight represent the number of times a reorganization unit is referenced *within* τ references of accessing another unit is that we lose information about the exact sequence in which the references occur. For instance, Figure 9(a) depicts the graph for the reference string R, U, N, R, U, N. Observe that reorganization unit R is equally connected to unit U and unit N. The edge weights indicate that units U and N should be laid out after unit R but they do not tell the order in which these two units should be arranged. We could potentially figure out the order based on the edge of weight two from unit U to unit N. But to make it easier to find repeated sequences, the actual sequence of reference can be more accurately represented by employing a graduated edge



(a) Reference String: R, U.

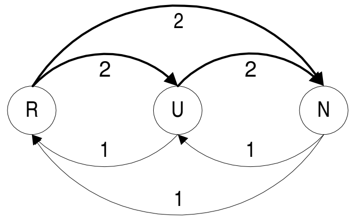


(b) Reference String: R, U, N.

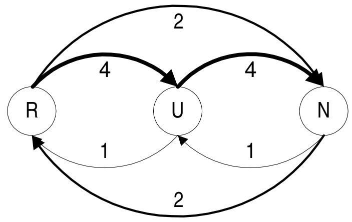


(c) Reference String: R, U, N, R, X, U, N.

Figure 8: Access Graph with a Context Size of Two.



(a) Uniform Edge Weight.



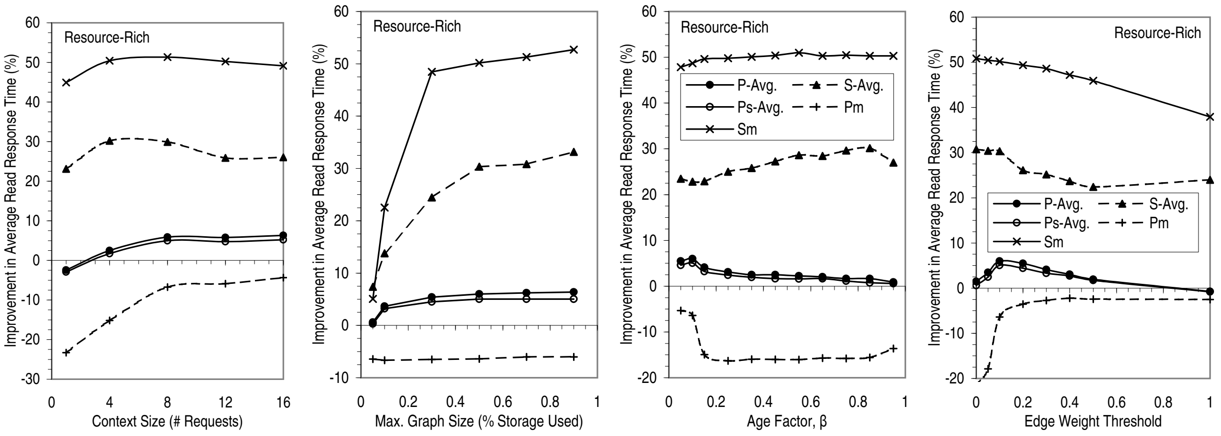
(b) Graduated Edge Weight.

Figure 9: The Effect of Graduated Edge Weight (Reference String = R,U,N,R,U,N, Context Size = 2).

weight scheme where the weight of edge $i \rightarrow j$ is a decreasing function of the number of references between when those two data units are referenced. For instance, suppose X_i denotes the reorganization unit referenced by the i-th read. For each X_n, we add an edge of weight $\tau - j + 1$ from X_{n-j} to X_n, where $j \leq \tau$. In the example of Figure 8(b), we would add an edge of weight *one* from unit R to unit N and an edge of weight *two* from unit U to unit N. Figure 9(b) shows that with such a graduated edge weight scheme, we can readily tell that unit R should be immediately followed by unit U when the reference string is R, U, N, R, U, N.

More generally, we can use the edge weight to carry two pieces of information – the number of times a reorganization unit is accessed within τ references of another, and the distance or number of intermediate references between when these two units are accessed. Suppose f is a parameter that determines the fraction of edge weight devoted to representing the distance information. Then for each X_n, we add an edge of weight $1 - f + f * \frac{\tau - j + 1}{\tau}$ from X_{n-j} to X_n, where $j \leq \tau$. We experimented with varying the weighting of these two pieces of information and found that $f = 1$ tends to work better although the difference is small (Figure A-9). The effect of varying f is small because our run discovery algorithm (to be discussed later) is able to determine the correct reference sequence most of the time, even without the graduated edge weights.

In Figures 10(a) and A-8(a), we analyze the effect of different context sizes on our various workloads. The context size should generally be large enough to allow pseudo repeated reference patterns to be effectively distinguished. For our workloads, performance clearly increases with the context size and tends to stabilize beyond about eight. Unless otherwise indicated, all the results in this paper assume a context size of nine. Note that the reorganization units can be of a fixed size but the results presented in Figure A-10 suggest that this is not very effective. In ALIS, each reorganization unit represents the data referenced in a request. By using such variable-sized reorganization units, we are able to achieve much better performance since the likelihood for a request to be split into multiple I/Os is reduced. Prefetch effectiveness is also enhanced because any discovered sequence is likely to contain only the data that is actually referenced. In addition, if a data block occurs in different request units, the same block could appear in multiple runs so that there are multiple copies of the block in the reorganized area, and this could help to distinguish different access sequences that include the same block.



(a) Context Size, τ.

(b) Graph Size.

(c) Age Factor, β.

(d) Edge Threshold.

Figure 10: Sensitivity of Run Clustering to Various Parameters (Resource-Rich).

Various pruning algorithms can be used to limit the size of the graph. We model the graph size by the number of vertices and edges in the graph. Each vertex requires the following fields - vertex ID (6 bytes), a pair of adjacency list pointers (2x4 bytes), pointer to next vertex (4 bytes), status byte (1 byte). Note that the graph is directed so we need a pair of adjacency lists for each vertex to be able to quickly determine both the incoming and the outgoing edges. Accordingly, there are two adjacency list entries (an outgoing and an incoming) for each edge. Each of these entries consists of the following fields – pointer to vertex (4 bytes), weight (2 bytes), pointer to next edge (4 bytes). Therefore each vertex in the graph requires 19 bytes of memory while each edge occupies 20 bytes. Whenever the graph size exceeds a predetermined maximum, we remove the vertices and edges with weight below some threshold which we set at the respective 10th percentile. In other words, we remove vertices weighing less than 90% of all the vertices and edges with weight in *aren't you saying the same thing here in 2 different ways?* the bottom 10% of all the edges. The weight of a vertex is defined as the weight of its heaviest edge. This simple *bottom pruning* policy adds no additional memory overhead and preserves the relative connectedness of the vertices so that the algorithm for discovering the runs is not confused when it tries to determine how to sequence the reorganization units.

Figures 10(b) and A-8(b) show the performance improvement that can be achieved as a function of the size of the graph. Observe from the figures that a graph smaller than 0.5% of the storage used is sufficient to realize most of the benefit of run clustering. This is the default graph size we use for the simulations reported in this paper. Memory of this size should be available when the storage system is relatively idle because caches larger than this are needed to effectively hold the working set [18]. A multiple-pass run clustering algorithm can be used to further reduce memory requirements. Note that high-end storage systems today host many terabytes of storage but the storage is used for different workloads and is partitioned into logical subsystems or volumes. These volumes can be individually reorganized so that the peak memory requirement for run clustering is greatly reduced from 0.5% of the total storage in use.

An idea for speeding up the graph build process and reducing the graph size is to pre-filter the reference stream to remove requests that do not occur frequently. The basic idea is to keep reference counts for each reorganization unit as in the case of heat clustering. In building the access graph, we ignore all the requests whose reference count falls below some threshold. Our experiments show that pre-filtering tends to reduce the performance gain (Figure A-11) because it is, for the most part, not as accurate as the graph pruning process in removing uninteresting information. But in cases where we are constrained by the graph build time, it could be a worthwhile option to pursue.

As in heat clustering, we age the edge weights to provide some continuity and avoid any dramatic fluctuations in performance. Specifically, we set

$$Weight_{new} = \beta Weight_{current} + (1-\beta) Weight_{old} \quad (2)$$

where $Weight_{new}$ is the edge weight used in the reorganization, $Weight_{current}$ is the edge weight collected since the last reorganization and $Weight_{old}$ is the previous value of $Weight_{new}$. The parameter β controls the relative weight placed on the current edge weight and those obtained in the past. In Figures 10(c) and A-8(c), we study how sensitive run clustering is to the value of the parameter β. Observe that as in heat clustering, the workloads are relatively stable over a wide range of β values with the PC workloads performing better for smaller values of β, meaning when more history is taken into account, and the server workloads preferring larger values of β. Such results reflect that fact that the PC workloads are less intense and have reference patterns that are repeated less frequently so that it is useful to look further back into history to find these patterns. This is especially the case for the merged PC workloads where the reference pattern of a given user can quickly become aged out before it is encountered again.

Throughout the design of ALIS, we try to desensitize its performance to the various parameters so that it is not catastrophic for somebody to "configure the system wrongly". To reflect the likely situation that ALIS will be used with a default setting, we base our performance evaluation on parameter settings that are good for an entire class of workloads rather than on the best values for each individual workload. Therefore, the results in this paper assume a default β value of 0.1 for all the PC workloads and 0.8 for all the server workloads. A useful piece of future work would be to devise ways to set the various parameters dynamically to adapt to each individual workload. Figures 10(c) and A-8(c) suggest that the approach of using hill-climbing to gradually adjust the value of β until a local optimum is reached should be very effective because the local optimum is also the global optimum. This characteristic is generally true for the parameters in ALIS.

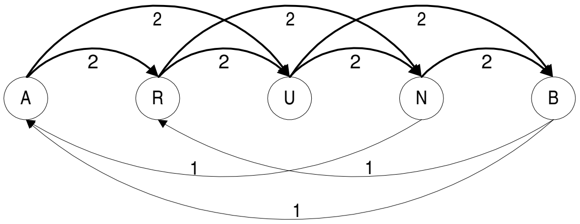
5.2.2 Mining Access Patterns

The second step in run clustering is to analyze the access graph to discover desirable sequences of reorganization units, which should correspond to the runs in the reference stream. This process is similar to the graph-theoretic clustering problem with an important twist that we are interested in the sequence of the vertices. Let G be an access graph built as outlined above and R, the target sequence or run. We use $|R|$ to denote the number of elements in R and $R[i]$, the ith element in R. By convention we refer to $R[1]$ as the front of R and $R[|R|]$ as the back of R. The following outlines the algorithm that ALIS uses to find R.

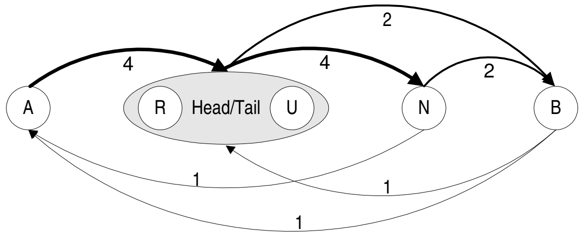
1. Find the heaviest edge linking two unmarked vertices.
2. Initialize R to the heaviest edge found and mark the two vertices.
3. Repeat
4. Find an unmarked vertex u such that $headweight = \sum_{i=1}^{Min(\tau,|R|)} Weight(u, R[i])$ is maximized.
5. Find an unmarked vertex v such that $tailweight = \sum_{i=1}^{Min(\tau,|R|)} Weight(R[|R| - i + 1], v)$ is maximized.
6. if $headweight > tailweight$
7. Mark u and add it to the front of R.
8. else
9. Mark v and add it to the back of R.
10. Goto Step 3.

In steps 1 and 2, we initialize the target run by the heaviest edge in the graph. Then in steps 3 to 10, we inductively grow the target run by adding a vertex at a time. In each iteration of the loop, we select the vertex that is most strongly connected to either the $head$ or $tail$ of the run, the $head$ or $tail$ of the run being, respectively, the first and last τ members of the run and τ is the context size used to build the access graph. Specifically, in step 4, we find a vertex u such that the weight of all its edges incident on the vertices in the $head$ is the highest. Vertex u represents the reorganization unit that we believe is most likely to immediately precede the target sequence in the reference stream. Therefore, if we decide to include it in the target sequence, we will add it as the first entry (Step 7). Similarly, in step 5, we find a vertex v that we believe is most likely to immediate follow the target run in the reference stream. The decision of which vertex u or v to include in the target run is made greedily. We simply select the unit that is most likely to immediate precede or follow the target sequence, *i.e.,* the vertex that is most strongly connected to the respective ends of the target sequence (Step 6). By selecting the next vertex to include in the target run based on its connectedness to the first or last τ members of the run, the algorithm is following the way the access graph is built using a context of τ references to recover the original reference sequence. The use of a context of τ references also allows the algorithm to distinguish between repeated patterns that share some common reorganization units.

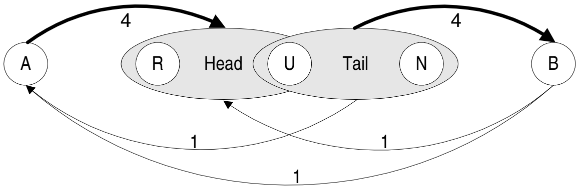
In Figure 11, we step through the operation of the algorithm on a graph for the reference string A, R, U, N, B, A, R, U, N, B. For simplicity, we assume that we use uniform edge weights and the context size is



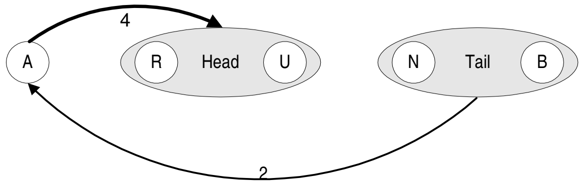
(a) Discovered Run = ϕ.



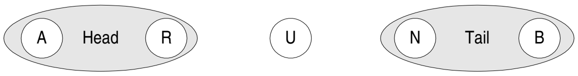
(b) Discovered Run = R, U.



(c) Discovered Run = R, U, N.



(d) Discovered Run = R, U, N, B.



(e) Discovered Run = A, R, U, N, B.

Figure 11: The Use of Context in Discovering the Next Vertex in a Run (Reference String = A, R, U, N, B, A, R, U, N, B, Context Size = 2).

two. Without loss in generality, suppose we pick the edge $R \rightarrow U$ in step 1. Since the target sequence has only two entries at this point, the $head$ and $tail$ of the sequence are identical and contain the units R and U (Figure 11(b)). By considering the edges of *both* unit R and unit U, the algorithm easily figures out that A is the unit most likely to immediate precede the sequence R, U while N is the unit most likely to immediately follow it. Note that looking at unit U alone, we would not be able tell whether N or B is the unit most likely to immediately follow the target sequence. To grow the target run, we can either add unit A to the front or unit N to the rear. Based on Step 6, we add unit N to the rear of the target sequence. Figure 11(c) shows the next iteration in the run discovery process where it becomes clear that $head$ refers to the first τ members of the target run while $tail$ refers to the last τ members.

The process of growing the target run continues until $headweight$ and $tailweight$ fall below some edge weight threshold. The edge weight threshold ensures that a sequence (*e.g.,* $u \rightarrow head$) becomes part of the run only if it occurs frequently. The threshold is therefore conveniently expressed relative to the value that $headweight$ or $tailweight$ would assume if the sequence were to occur only *once* in every reorganization interval. In Figures 10(d) and A-8(d), we investigate the effect of varying the edge weight threshold on the effectiveness of run clustering. Observe that being more selective in picking the vertices tends to reduce performance except at very small threshold values for the PC workloads. As we have noted earlier, the PC workloads tend to have less repetition and a lot more churn in their reference patterns so that it is necessary to filter out some of the noise and look further into the past to find repeated patterns. The server workloads are much easier to handle and respond well to run clustering even without filtering. The jagged nature of the plot for the average of the server workloads (S-Avg.) results from DS1, which being only seven days long is too short for the edge weights to be smoothed out. In this paper, we assume a default value of 0.1 for the edge weight threshold for all the workloads.

A variation of the run discovery algorithm is to terminate the target sequence whenever $headweight$ and $tailweight$ are much lower than (*e.g.,* less than half) their respective values in the previous iteration of the loop (steps 3-10). The idea is to prevent the algorithm from latching onto a run and pursuing it too far, or in other words, from going too deep down what could be a local minimum. Another variation of the algorithm is to add u to the target run only if the heaviest outgoing edge of u is to one of the vertices in $head$ and to add v to the run only if the heaviest incoming edge of v is from one of the vertices in $tail$. We experimented with both variations and found that they do not offer consistently better performance. As we shall see later, the workloads do change over time so that excessive optimization based on the past may not be productive.

The whole algorithm is executed repeatedly to find runs of decreasing desirability. In Figure A-12, we study whether it makes sense to impose a minimum run length. Runs that are shorter than the minimum are discarded. Recall that the context size is chosen to allow pseudo repeated patterns to be effectively distinguished, so the context size is intuitively the resolution limit of our run discovery algorithm. A useful run should therefore be at least as long as the context size. This turns out to agree with our experimental results. Note that the run discovery process is very efficient, requiring only $O(e \cdot log(e) + v)$ operations, where e is the number of edges and v, the number of vertices. The $e \cdot log(e)$ term results from sorting the edges once to facilitate step 1. Then each vertex is examined at most once.

Note that after a reorganization unit is added to a run, the run clustering algorithm marks it to prevent it from being included again in any run. An interesting variation of the algorithm is to allow multiple copies of a reorganization unit to exist either in the same run or in different runs. This is motivated by the fact that some data blocks, for instance those corresponding to shared libraries, may appear in more than one access pattern. The basic idea in this case is to not mark a vertex after it has been added to a run. Instead, we remove the edges that are used to include that particular vertex in the run. We experimented with this variation of the algorithm but found that it is not significantly better.

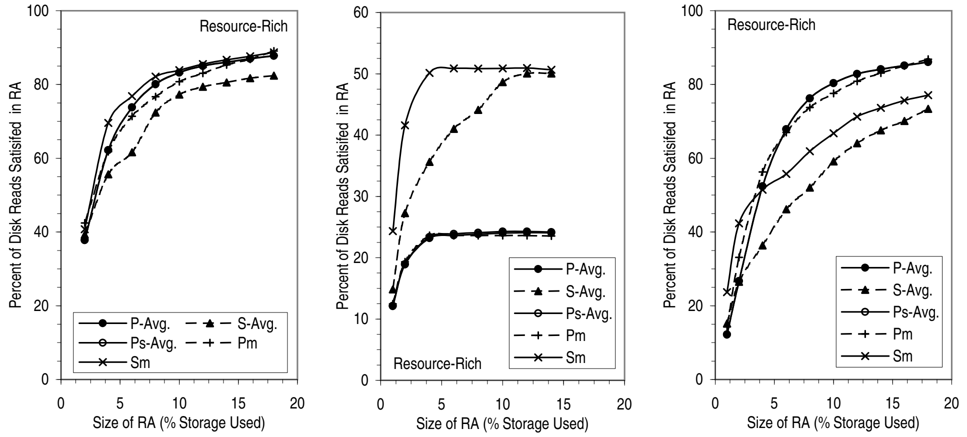
After the runs have been discovered and laid out in the reorganized area, the traffic redirector decides whether a given request should be serviced from the runs. This decision can be made by conditioning on the context or the recent reference history. Suppose that a request matches the kth reorganization unit in run R. We define the context match with R as the percentage of the previous τ requests that are in $R[k-\tau]$... $R[k-1]$. A request is redirected to R only if the context match with R exceeds some value. For our workloads, we find that it is generally better to always try to read from a run (Figure A-13). In the variation of the run discovery algorithm that allows a reorganization unit to appear in multiple runs, we use the context match to rank the matching runs.

5.3 Heat and Run Clustering Combined

In our analysis of run clustering, we observed that the improvement in read miss ratio significantly exceeds the improvement in read response and service time, especially for the PC workloads (*e.g.,* Figures 14 and A-17). It turns out that this is because many of the references cannot be clearly associated with any repeated sequence. For instance, Figures 12(b) and A-14(b) show that for the PC workloads, only about 20–30% of the disk read requests can be satisfied from the reorganized area with run clustering. Thus in practice, the disk head has to move back and forth between the runs in the reorganized area and the remaining hot spots in the home area. In other words, although the number of disk reads is reduced by run clustering, the time taken to service the remaining reads is lengthened. In the next section, we will study placing the reorganized region at different offsets into the volume. Ideally, we would like to locate the reorganized area near the remaining hot spots but these are typically distributed across the disk so that no single good location exists. Besides, figuring out where these hot spots are *a priori* is difficult. We believe a more promising approach is to try to satisfy more of the requests from the reorganized area. One way of accomplishing this is to simply perform heat clustering in addition to run clustering.

Since the runs are specific sequences of reference that are likely to be repeated, we assign higher priority to them. Specifically, on a read, we first attempt to satisfy the request by finding a matching run. If no such run exists, we try to read from the heat-clustered region before falling back to the home area. The reorganized area is shared between heat and run clustering, with the runs being allocated first. In this paper, all the results for heat and run clustering combined assume a default reorganized area that is 15% of the storage size and that is located at a byte offset 30% into the volume. We will investigate the performance sensitivity to these parameters in the next section. We also evaluated the idea of limiting the size of the reorganized area that is devoted to the runs but found that it did not make a significant difference (Figure A-15). Sharing the reorganized area dynamically between heat and run clustering works well in practice because a workload with many runs is not likely to gain much from the additional heat clustering while one with few runs will probably benefit a lot.

In Figures 12(c) and A-14(c), we plot the percent of disk reads that can be satisfied from the reorganized area when run clustering is augmented with heat clustering. Note that the number of disk reads is not constant across



(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure 12: Percent of Disk Reads Satisfied in Reorganized Area (Resource-Rich).

the different clustering algorithms. In particular, when run clustering is performed in addition to heat clustering, many of the disk reads that could be satisfied from the reorganized area are eliminated by the increased effectiveness of sequential prefetch. Therefore, the "hit" ratio of the reorganized area with heat and run clustering combined is somewhat less than with heat clustering alone. But it is still the case that the majority of disk reads can be satisfied from the reorganized area.

Such a result suggests that in the combined case, we can be more selective about what we consider to be part of a run because even if we are overly selective and miss some blocks, these blocks are likely to be found nearby in the adjacent heat-clustered region. We therefore reevaluate the edge weight threshold used in the run discovery algorithm. Figures 13 and A-16 summarize the results. Notice that compared to the case of run clustering alone (Figures 10(d) and A-8(d)), the plots are much more stable and the performance is less sensitive to the edge weight threshold, which is a nice property. As expected, the performance is better with larger threshold values when heat clustering is performed in addition to run clustering. Thus, we increase the default edge weight threshold for the PC workloads to 0.4 when heat and run clustering are combined.

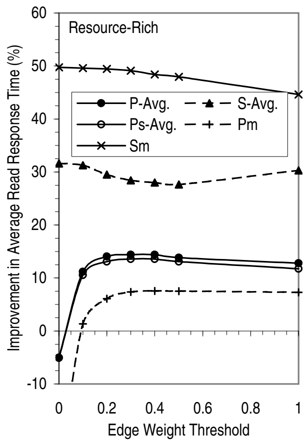


Figure 13: Sensitivity of Heat and Run Clustering Combined to Edge Threshold (Resource-Rich).

6 Performance Analysis

6.1 Clustering Algorithms

In Figures 14 and A-17, we summarize the performance improvement achieved by the various clustering schemes. In general, the PC workloads are improved less by ALIS than the server workloads. This could result from file system differences or the fact that the PC workloads, being more recent than the server workloads, have comparatively more caching upstream in the file system where a lot of the predictable (repeated) references are satisfied. Also, access patterns in the PC workloads tend to be more varied and to repeat much less frequently than in the server workloads because the repetitions likely result from the same user repeating the same task rather than many different users performing similar tasks. Moreover, PCs mostly run a fixed set of applications which are installed sequentially and which use large sequential data sets (*e.g.,* a Microsoft Word file is very large even for a short document). The in-

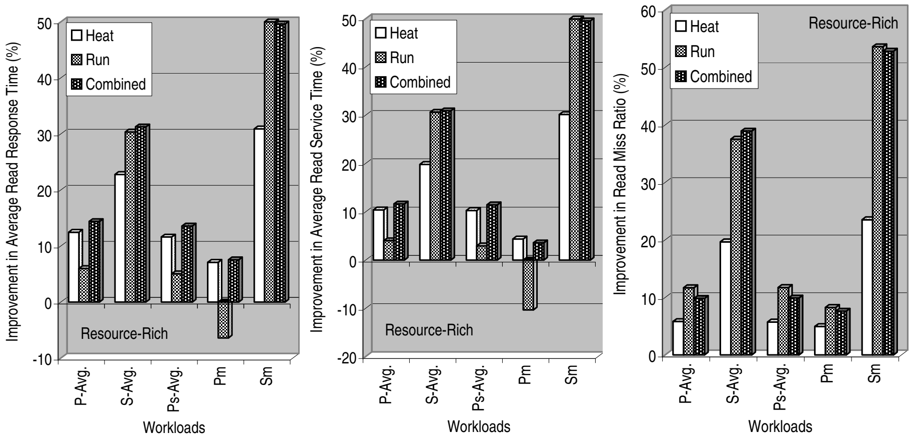


Figure 14: Performance Improvement with the Various Clustering Schemes (Resource-Rich).

creased availability of storage space in the more recent (PC) workloads further suggests reduced fragmentation which means less potential for ALIS to improve the block layout. Note, however, that most of the storage space in one of the server workloads, DS1, is updated in place and should have little fragmentation. Yet ALIS is able to improve the performance for this workload by a lot more than for the PC workloads.

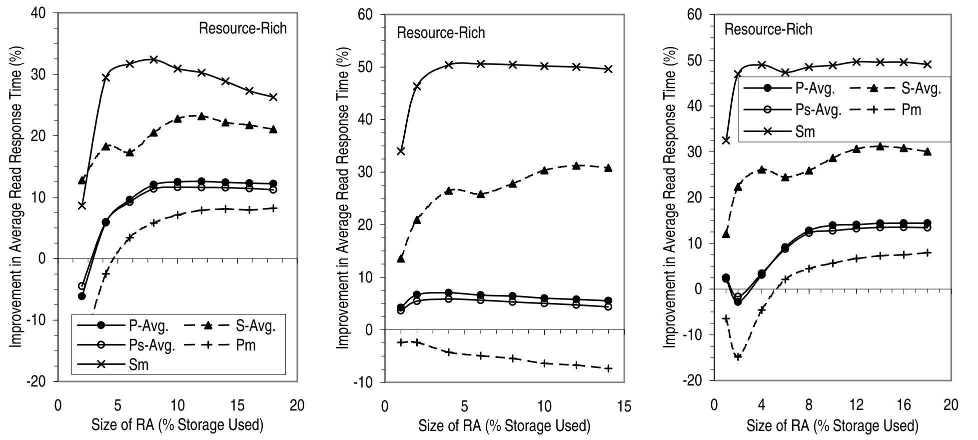
Observe further that combining heat and run clustering enables us to achieve the greater benefit of the two schemes. In fact, the performance of the combined scheme is clearly superior to either technique alone in practically all the cases. Specifically, the read response time for the PC workloads is improved on average by about 17% in the resource-poor environment and 14% in the resource-rich environment. The sped-up PC workloads are improved by about the same amount while the merged PC workload is improved by just under 10%. In general, the merged PC workload is difficult to optimize for because the repeated patterns, already few and far in between, are spread further apart than in the case of the base PC workloads. For the base server workloads on average, the improvement in read response time ranges from about 30% in the resource-rich environment to 37% in the resource-poor environment. The merged server workload is improved by as much as 50%. Interestingly, one of the PC users, P10, the chief technical officer, was running the disk defragmenter, Diskeeper [8]. Yet in both the resource-poor and resource-rich environments, run and heat clustering combined improves the read performance for this user by 15%, which is about the average for all the PC workloads.

6.2 Reorganized Area

Earlier in the paper, we said that reorganizing a small fraction of the stored data is enough for ALIS to achieve most of the potential performance improvement. In Figures 15 and A-18, we quantify what we mean by a small fraction. Observe that for all our workloads, a reorganized area less than 10% the size of the storage used is sufficient to realize practically all the benefit of heat clustering. For run clustering, the reorganized region required to get most of the advantage is even smaller. Combining heat and run clustering, we find that by reorganizing only about 10% of the storage space, we are able achieve most of the potential performance improvement for all the workloads except the server workloads which require on average about 15%. A storage overhead of 15% compares very favorably with other accepted techniques for increasing I/O performance such as mirroring in which the disk space is doubled. Given the technology trends, we believe that, in most cases, this 15% storage overhead is well worth the resulting increase in performance.

Note that performance does not increase monotonically with the size of the reorganized region, especially for small reorganized areas. In general, blocks are copied into the reorganized region and rearranged based on the prediction that the new arrangement will outperform the original layout. For some blocks, the prediction turns out to be wrong so that as more blocks are reorganized, the performance sometimes declines.

Besides the size of the reorganized region, another interesting question is where to locate it. If there is a constant number of sectors per cylinder and accesses are uniformly distributed, we would want to place the reorganized area at the center of the disk. However, modern disks are zoned so that the linear density, and hence the data rate, at the outer edge is a lot higher than at the inner edge. To leverage this higher sequential performance, the reorganized region should be placed closer to the outer edge. The complicating factor is that in practice, accesses are not uniformly distributed so that



(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure 15: Sensitivity to Size of Reorganized Area (Resource-Rich).

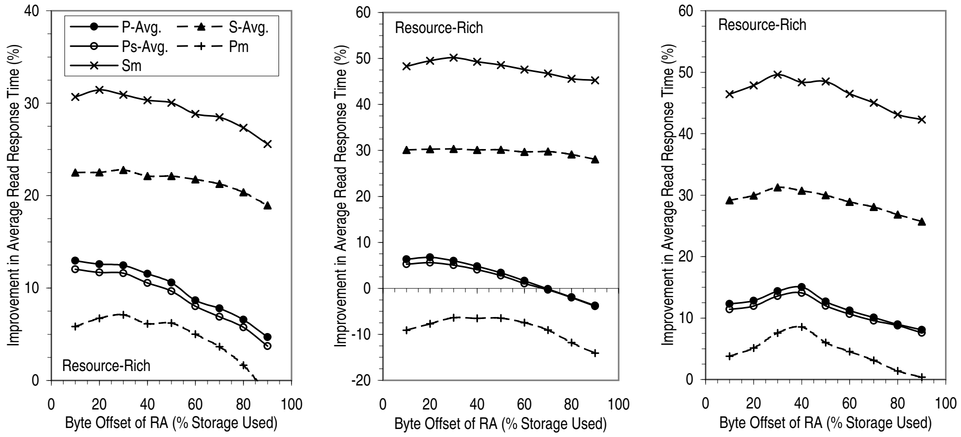
the optimal placement of the reorganized area depends on the workload characteristics. Specifically, it is advantageous to locate the reorganized area near to any remaining hot regions in the home area but determining these hot spots ahead of time is difficult. Besides, they are typically distributed across the disk so that no single good location exists.

In Figures 16 and A-19, we see these various effects at work in our workloads. We assume the typical situation where volumes are laid out inwards from the outer edge of the disk. As discussed earlier, we use, for each workload, the closest fitting member of the IBM Ultrastar 73LZX family of disks. This results in an average storage used to disk capacity ratio of about 65%. Recall that heat and run clustering combined has the nice property that most of the disk reads are either eliminated due to the more effective sequential prefetch, or can be satisfied from the reorganized area. Any remaining disk reads will tend to exhibit less locality and be spread out across the disk. In other words, the remaining disk reads will tend to be uniformly distributed. Therefore, in this case, we can predict that placing the reorganized area somewhere in the middle of the space used should minimize any disk head movement between the reorganized region and the home area. Empirically, we find that for all our workloads, locating the reorganized area at a byte offset 30-40% into the space used works well. Given that there are more sectors per track at the outer edge, this corresponds to roughly a 24-33% radial distance offset from the outer edge.

When data is replicated and reorganized, there may be multiple copies of a given block, and at reorganization times, the locations and numbers of copies may change. If blocks are cached by their physical addresses, the effectiveness of the cache could be affected. We studied the issue of whether changing the address of previously cached blocks to the address of one of the newly reorganized copies (if any) had a performance impact; we found that any such effect was minor [22]. The results presented in this paper assume that such remapping of cached blocks is performed after every reorganization.

6.3 Write Policy

In general, writes or update operations complicate a system and throttle its performance. For a system such as ALIS that replicates data, we have to ensure that all the copies of a block are either updated or invalidated whenever the block is written to. In our analysis of the workloads [17], we discover that blocks that are updated tend to be updated again rather than read. This suggests that we should update only one of the copies and invalidate the others. But the question remains of which copy to update. A second issue related to how writes are handled is whether the read access patterns differ from the write access patterns, and if it is possible to lay the blocks out to optimize for both. We know that the set of blocks that are both actively read and written tend to be small [17] so it is likely that read performance can be optimized without significantly degrading write performance. But should we try to optimize for both reads and writes by considering writes in addition to reads when tabulating the reference count and when building the access graph?



(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure 16: Sensitivity to Placement of Reorganized Area (Resource-Rich).

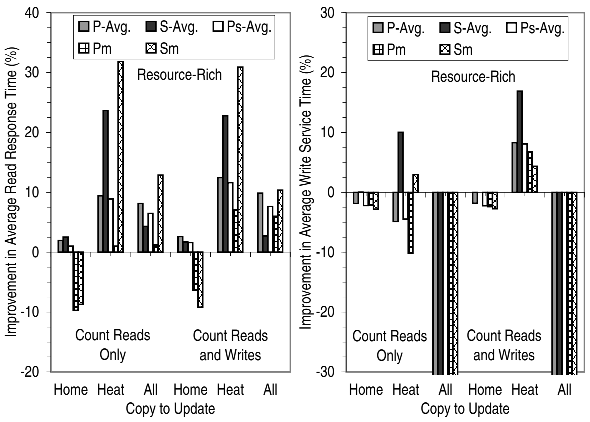
In Figures 17(a) and A-20(a), we show the performance effect of the different policies for handling writes. Observe that for heat clustering, updating the copy in the reorganized area offers the best read and write performance. Incorporating writes in the reference count speeds up the writes and in the case of the PC workloads, also improves the read performance. The results in this paper therefore assume that writes are counted in heat clustering. Figures 17(b) and A-20(b) present the corresponding results for run clustering. Note that including write requests in the access graph improves the write performance for some of the workloads but decreases read performance across the board. Therefore, the default policy in this paper is to consider only reads for run clustering. As for which copy to update, the simulation results suggest updating the runs for the server workloads and invalidating the other copies. For the PC workloads, updating the runs increases read performance slightly but markedly degrades write performance. Therefore, the default policy for the PC workloads is to update the home copy and invalidate the others. That the read performance for PC workloads increases only slightly when runs are updated is not surprising since the runs in this environment are often repeated reads of application binaries, and these are written only when the applications were first installed.

Next, we investigate write policies for the combination of heat and run clustering in Figures 17(c) and A-20(c). We introduce a policy called *RunHeat* that performs a write by first attempting to update the affected blocks in the run-clustered region of the reorganized area. If a block does not exist in the run-clustered region, the policy tries to update that block in the heat-clustered region. If the block does not exist anywhere in the reorganized area, the original copy in the home area is updated. As shown in the figures, RunHeat is the best write policy for all the workloads as far as read performance is concerned. Furthermore, it does not degrade write performance for any of the workloads, and in fact achieves a sizeable improvement of about 5-10% in the average write service time for most of the workloads and up to 22% for the base server workloads in the resource-poor environment. This is the default write policy we use for heat and run clustering combined.

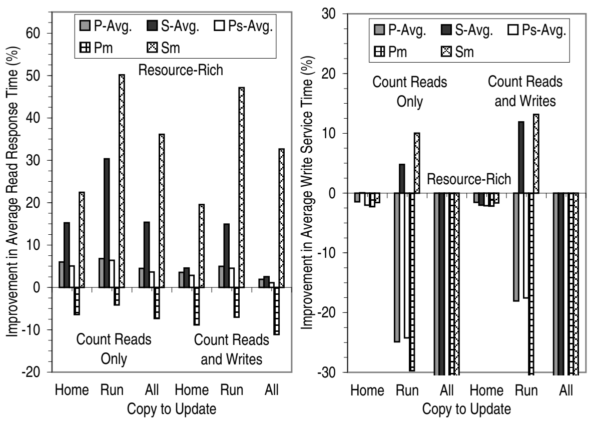
6.4 Workload Stability

The process of reorganizing disk blocks entails a lot of data movement and may potentially affect the service rate of any incoming I/Os. In addition, resources are needed to perform the workload analysis and the optimization that produces the reorganization plan. Therefore it is important to understand how frequently the reorganization needs to be performed and the tradeoffs involved. Figures 18 and A-21 present the sensitivity of the various clustering strategies to the reorganization interval.

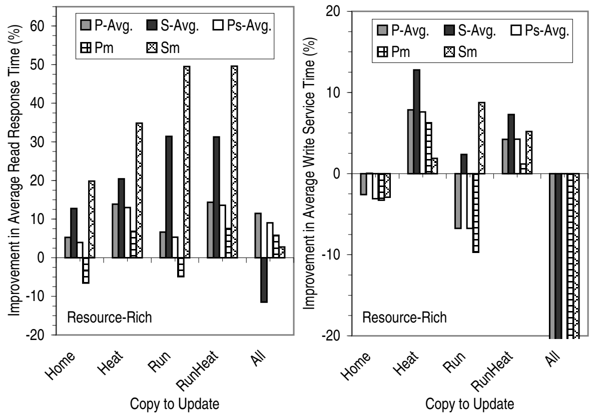
We would expect daily reorganization to perform well because of the diurnal cycle. Our results confirm this. They also show that less frequent reorganization tends to only affect the improvement gradually so that reorganizing daily to weekly is generally adequate. This is consistent with findings in [43]. The only exception is for the average of the server workloads where the effectiveness of ALIS plummets if we reorganize less fre-



(a) Heat Clustering.



(b) Run Clustering.



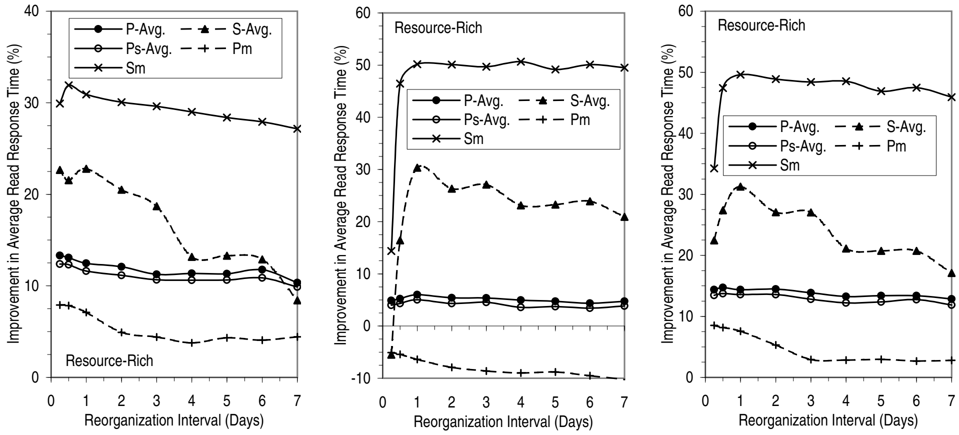
(c) Heat and Run Clustering Combined.

Figure 17: Sensitivity to Write Policies (Resource-Rich).

quently than once every three days. Recall that one of the components of this average is DS1, which is only seven days long. Because we use the first three days of this trace for warm up purposes, if we reorganize less frequently than once every three days, the effect of the reorganization will not be fully reflected. Note that for run clustering and combined heat and run clustering, reorganizing more than once a day performs poorly. This is because the workloads do vary over the course of a day so that if we reorganize too frequently, we are always "chasing the tail" and trying to catch up. Unless otherwise stated, all the results in this paper are for daily reorganization.

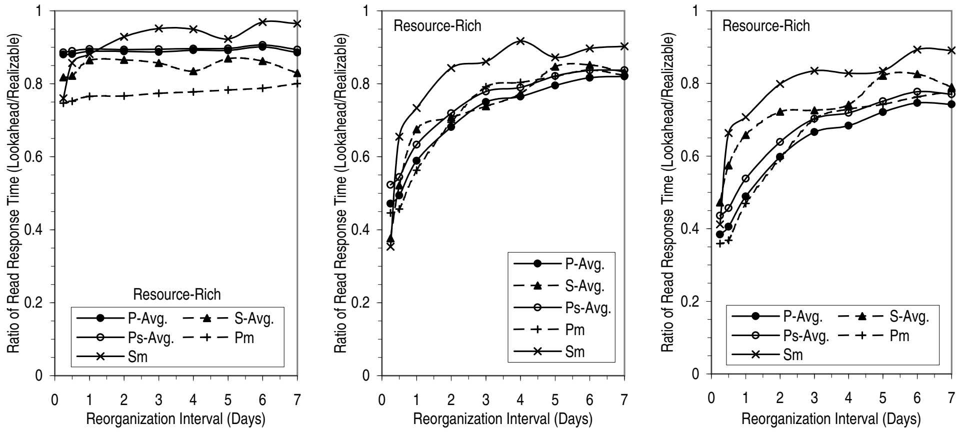
More generally, the various clustering strategies are all based on the reference history. They try to predict future reference patterns by assuming that these patterns are likely to be those that have occurred in the past. The effectiveness of these algorithms is therefore limited by the extent to which workloads vary over time. The above results suggest that there are portions of the workloads that are very stable and are repeated daily since there is but a small effect in varying the reorganization frequency from daily to weekly. To gain further insight into the stability of the workloads, we consider the ideal case where we can look ahead and see the future references of a workload. In Figures 19 and A-22, we show how much better these algorithms perform when they have knowledge of future reference patterns as compared to when they have to predict the future reference patterns from the reference history.

In the figures, the "realizable" performance is that obtained when the reorganization is performed based on the past reference stream or the reference stream seen so far. This is what we have assumed all along. The "looka-



(a) Heat Clustering. (b) Run Clustering. (c) Heat and Run Clustering Combined.

Figure 18: Sensitivity to Reorganization Interval (Resource-Rich).



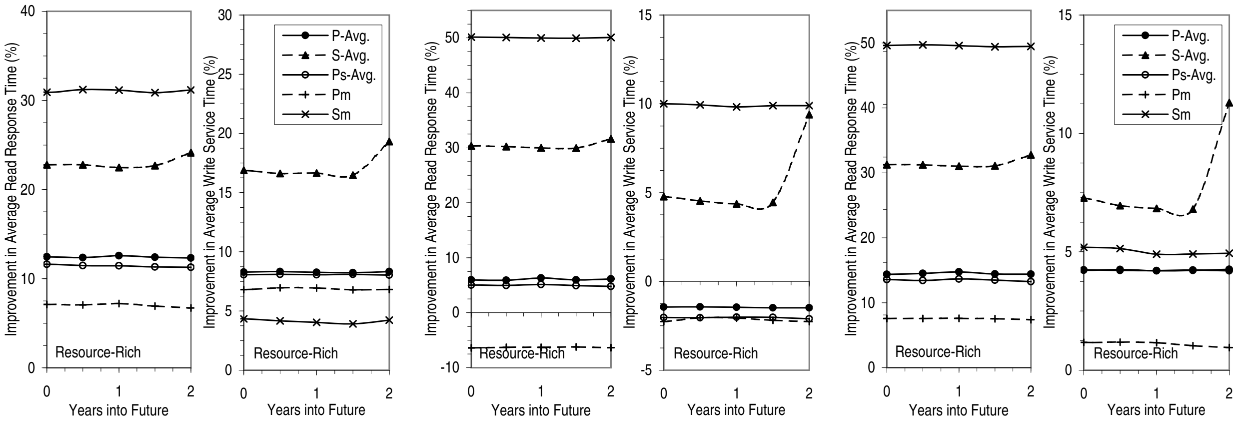
(a) Heat Clustering. (b) Run Clustering. (c) Heat and Run Clustering Combined.

Figure 19: Performance with Knowledge of Future Reference Patterns (Resource-Rich).

head" performance is that achieved when the various clustering algorithms are operated on the future reference stream, specifically the reference stream in the next reorganization interval. (Note that this is not the lookahead optimal algorithm.) Observe that for heat clustering, the difference is small, meaning that the workload characteristic heat clustering exploits is relatively stable. On the other hand, for run clustering and combined heat and run clustering, the reference history is not that good a predictor of the future. With a reorganization frequency of once a day, having forward knowledge outperforms history-based prediction by about 40-50%. In other words, significant portions of the workloads are time-varying or cannot be predicted from past references. This suggests that excessive mining of the reference history for repeated sequences or runs may not be productive.

6.5 Effect of Improvement in the Underlying Disk Technology

Disk technology is constantly evolving. Mechanically, the disk arm is becoming faster over time and the disk is spinning at a higher rate. The recording density



(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

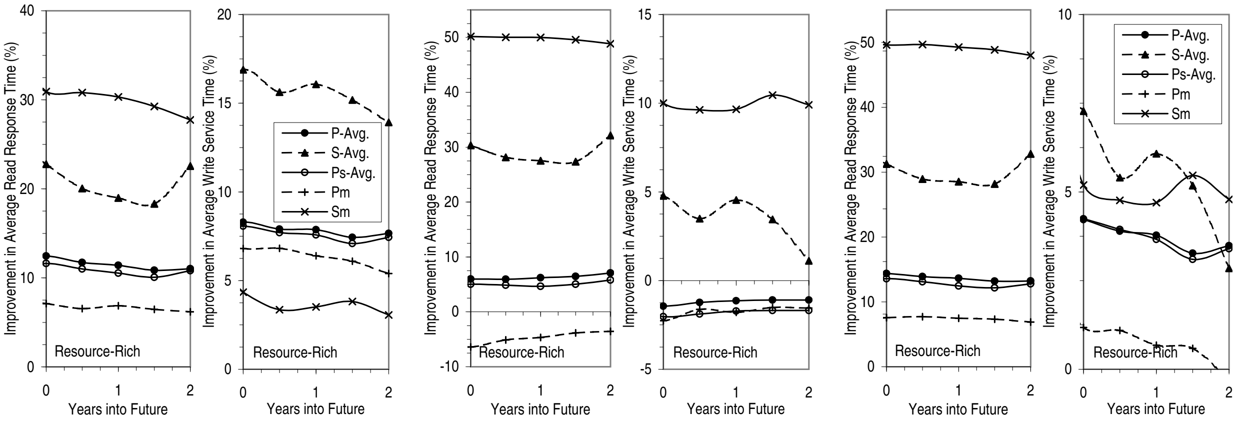
Figure 20: Effectiveness of the Various Clustering Techniques as Disks are Mechanically Improved over Time (Resource-Rich).

is also increasing with each new generation of the disk so that there are more sectors per track and more tracks per inch. The net effect of these trends is that less physical movement is needed to access the same data, and the same physical movement takes a shorter amount of time. We have demonstrated that ALIS is able to achieve dramatic improvement in performance for a variety of real workloads. An interesting issue is whether the effect of ALIS, which tries to reduce both the number of physical movements and the distance moved, will be increased or diminished as a result of these technology trends. It has been pointed out previously that as disks become faster, the benefit of reducing the seek distance will be lessened [43]. In ALIS, we emphasize clustering strategies that not only reduce the seek distance but, more importantly, also eliminate some of the I/Os by increasing the effectiveness of sequential prefetch. This latter effect should be relatively stable and could in fact increase over time as more resources are devoted to prefetching to leverage the rapidly growing disk transfer rate. In this section, we will empirically verify that the benefit of ALIS persists as disk technology evolves.

6.5.1 Mechanical Improvement

We begin by examining the effect of improvement in the mechanical or moving parts of the disk, specifically, the reduction in seek time and the increase in rotational speed; we do not consider density changes in this subsection. The average seek time is generally taken to be the average time needed to seek between two random blocks on the disk. Based on the performance characteristics of server disks introduced by IBM over the last ten years, we found that on average, seek time decreases by about 8% per year while rotational speed increases by about 9% per year [18]. Together, the improvement in the seek time and the rotational speed translate into an 8% yearly improvement in the average response and service times for our various workloads [18].

In Figures 20 and A-23, we investigate how the effect of ALIS changes as the disk is improved mechanically at the historical rates. Observe from the figures that the benefit of ALIS is practically constant over the two-year period. In fact, the plots show a slight upward trend, especially for the server workloads in the resource-poor environment. This slight increase in the effectiveness of ALIS stems from the fact that as the disk becomes faster, it will have more free resources (time) to perform opportunistic prefetch and with ALIS, such opportunistic prefetch is more likely to be useful. There is an abrupt rise in some of the S-Avg. plots at the end of the two-year period because DS1, one of the components of S-Avg., is sensitive to how the blocks are laid out in tracks since some of its accesses, especially the writes, occur in specific patterns. As the disk is sped up, the layout of blocks in the tracks have to be adjusted to ensure that transfers spanning multiple tracks do not "miss revolutions". In some cases, consecutively accessed blocks become poorly positioned rotationally [18] so that the benefit of automatically reorganizing the blocks is especially pronounced. Such situations highlight the value of ALIS in ensuring good performance and reducing the likelihood of unpleasant surprises due to poor block layout.



(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure 21: Effectiveness of the Various Clustering Techniques as Disk Recording Density is Increased over Time (Resource-Rich).

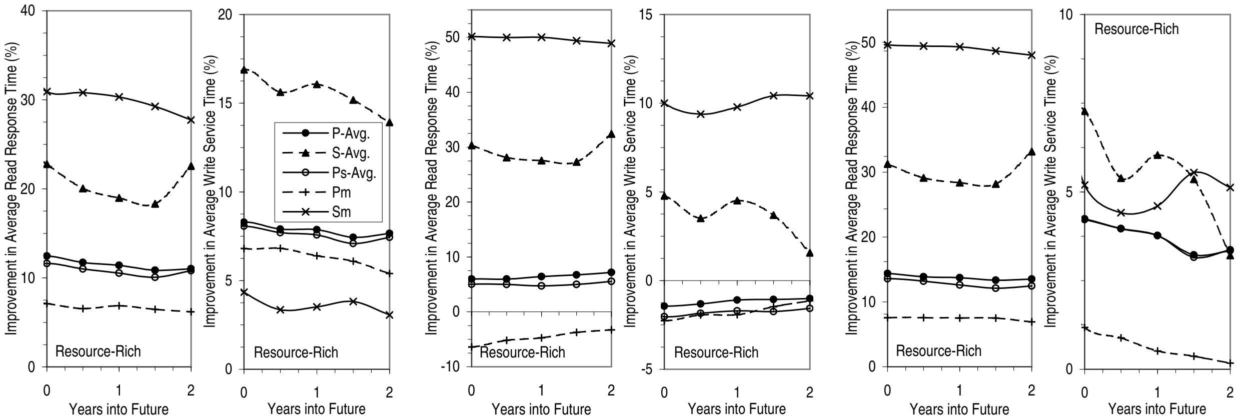
6.5.2 Increase in Recording Density

Increasing the recording or areal density reduces the cost of disk-based storage. Areal density improvement also directly affects performance because as bits are packed more closely together, they can be accessed with a smaller physical movement. Recently, linear density has increased by about 21% per year while track density has risen by approximately 24% per year [18]. For our workloads, the average response and service times are improved by about 9% per year as a result of the increase in areal density [18].

In Figures 21 and A-24, we analyze how areal density improvement affects the effectiveness of ALIS. Observe that there is a downward trend in most of the plots, especially those that relate to heat clustering. This is expected because heat clustering derives part of its benefit from seek distance reduction which is less effective as disks become denser and the difference between a long and short seek is reduced. The improvement in write performance, also being dependent on a reduction in seek distance, shows a similar downward trend. On the other hand, the performance benefit of run clustering is relatively insensitive to the increase in areal density since the main effect of run clustering is to reduce the number of I/Os by increasing the effectiveness of sequential prefetch. The same is true of heat and run clustering combined. Such a result is quite remarkable because at the historical rate of increase in areal density, two years of improvement translates into more than a doubling of disk capacity. This means that in going forward two years in time in Figures 21 and A-24, we are seriously short-stroking the disk by using less than half the available disk space. Yet ALIS is still able to achieve a dramatic improvement in performance.

6.5.3 Overall Effect of Disk Technology Trends

Putting together the effect of mechanical improvement and areal density scaling, we obtain the overall performance effect of disk technology evolution. For our various workloads, the average response and service time are projected to improve by about 15% per year [18]. In Figures 22 and A-25, we plot the additional performance improvement that ALIS can provide. Note that the benefit of ALIS with heat and run clustering combined is generally stable over time with only a very slight downward inclination. In the worst case, going forward two years in time reduces the improvement with ALIS from 50% to 48% for the merged server workload in the resource-rich environment. As discussed earlier, this is quite impressive because at the end of the two-year period, we are using less than half the available disk space. In the more realistic situation where we try to take advantage of the increased disk space, the benefit of ALIS will be even more enduring. Note that we did not re-optimize any of the parameters of the underlying storage system for ALIS. We simply use the settings that have been found to work well for the base system [18]. If we were to tune the underlying storage system to exploit the increasing gap between random and sequential performance, and the improved locality that ALIS provides, the benefit of ALIS should be all the more stable and substantial.



(a) Heat Clustering. (b) Run Clustering. (c) Heat and Run Clustering Combined.

Figure 22: Effectiveness of the Various Clustering Techniques as Disk Technology Evolves over Time (Resource-Rich).

7 Conclusions

In this paper, we propose ALIS, an introspective storage system that continually analyzes I/O reference patterns to replicate and reorganize selected disk blocks so as to improve the spatial locality of reference, and hence leverage the dramatic improvement in disk transfer rate. Our analysis of ALIS suggests that disk block layout can be effectively optimized by an autonomic storage system, without human intervention. Specifically, we find that the idea of clustering together hot or frequently accessed data has the potential to significantly improve storage performance, if handled in such a way that existing spatial locality is not disrupted. In addition, we show that by discovering repeated read sequences or runs and laying them out sequentially, we can greatly increase the effectiveness of sequential prefetch. By further combining these two ideas, we are able to reap the greater benefit of the two schemes and achieve performance that is superior to either technique alone. In fact, with the combined scheme, most of the disk reads are either eliminated due to the more effective prefetch or can be satisfied from the reorganized data, an outcome which greatly simplifies the practical issue of deciding where to locate the reorganized data.

Using trace-driven simulation with a large collection of real server and PC workloads, we demonstrate that ALIS is able to far outperform prior techniques in both an environment where the storage system consists of only disks and low-end disk adaptors, and one where there is a large outboard controller. For the server workloads, read performance is improved by between 31% and 50% while write performance is improved by as much as 22%. The read performance for the PC workloads is improved by about 15% while the writes are faster by up to 8%. Given that disk performance, as perceived by real workloads, is increasing by about 8% per year assuming that the disk occupancy rate is kept constant [18], such improvements may be equivalent to as much as several years of technological progress at the historical rates. As part of our analysis, we also examine how improvement in disk technology will impact the effectiveness of ALIS and confirm that the benefit of ALIS is relatively insensitive to disk technology trends.

8 Future Work

ALIS currently behaves like an open-loop control system that is driven solely by the workload and a simple performance model of the underlying storage system, namely that it is able to service sequential I/O much more efficiently than random I/O. Because the performance of disks today is so much higher when data is read sequentially rather than randomly, this simple performance model is sufficiently accurate to produce a dramatic performance improvement. But for increased robustness, it would be worthwhile to explore the idea of incorporating some feedback into the optimization process to, for example, turn ALIS off when it is not performing well or, at a finer granularity, influence how blocks are laid out.

In the design of ALIS, we try to desensitize its performance to the various parameters so that it is not catastrophic for somebody to "configure the system wrongly". To reflect the likely situation that ALIS will be used with a default setting, we base our performance evaluation on parameter settings that are good for an entire class of workloads rather than on the best values for each individual workload. A useful piece of future work would be to devise ways to set the various parameters dynamically to adapt to each individual workload. The general approach of using hill-climbing to gradually adjust each knob until a local optimum is reached should be very effective for ALIS because the results of our various sensitivity studies suggest that for the parameters in ALIS, the local optimum is likely to be also the global optimum.

Acknowledgements

The authors would like to thank Ruth Azevedo, Jacob Lorch, Bruce McNutt and John Wilkes for providing the traces used in this study. Thanks are also owed to William Guthrie who shared with us his expertise in modeling disk drives, and to Ed Grochowski who provided the historical performance data for IBM disk drives. In addition, the authors are grateful to John Palmer and Jai Menon for helpful comments on versions of this paper.

References

[1] S. Akyürek and K. Salem, "Adaptive block rearrangement," *ACM Transactions on Computer Systems*, 13, 2, pp. 89–121, May 1995.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, (Pacific Grove, CA), pp. 198–212, Oct. 1991.

[3] B. E. Bakke, F. L. Huss, D. F. Moertl, and B. M. Walk, "Method and apparatus for adaptive localization of frequently accessed, randomly addressed data." U.S. Patent 5765204. Filed June 5, 1996. Issued June 9, 1998.

[4] S. D. Carson and P. F. Reynolds Jr, "Adaptive disk reorganization," Techical Report UMIACS-TR-89-4, Department of Computer Science, University of Maryland, Jan. 1989.

[5] C. L. Chee, H. Lu, H. Tang, and C. V. Ramamoorthy, "Adaptive prefetching and storage reorganization in a log-structured storage system,"

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: high-performance, reliable secondary storage," *ACM Computing Surveys*, 26, 2, pp. 145–185, June 1994.

[7] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson, "A quantitative analysis of cache policies for scalable network file systems," *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 150–160, May 1994.

[8] Executive Software International Inc., "Diskeeper 6.0 second edition for Windows," 2001. http://www.execsoft.com/diskeeper/diskeeper.asp.

[9] D. Ferrari, "Improvement of program behavior," *Computer*, 9, 11, pp. 39–47, Nov. 1976.

[10] G. R. Ganger and M. F. Kaashoek, "Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files," *Proceedings of USENIX Technical Conference*, (Anaheim, CA), pp. 1–17, Jan. 1997.

[11] J. Gray, "Put EVERYTHING in the storage device." Talk at NASD Workshop on Storage Embedded Computing, June 1998. http://www.nsic.org/nasd/1998-jun/gray.pdf.

[12] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," *Proceedings of Summer USENIX Conference*, pp. 197–207, June 1994.

[13] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson, "Multiple prefetch adaptive disk caching," *IEEE Transactions on Knowledge and Data Engineering*, 5, 1, pp. 88–103, Feb. 1993.

[14] E. Grochowski, "IBM magnetic hard disk drive technology," 2002. http://www.hgst.com/hdd/technolo/grochows/grocho01.htm.

[15] R. R. Heisch, "Trace-directed program restructuring for AIX executables," *IBM Journal of Research and Development*, 38, 5, pp. 595–603, Sept. 1994.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, second ed., 1996.

[17] W. W. Hsu and A. J. Smith, "Characteristics of I/O traffic in personal computer and server workloads," *IBM Systems Journal*, 42, 2, pp. 347–372, 2003.

[18] W. W. Hsu and A. J. Smith, "The real effect of I/O optimizations and disk improvements." Technical Report, CSD-03-1263, Computer Science Division, University of California, Berkeley, July 2003. Also available as Chapter 3 of [22].

[19] W. W. Hsu, A. J. Smith, and H. C. Young, "Projecting the performance of decision support workloads on systems with smart storage (SmartSTOR)," *Proceedings of IEEE Seventh International Conference on Parallel and Distributed Systems (ICPADS)*, (Iwate, Japan), pp. 417–425, July 2000.

[20] W. W. Hsu, A. J. Smith, and H. C. Young, "Analysis of the characteristics of production database workloads and comparison with the TPC benchmarks," *IBM Systems Journal*, 40, 3, pp. 781–802, 2001.

[21] W. W. Hsu, A. J. Smith, and H. C. Young, "I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level," *ACM Transactions on Database Systems*, 26, 1, pp. 96–143, Mar. 2001.

[22] W. W. Hsu, *Dynamic Locality Improvement Techniques for Increasing Effective Storage Performance*, PhD thesis, University of California, Berkeley, 2002. Available as Technical Report CSD-03-1223, Computer Science Division, University of California, Berkeley, Jan. 2003.

[23] IBM Corp., *Autonomic Computing: IBM's Perspective on the State of Information Technology*, 2001. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[24] IBM Corp., *Ultrastar 73LZX Product Summary Version 1.1*, 2001.

[25] Intel Corp., "Intel application launch accelerator," Mar. 1998. http://www.intel.com/ial/ala.

[26] K. Keeton, D. Patterson, and J. Hellerstein, "A case for intelligent disks (IDISKs)," *ACM SIGMOD Record*, 27, 3, pp. 42–52, 1998.

[27] D. E. Knuth, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, MA, 1973.

[28] T. M. Kroeger and D. D. E. Long, "Predicting file system actions from prior events," *Proceedings of USENIX Annual Technical Conference*, pp. 319–328, Jan. 1996.

[29] H. Lei and D. Duchamp, "An analytical approach to file prefetching," *Proceedings of the USENIX Annual Technical Conference*, (Anaheim, CA), pp. 275–288, Jan. 1997.

[30] J. R. Lorch and A. J. Smith, "The VTrace tool: Building a system tracer for Windows NT and Windows 2000," *MSDN Magazine*, 15, 10, pp. 86–102, Oct. 2000.

[31] T. Masuda, H. Shiota, K. Noguchi, and T. Ohki, "Optimization of program organization by cluster analysis," *Proceedings of IFIP Congress*, pp. 261–265, 1974.

[32] 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," *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, (Saint-Malo, France), pp. 238–251, Oct. 1997.

[33] S. McDonald, "Dynamically restructuring disk space for improved file system performance," Techical Report 88-14, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, July 1988.

[34] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Transactions on Computer Systems*, 2, 3, pp. 181–197, Aug. 1984.

[35] B. McNutt, "MVS DASD survey: Results and trends," *Proceedings of Computer Measurement Group (CMG) Conference*, (Nashville, TN), pp. 658–667, Dec. 1995.

[36] L. W. McVoy and S. R. Kleiman, "Extent-like performance from a UNIX file system," *Proceedings of Winter USENIX Conference*, (Dallas, TX), pp. 33–43, Jan. 1991.

[37] Mesquite Software Inc., *CSIM18 simulation engine (C++ version)*, 1994.

[38] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: A case for log-structured file systems," *Operating Systems Review*, 23, 1, pp. 11–28, Jan. 1989.

[39] M. Palmer and S. B. Zdonik, "Fido: A cache that learns to fetch," Techical Report CS-91-15, Department of Computer Science, Brown University, Feb. 1991.

[40] J. K. Peacock, "The counterpoint fast file system," *USENIX Conference Proceedings*, (Dallas, TX), pp. 243–249, Winter 1988.

[41] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," *Proceedings of International Conference on Very Large Data Bases (VLDB)*, (New York, NY), pp. 62–73, Aug. 1998.

[42] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," *Proceedings of USENIX Annual Technical Conference*, (Berkeley, CA), pp. 41–54, June 2000.

[43] C. Ruemmler and J. Wilkes, "Disk shuffling," Techical Report HPL-91-156, HP Laboratories, Oct. 1991.

[44] C. Ruemmler and J. Wilkes, "UNIX disk access patterns," *Proceedings of USENIX Winter Conference*, (San Diego, CA), pp. 405–420, Jan. 1993.

[45] A. J. Smith, "Disk cache — miss ratio analysis and design considerations," *ACM Transactions on Computer Systems*, 3, 3, pp. 161–203, Aug. 1985.

[46] A. J. Smith, "Trace driven simulation in research on computer architecture and operating systems," *Proceedings of Conference on New Directions in Simulation for Manufacturing and Communications*, (Tokyo, Japan), pp. 43–49, Aug. 1994.

[47] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Transactions on Database Systems*, 3, 3, pp. 223–247, Sept. 1978.

[48] C. Staelin and H. Garcia-Molina, "Smart filesystems," *Proceedings of USENIX Winter Conference*, pp. 45–52, Jan. 1991.

[49] Symantec Corp., "Norton Utilities 2002," 2001. http://www.symantec.com/nu/nu_9x.

[50] M. M. Tsangaris and J. F. Naughton, "On the performance of object clustering techniques," *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 144–153, June 1992.

[51] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys*, 29, 2, pp. 128–170, June 1997.

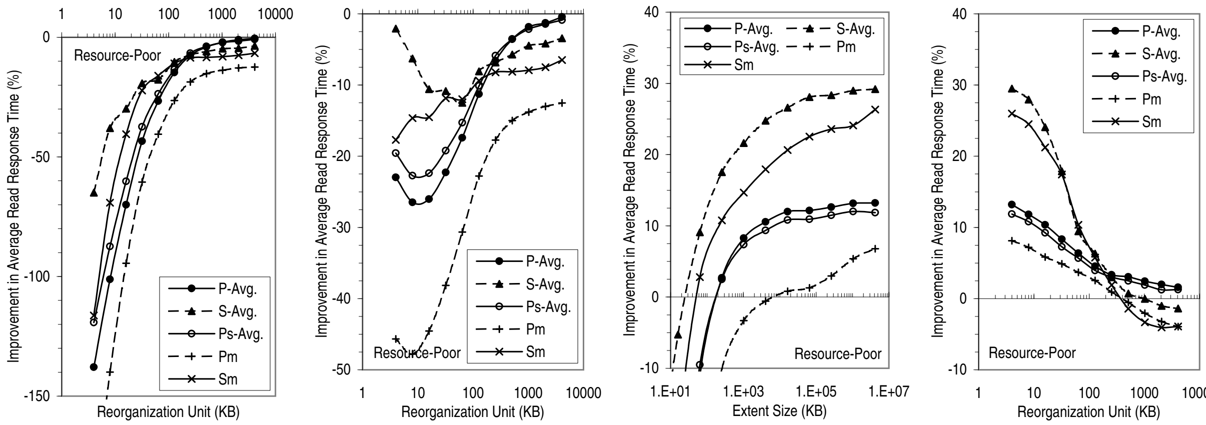
[52] J. S. M. Verhofstad, "Recovery techniques for database systems," *ACM Computing Surveys*, 10, 2, pp. 167–195, June 1978.

[53] P. Vongsathorn and S. D. Carson, "A system for adaptive disk rearrangement," *Software – Practice and Experience*, 20, 3, pp. 225–242, Mar. 1990.

[54] A. E. Whipple II, "Optimizing a magnetic disk by allocating files by the frequency a file is acessed/updated or by designating a file to a fixed location on a disk." U.S. Patent 5333311. Filed Dec. 10, 1990. Issued July 26, 1994.

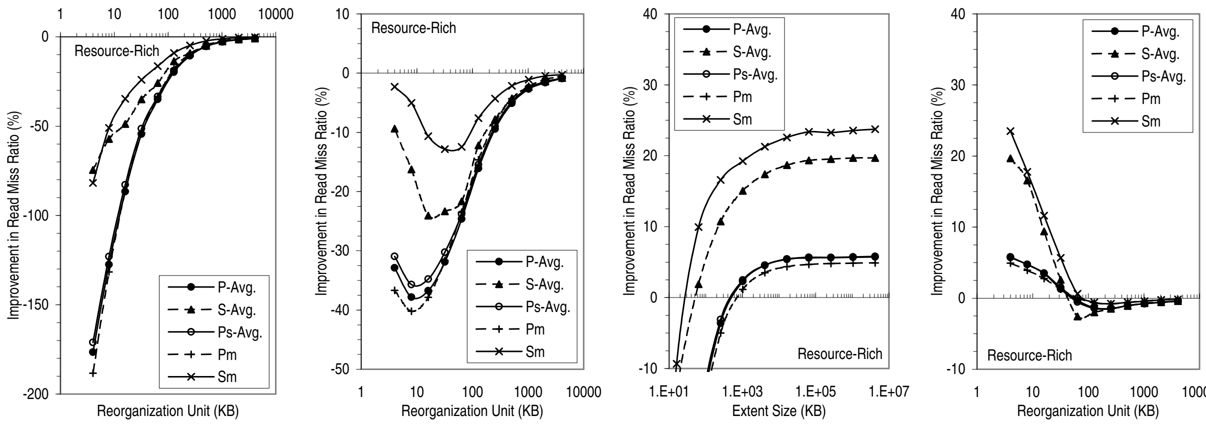
[55] M. Zhou and A. J. Smith, "Analysis of personal computer workloads," *Proceedings of International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, (College Park, MD), pp. 208–217, oct 1999.

Appendix



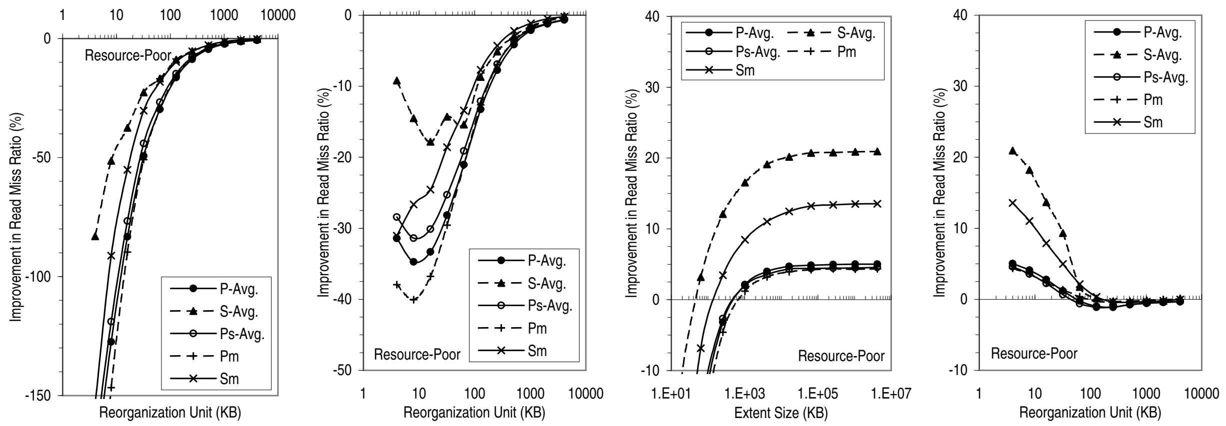
(a) Organ Pipe. (b) Link Closure. (c) Packed Extents. (d) Sequential.

Figure A-1: Effectiveness of Various Heat Clustering Block Layouts at Improving Average Read Response Time (Resource-Poor).



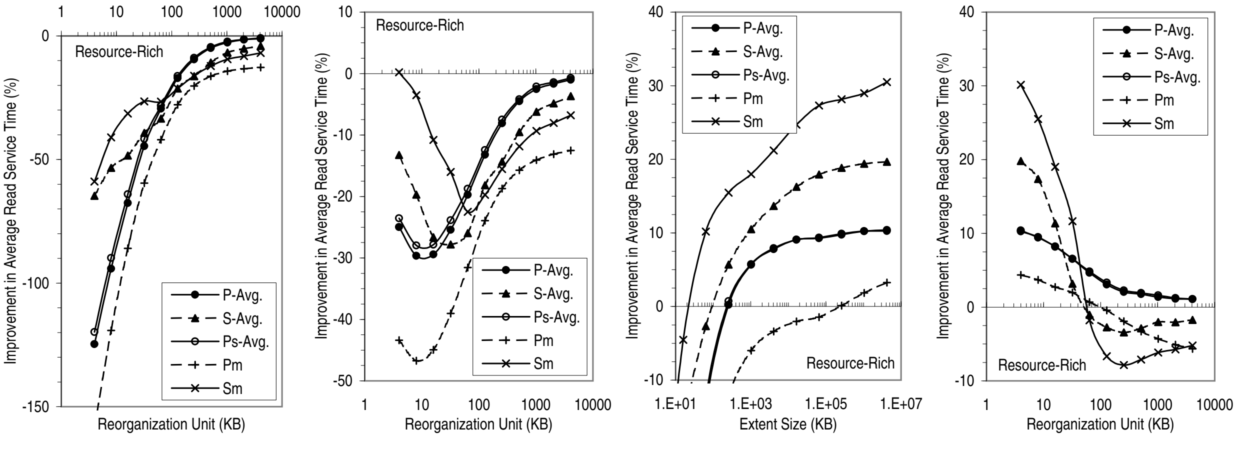
(a) Organ Pipe. (b) Link Closure. (c) Packed Extents. (d) Sequential.

Figure A-2: Effectiveness of Various Heat Clustering Block Layouts at Reducing Read Miss Ratio (Resource-Rich).



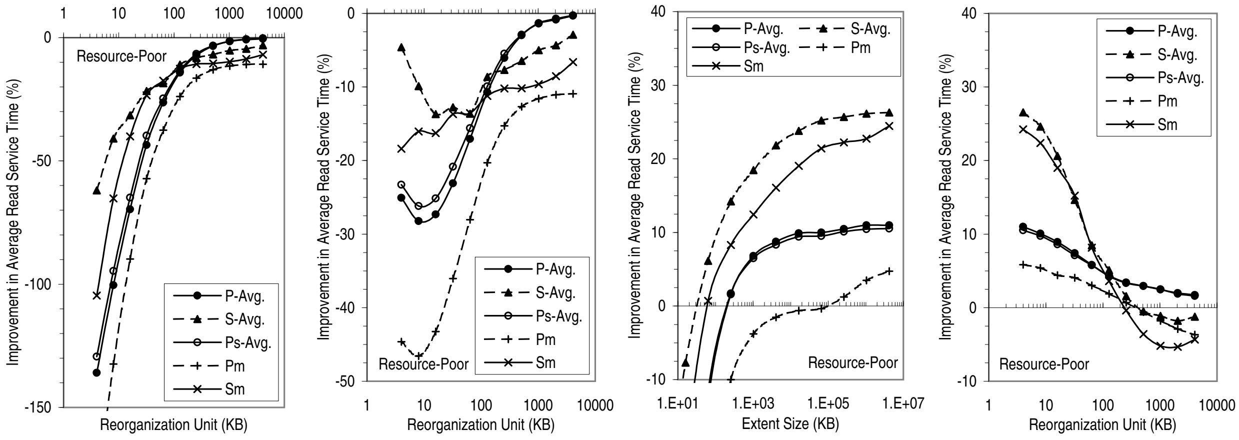
(a) Organ Pipe. (b) Link Closure. (c) Packed Extents. (d) Sequential.

Figure A-3: Effectiveness of Various Heat Clustering Block Layouts at Reducing Read Miss Ratio (Resource-Poor).



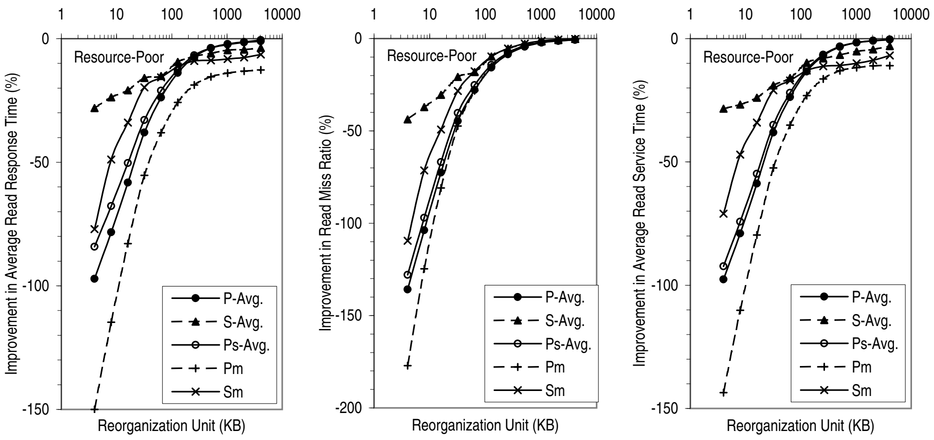
(a) Organ Pipe. (b) Link Closure. (c) Packed Extents. (d) Sequential.

Figure A-4: Improvement in Average Read Service Time for the Various Block Layouts in Heat Clustering (Resource-Rich).

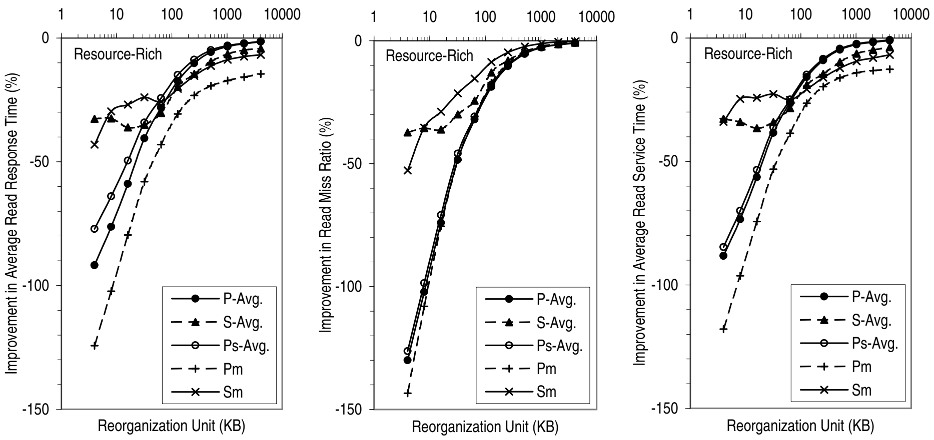


(a) Organ Pipe. (b) Link Closure. (c) Packed Extents. (d) Sequential.

Figure A-5: Improvement in Average Read Service Time for the Various Block Layouts in Heat Clustering (Resource-Poor).



(a) Resource-Poor.



(b) Resource-Rich.

Figure A-6: Effectiveness of Heat Layout at Improving Read Performance.

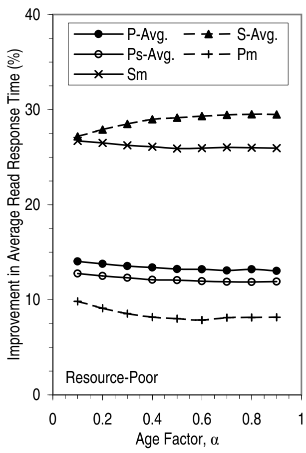
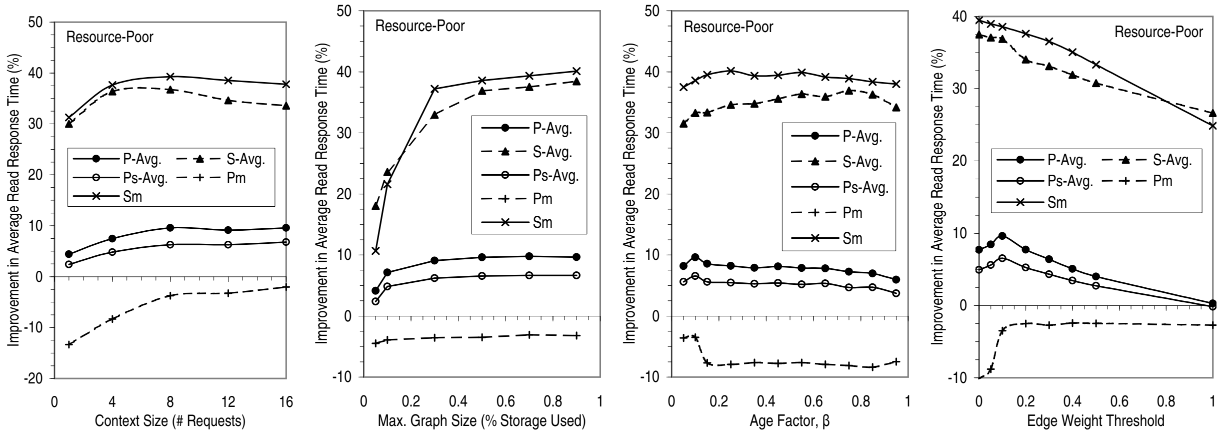


Figure A-7: Sensitivity of Heat Clustering to Age Factor, α (Resource-Poor).



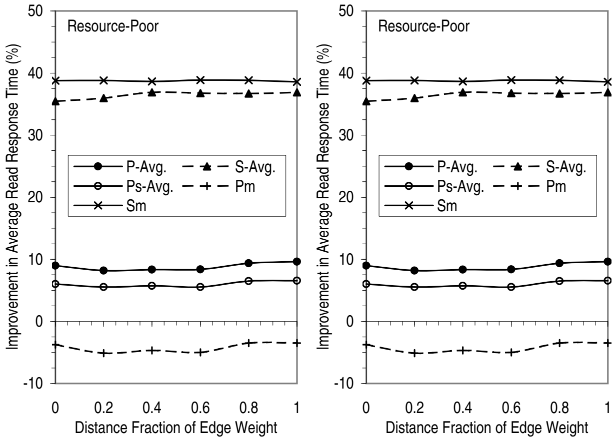
(a) Context Size, τ.

(b) Graph Size.

(c) Age Factor, β.

(d) Edge Threshold.

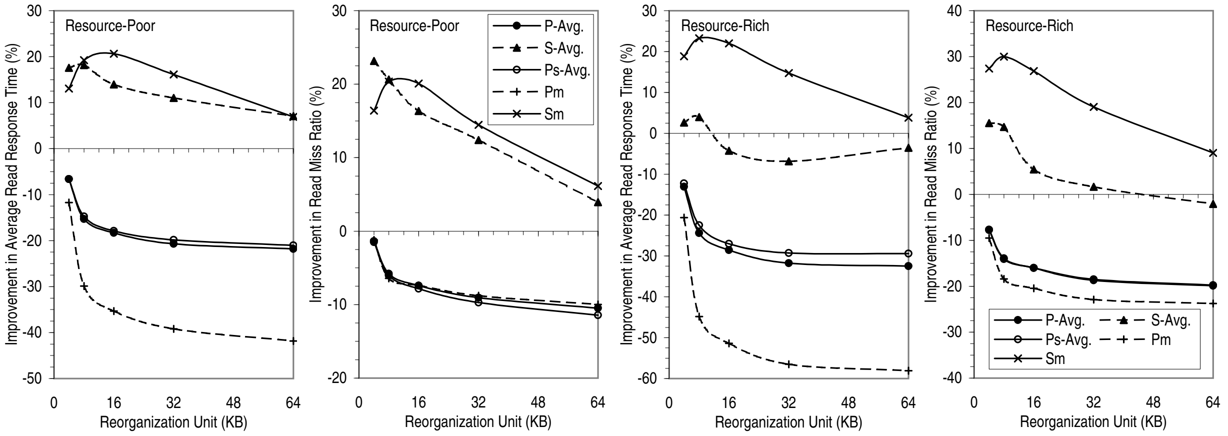
Figure A-8: Sensitivity of Run Clustering to Various Parameters (Resource-Poor).



(a) Resource-Poor.

(b) Resource-Rich.

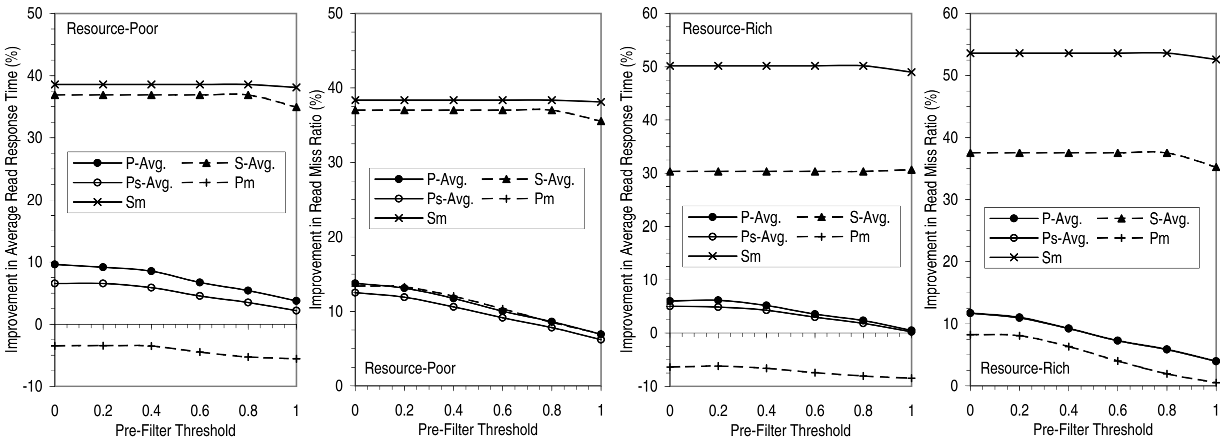
Figure A-9: Sensitivity of Run Clustering to Weighting of Edges.



(a) Resource-Poor.

(b) Resource-Rich.

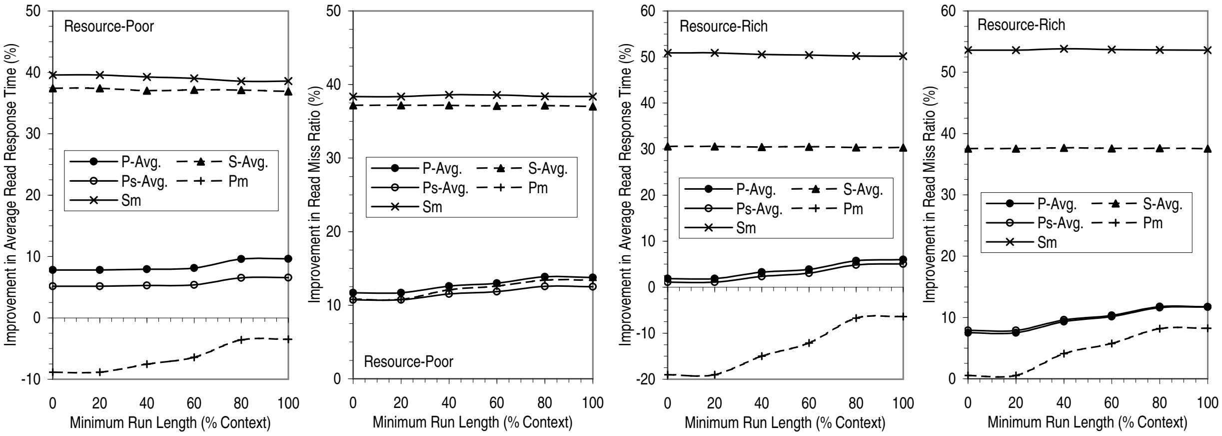
Figure A-10: Effectiveness of Run Clustering with Fixed-Sized Reorganization Units.



(a) Resource-Poor.

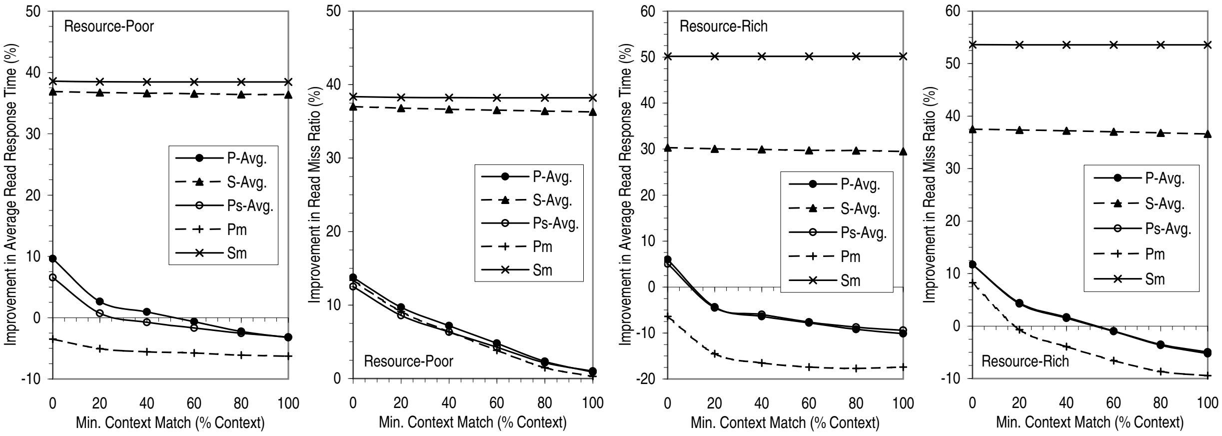
(b) Resource-Rich.

Figure A-11: Effect of Pre-Filtering on Run Clustering.



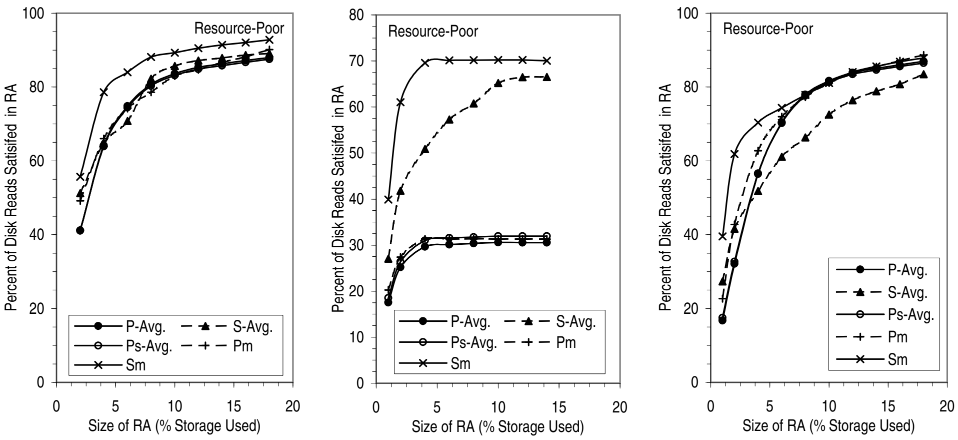
(a) Resource-Poor. (b) Resource-Rich.

Figure A-12: Effect of Imposing Minimum Run Length.



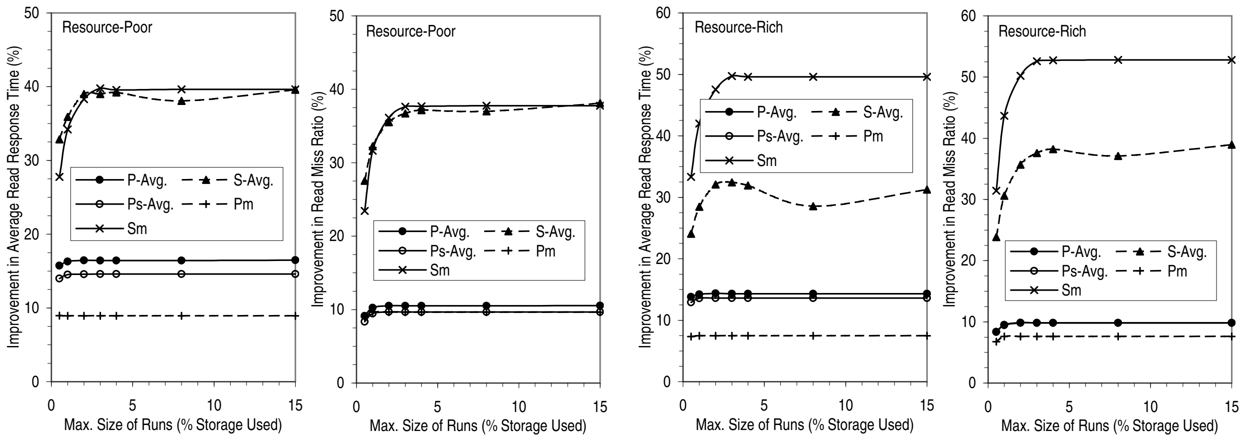
(a) Resource-Poor. (b) Resource-Rich.

Figure A-13: Effect of Using a Run only when the Contexts Match.



(a) Heat Clustering. (b) Run Clustering. (c) Heat and Run Clustering Combined.

Figure A-14: Percent of Disk Reads Satisfied in Reorganized Area (Resource-Poor).



(a) Resource-Poor. (b) Resource-Rich.

Figure A-15: Effect of Limiting the Total Size of Runs in the Reorganized Area.

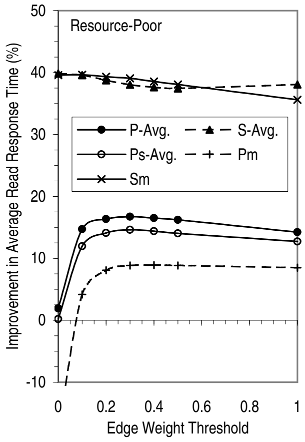


Figure A-16: Sensitivity of Heat and Run Clustering Combined to Edge Threshold (Resource-Poor).

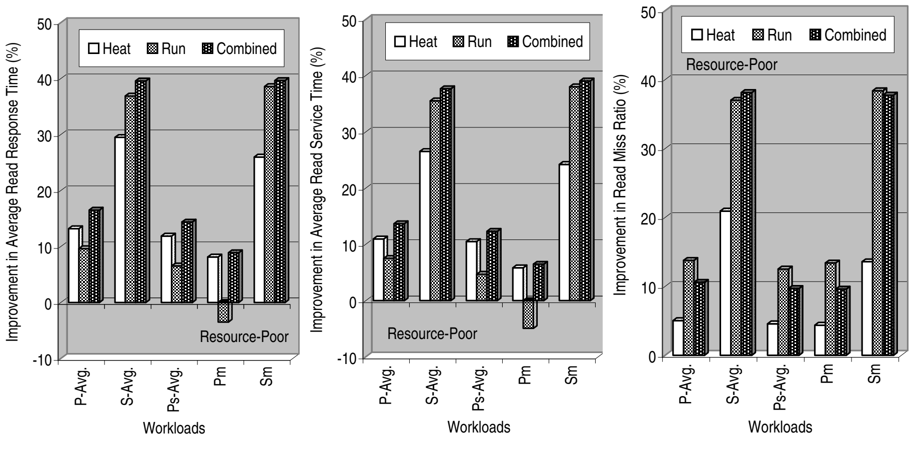
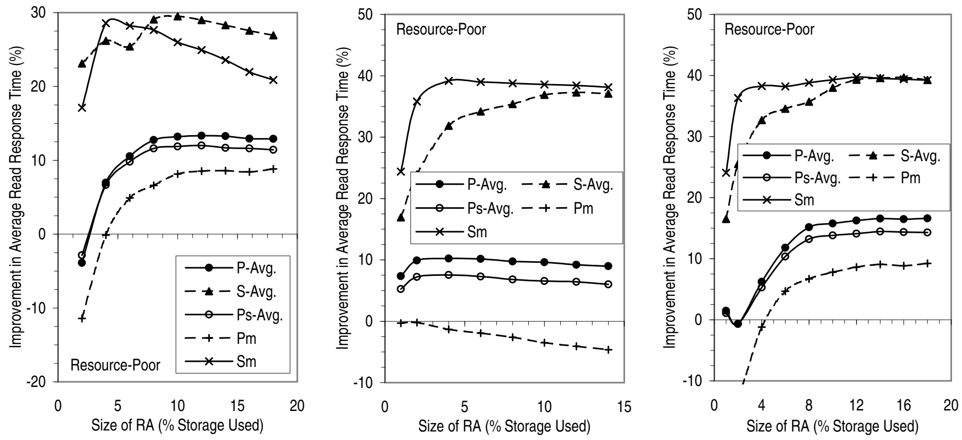


Figure A-17: Performance Improvement with the Various Clustering Schemes (Resource-Poor).

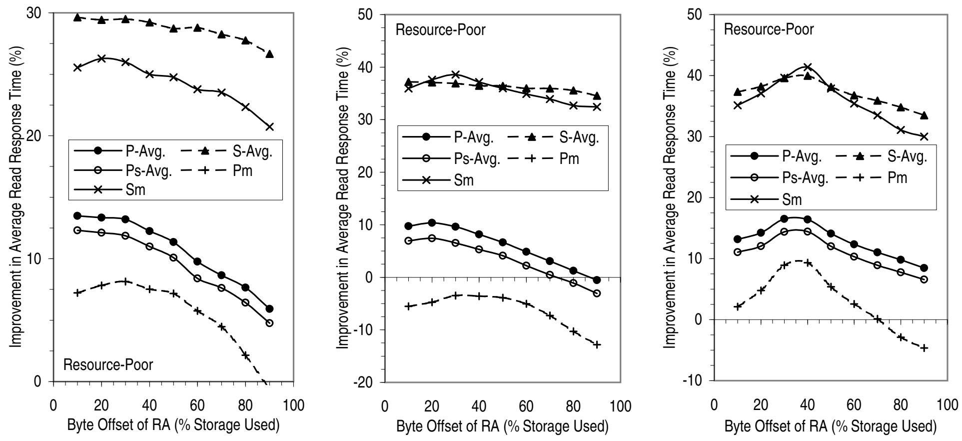


(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure A-18: Sensitivity to Size of Reorganized Area (Resource-Poor).

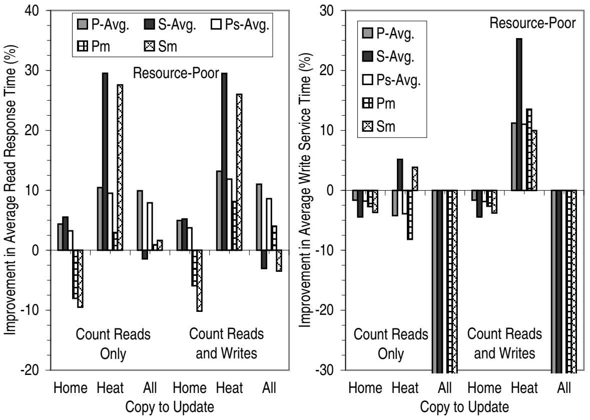


(a) Heat Clustering.

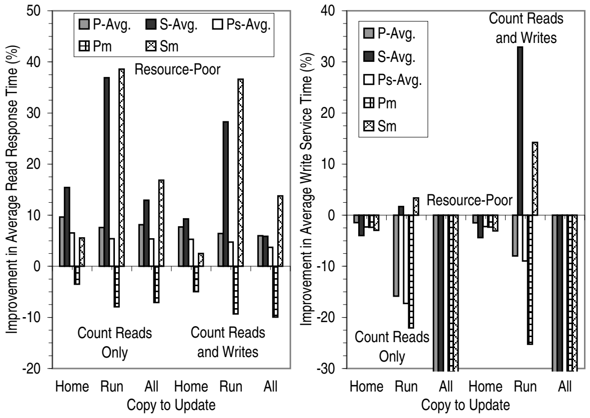
(b) Run Clustering.

(c) Heat and Run Clustering Combined.

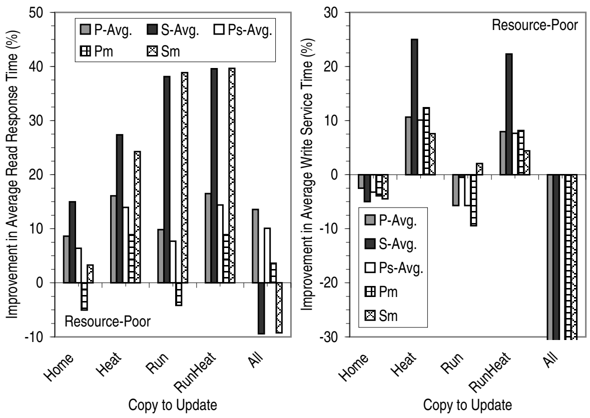
Figure A-19: Sensitivity to Placement of Reorganized Area (Resource-Poor).



(a) Heat Clustering.

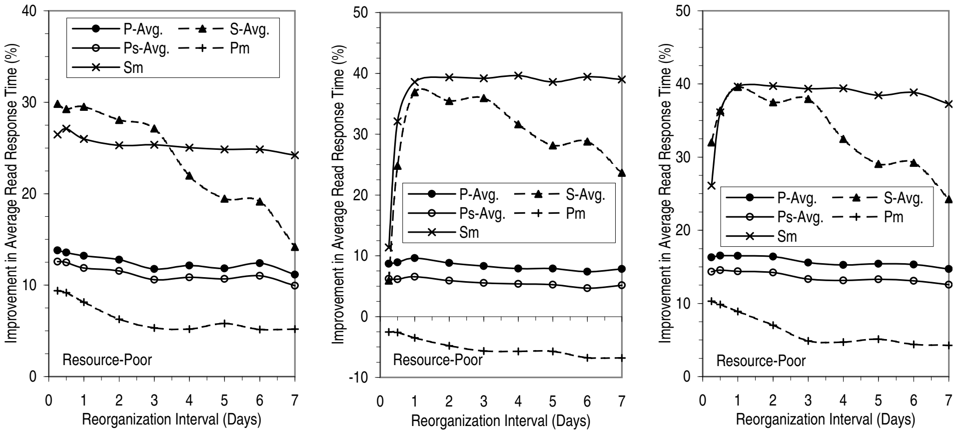


(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure A-20: Sensitivity to Write Policies (Resource-Poor).

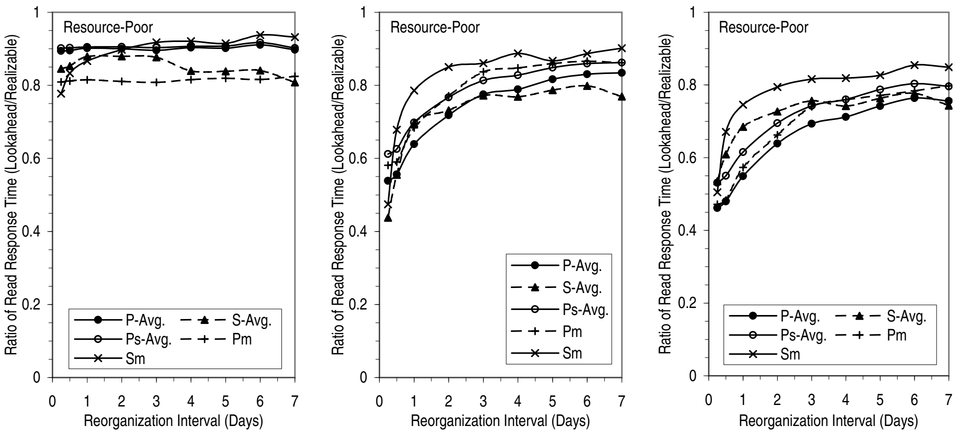


(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure A-21: Sensitivity to Reorganization Interval (Resource-Poor).

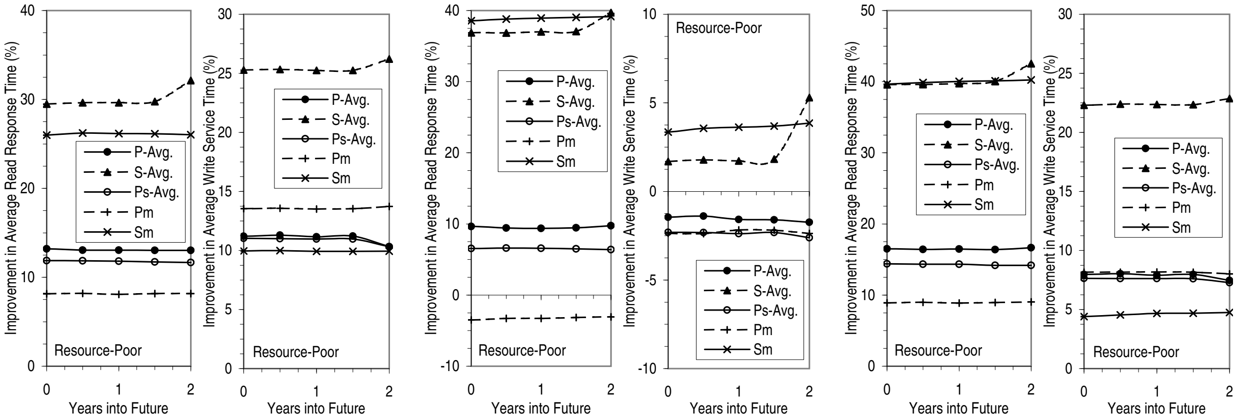


(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure A-22: Performance with Knowledge of Future Reference Patterns (Resource-Poor).

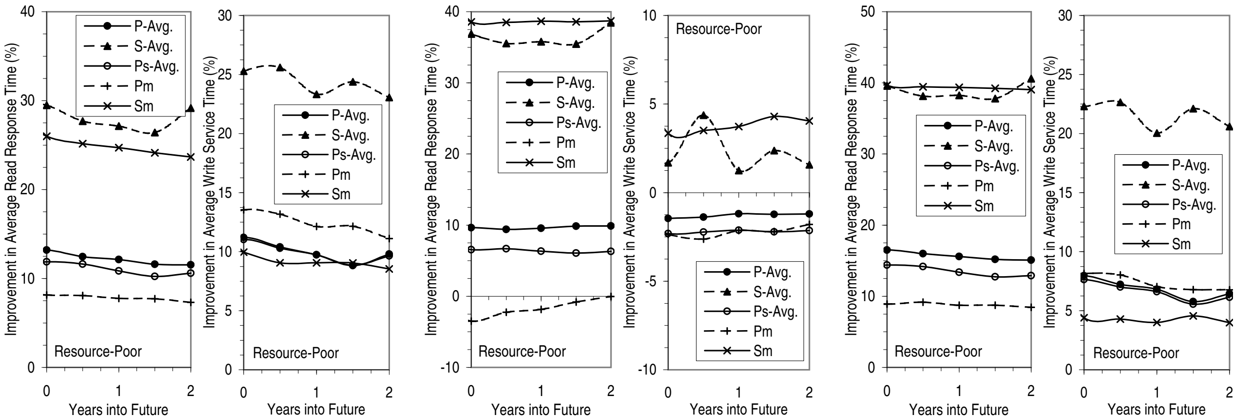


(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure A-23: Effectiveness of the Various Clustering Techniques as Disks are Mechanically Improved over Time (Resource-Poor).

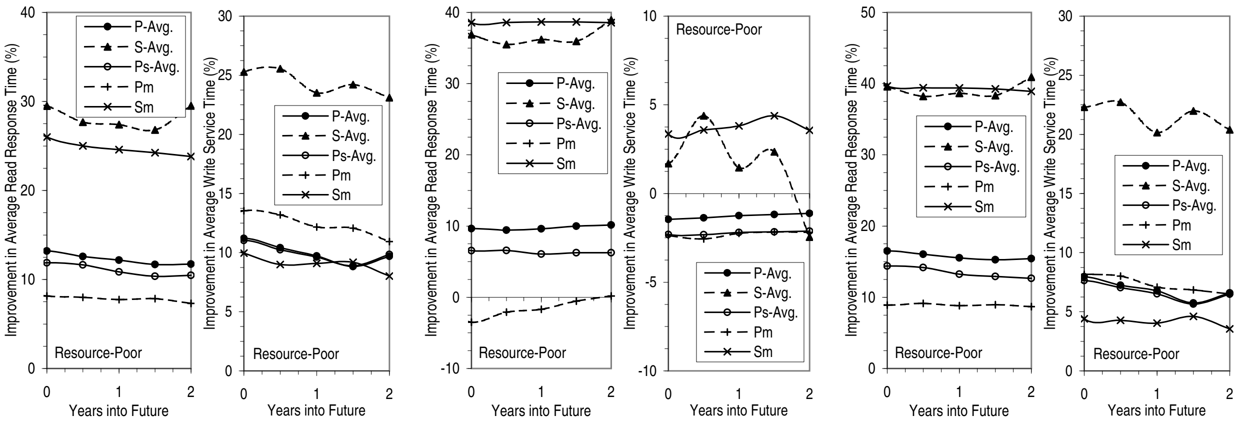


(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure A-24: Effectiveness of the Various Clustering Techniques as Disk Recording Density is Increased over Time (Resource-Poor).



(a) Heat Clustering.

(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure A-25: Effectiveness of the Various Clustering Techniques as Disk Technology Evolves over Time (Resource-Poor).