Disk Caching in Large Databases and Timeshared Systems [1]

Barbara Tockey Zivkov [2]
Alan Jay Smith
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776, USA
September, 1996

Abstract:

We present the results of a variety of trace-driven simulations of disk cache design. Our traces come from a variety of mainframe timesharing and database systems in production use. We compute miss ratios, run lengths, traffic ratios, cache residency times, degree of memory pollution and other statistics for a variety of designs, varying block size, prefetching algorithm and write algorithm. We find that for this workload, sequential prefetching produces a significant (about 20%) but still limited improvement in the miss ratio, even using a powerful technique for detecting sequentiality. Copy-back writing decreased write traffic relative to write-through; periodic flushing of the dirty blocks increased write traffic only slightly compared to pure write-back, and then only for large cache sizes. Write-allocate had little effect compared to no-write-allocate. Block sizes of over a track don't appear to be useful. Limiting cache occupancy by a single processor transaction appears to have little effect. This study is unique in the variety and quality of the data used in the studies.

1 Introduction

Disk cache is a level of cache storage intermediate between disk devices and the CPU. With disk access times of around 10 milliseconds improving only slowly and CPU cycle times of around 10 nanoseconds, and decreasing quickly, such a level of storage is becoming increasingly important.

This paper presents a trace-driven study of disk caching in a variety of environments. We obtained traces of disk reference behavior from large databases and timeshared computers and used these to drive simulations of disk caches of various sizes with different management policies.

This study is uniquely valuable in several ways: our traces represent a greater variety of workloads than any others previously published, there are no comparable studies for database traces, and our data come from production systems. Although the traces come from mainframe systems, which are a shrinking fraction of the computing universe, the size and power of the systems traced are comparable to today's PC and Unix system server environments, and should accurately reflect performance on today's systems, for comparable workloads. We note that the type of workloads considered here—databases, transaction processing, and general time sharing, are different than those seen until recently on PC and Unix systems. Such workloads are rapidly moving to PC and Unix systems, however, and we believe that the file system workloads observed here will also be typical of similar user workloads even for the different operating systems and physical configurations of PC/Unix systems.

1. The authors' research has been or is supported in part by the National Science Foundation under grants MIP-9116578 and CCR-9117028, by the State of California under the MICRO program, and by Intel Corporation, Sun Microsystems, Apple Computer Corporation, and the International Business Machines Corporation.
2. Silicon Graphics, Inc., 2011 North Shoreline Blvd., Mountain View, CA, 94039

In Section 2 we discuss some of the fundamentals of disk caching. Section 3 provides an overview of previous work and disk cache products. Section 4 describes the methodology we used in this study, trace-driven simulation, and the traces we used in our experiments. Section 5 characterizes the data, and Section 6 presents the results of our disk cache simulations. Finally, in Section 7 we draw conclusions and make suggestions for future research.

2 Terms and Concepts

A disk cache is part of a computer system's *memory hierarchy*, which typically consists of the *CPU cache* and *main memory*, *disk* and *disk cache*, and *tertiary storage* such as tape, CDROM, etc. The disk cache functions like any other cache, temporarily holding (*caching*) the contents of the disk address space. Disk caches work because of the *principle of locality* ([Denn68],) which has two parts, *locality by space* and *locality by time*. Locality by time means that the same information tends to be reused, and locality by space means that information near the current locus of reference is also likely to be used.

The most common metric for evaluating a cache is the *miss ratio*, which is the fraction of references not satisfied by the cache. Disk caches typically have 5%-30% miss ratios. The normal mode of operation of a cache is *demand fetch*, by which information is loaded into the cache only when it is requested by the CPU. It is possible to also use a *prefetch algorithm* by which information not requested by the CPU can at times be loaded into the cache. The use of a prefetch algorithm results in the utility of a measure called the *traffic ratio*, which is the ratio of blocks fetched into the cache to the number of requests by the CPU. If there is no prefetching, traffic ratio is equal to the miss ratio; it is the sum of the miss ratio and the *prefetch ratio*. Other aspects of the cache design space include the *replacement algorithm* (which block to replace when a fetch causes the cache to overflow,) the *write policy* (when to transfer modified information to the disk,) and the *block size* (the unit of information storage in the cache.)

These terms are described in more detail in Section A2.

3 Previous Work

While the study of disk caches has not enjoyed the same amount of research effort as the study of CPU caches, there have been a number of studies since its first mention in the literature in 1968 (in [Stev68], as noted in [Smit87].) Some of these studies are based on data collected on “real” systems, while others are purely mathematical models. A number of disk cache designs have been proposed and/or implemented, both as research vehicles and as commercial products, intended for such systems as mainframes and networks of workstations, aimed at both commercial and technical environments. The following discussion outlines some of the more salient work. We discuss the literature in greater detail in Section A3, and present some work not covered here.

3.1 Data-driven Studies

There have been a number of interesting data-driven studies based on traces of I/O references. We will refer to results from some of the following papers in the ensuing discussion of our experiments.

General aspects of disk cache such as its effect on system-level performance, and design considerations of disk cache are presented in depth in [Smit85]. [Frie83], [Effe84], [Oust85] and [Bozi90] find that modest amounts of cache can capture the bulk of file references. [Hosp92] presents an analysis of the break-even point between cache management overhead and decreased access time for cache hits. [Lee88] and [Rich93] find that *metadata*, which implement the structure of file systems, have lower miss ratios than file system data themselves.

Sequentiality and prefetching are studied in [Smit78a] and [Kear89], which find sequentiality common in traces of database references. Variable prefetching algorithms are proposed in [Smit78a], [Thom87], [Ciga88], [Shih90], [Grim93] and [Kroe96]. Generally, basing prefetch decisions on observed sequential reference behavior outperforms pure demand-fetch algorithms, and can perform as well as non-variable prefetch algorithms while prefetching on average, fewer disk blocks. [Kimb96] finds that the distribution of disk blocks on disk has a considerable effect on the performance of prefetch.

Disk caching in the context of distributed systems is discussed in [Thom87], [Maka90], [Munt92] and [Minn93], including such issues as the effect of the consistency protocol on hit ratio and intermediate cache-only servers. [Bolo93] discusses name caching in the X.500 directory system. [Pura96] studies write-only file caches in parallel file systems; delaying writes allows for their optimization.

Several studies propose modifications to existing disk cache buffer schemes including adding frequency-of-use information to LRU ([Robi90], [Kame92], and [Will93],) caching derived data instead of the data themselves ([Ordi93], [Kare94]) delaying writes of dirty data to disk ([Oust85] and [Thom87],) and optimizing write-back operations by "piggy-backing" them onto disk reads ([Bisw93] and [Mill95].)

Six studies extend the concept of trace-driven simulation: [Thek92] presents a method to generate new traces from old, [Levy95] combines separate traces into a single trace, [Wu94] extends single-pass simulation to write-back caches of differing block sizes, and [McNu93] and [Phal95] use data derived from traces, instead of the traces themselves, to drive simulations. [Varm95] derives a synthetic workload from traces to study destaging in a disk array with a non-volatile write cache.

3.2 Analytical Studies

Although no generally-accepted model of disk reference behavior has been developed, there have been a number of analytical studies of disk caching.

General models to analyze the effect of hit ratio and other cache performance parameters on I/O subsystem performance are developed in [Buze83], [Mank83], [Mara83], [Lazo86] and [Bran88]. [Dan90a] and [Dan90b] analyze the complex interactions of disk cache and concurrency control protocol for coupled systems sharing common databases.

Alternatives and improvements to standard LRU disk caches include fixing frequently-used blocks in cache ([Casa89],) "write-only" disk cache ([Solw90],) prefetch ([Palm91], [Ng96], [Solo96],) making allocation decisions based on predicted resulting performance ([Falo91],) and delaying writes ([Cars92].) The warm-up and transient behavior of LRU caches is studied in [Bhid93].

3.3 Implementations and Commercial Products

There are a large number of disk cache implementations, both as experimental research systems and as commercial products, in systems ranging from personal computers to regional information systems.

The first disk cache product was the Memorex 3770, described in [Memo78], which consisted of a 1-to-18-MB cache controlled by a microprocessor, implementing a least-recently-used replacement algorithm. The IBM 3880 Model 13 and 23 storage controllers ([IBM83a], [IBM83b], [IBM85a], [IBM85b]) were IBM's first caching storage controllers for disk I/O. IBM's current products, the 3990 Models 3 and 6 ([IBM87a], [IBM87b], [IBM95]) increase the number of paths through the cache. The 3990-3 offered two new modes of caching operations: "DASD fast write" stores data simultaneously to cache and to non-volatile storage, eliminating disk access time while providing the reliability of a write-through write policy, and "cache fast write," which stores temporary data only to cache. The 3990-6 introduces "record-level caching" in which only the record requested is cached, instead of typical "track caching" in which the record requested and all others following it in the track are cached. For on-line transaction processing workloads, which have little spatial locality in their read requests, caching only the record requested prevents cluttering the cache with unused data ([Arti95].)

Workstation-class computer systems typically function in a distributed computing environment consisting of file servers and clients. An early caching file system for a workstation is described in [Schr85]. The Sprite operating system's file cache ([Nels88]) is notable for its dynamically varying allocation of physical memory between the virtual memory subsystem and the disk cache. In [Welc91], Welch reports that almost half the data generated by applications was never written through to the Sprite server due to its delayed write policy, and that the client read miss ratio was reasonable (35-39%.)

[Dixo84] is a patent for a disk cache with variable prefetch which also dynamically adjusts thresholds controlling the amount of prefetch. In [Kotz90] and [Arun96], prefetching is added to a medium- to large-scale MIMD shared-memory multiprocessor. [Patt93] extends the idea of prefetching in a workstation with the use of application-based hints about which files will be used in the future, and [Patt95] refines the decision to prefetch based on a cost-benefit analysis of prefetching a disk block.

Supercomputers such as those produced by Cray Research, Inc., and its UNICOS 8.0 operating system, provide multiple ways to improve I/O performance, including specifying disk cache block size, capacity and read-ahead (prefetch) capability via library calls, for disk data cached either in main memory or on a solid-state storage device. The *read* system call can also be specified to operate asynchronously: at the time of the system call, the transfer of data to cache is started; later, a synchronizing system call is made to complete the access, if necessary. Data with known poor response to disk cache can be designated to bypass cache. Disk controllers implement read-ahead. I/O optimization in Cray supercomputers is discussed in [Yim95] and [CRAY94]. A similar library of disk cache routines for Intel iPSC/860 and Paragon supercomputers is described in [D'Aze95].

Tokunaga, Hirai and Yamamoto implement a disk controller which treats files differently based on their sequentiality and permanence in [Toku80]. Prefetch is performed on the basis of sequentiality and write policy is chosen based on whether the file is permanent or temporary.

In [Moha95], Mohan implements a method for guaranteeing the detection of partial disk writes in a database system with write-ahead logging and multi-sector pages. Mohan's method is compatible with disk read-write optimizations such as out-of-order sector writes.

Ruemmler and Wilkes provide an overview of SCSI (Small Computer Systems Interconnect) disk drives in [Ruem94], including a review of SCSI disk caching characteristics. SCSI disks typically include a cache with size range from 64 KB to 1 MB, providing read-ahead caching, retrieving and caching data immediately following that which was just requested. Read-aheads may be serviced partially out of the cache or may totally bypass the cache, even though they may partially hit in the cache. Controller configuration commands may segment the cache between several unrelated data items. Caching of write data while maintaining reliability in the face of power outages results in several issues regarding when to report the write as complete. Several techniques exist, including one in which writes will be reported as complete after the cache has been written but before the disk has been written. Or individual writes may be flagged as "must not be immediately-reported," and must be written to disk before reported complete.

4 Methodology

In this section we discuss the methodology we use in this investigation, trace-driven simulation. We also describe the traces we used.

4.1 Trace-driven simulation

This study employs the well-known method of trace-driven simulation. An experiment of this type has two phases: trace collection and simulation. In the trace collection phase, a system is instrumented, generally via software probes, and interesting behavior of the system "recorded." In the second phase, the resulting trace is "played back" and drives a software model of the system under study. For a complete description of the technique of trace-driven simulation and its limitations, see [Smit94].

Table 1. Trace Description

	Trace	Bank	Transport	Telecom	Oil	Monarch	Manu-facture	Time-share
	Entity	Security Pacific Bank	Crowley Maritime	(anon)	Gulf Oil Corp.	Monarch Marking Co.	(anon)	(anon)
	Date Col-lected	3/16/88	7/25/88	10/15/90	1/85	10/87	11/2/90	8/21/90
Hardware	**CPU**	IBM 3090-200 (2)	Hitachi/NAS XL-80	IBM 3090-600J (12)	IBM 3081	IBM 3081	DPS-90 (4 proc.)	DPS-9000 (2 proc.)
Hardware	**Disks**	IBM 3380 (15)	IBM 3380 (20)	IBM 3380/90 (35)	(n/a)	(n/a)	(n/a)	(n/a)
Software	**OS**	MVS-XA 2.1.7	MVS-XA 2.1	MVS-XA 3.1	MVS	MVS-XA	GCOS-8 4000	GCOS-8 4000
Software	**Data-base**	DB2 Rel. 3 Ver. 1.3	DB2 Rel. 3 Ver. 1.3	DB2 Rel. 1 Ver. 2.1	IMS Rel. 3 Ver. 1	IMS Rel. 3 Ver. 1	IDS-II	(none)
Trace Characteristics	**Disk Space Touched**	543.4 MB	123.3 MB	647.9 MB	359.4 MB	180.1 MB	542.6 MB	432.3 MB
Trace Characteristics	**Trace Length**	12.5 MB	16.4 MB	69.0 MB	19.0 MB	76.7 MB	15.0 MB	15.7 MB
Trace Characteristics	**# of Records**	470329	612188	2579815	714461	2878883	562527	585481
Trace Characteristics	**# of Xac-tions**	1563	3650	4477	4447	32762	693	426
Trace Characteristics	**Reads/ Writes**	347.2	55.09	43.92	48.36	26.04	13.77	7.104
Trace Characteristics	**Duration (h:m)**	1:15	1:15	0:15	12:00	7:00	0:35	0:59

Table 2: Transaction Characterization: The "#" rows indicate the number of data points in a particular distribution (e.g., number of complete transaction in a given trace); in the case of the "*average*" column, the "#" entry represents the total number of data points in all of the traces. The "*(percentile)*" lines indicate the percentile within the distribution that the average represents (e.g., the 5.864-second average complete transaction duration for the Transport trace represents the 95th percentile of that distribution.) "*Complete Transactions*" refer to those which conclude with an "*end*" record. "*All transactions*" includes those which had not ended at the end of the trace.

Statistic	Trans-port	Bank	Tele-com	Time-share	Manu-facture	Oil	Monarch	*average*
Transaction Duration (sec.): Complete Transactions								
#	3646	1561	4473	418	691	4445	32759	*47993*
Average (percentile)	5.864 (95.0%)	3.538 (98.5%)	1.886 (92.4%)	107.5 (86.8%)	81.33 (96.6%)	(n/a)	(n/a)	
Median	0.3088	0.2045	0.1610	1.413	4.283	(n/a)	(n/a)	
Std. Dev.	74.78	67.06	17.14	461.3	288.8	(n/a)	(n/a)	
Transaction Duration (sec.): All Transactions								
#	3650	1563	4477	426	693	4447	32762	*48018*
Average (percentile)	8.362 (95.9%)	7.428 (99.1%)	2.533 (94.7%)	160.6 (87.4%)	83.51 (96.8%)	(n/a)	(n/a)	
Median	0.3090	0.2052	0.1615	1.539	4.283	(n/a)	(n/a)	
Std. Dev.	128.1	138.2	30.05	615.8	294.6	(n/a)	(n/a)	
Transaction Size (KB referenced): Complete Transactions								
Average (percentile)	582.4 (97.7%)	749.0 (98.7%)	992.0 (76.2%)	1941 (86.8%)	3100 (93.2%)	618.8 (93.7%)	310.2 (86.4%)	*492.8*
Median	216	48	196	80	104	48	120	
Std. Dev.	5448	2239	13260	7796	15880	11470	1743	
Transaction Size (KB referenced): All Transactions								
Average (percentile)	606.0 (97.7%)	1162 (99.0%)	2198 (85.2%)	5356 (91.5%)	3100 (93.2%)	619.2 (93.7%)	310.4 (86.4%)	*650.4*
Median	136	48	196	84	104	48	120	
Std. Dev.	5556	26790	64480	34670	15970	11470	1783	
Transaction Intensity (KB referenced/sec.): Complete Transactions								
Average (percentile)	736.6 (63.6%)	765.6 (76.1%)	2423 (65.3%)	2150 (90.2%)	544.8 (95.8%)	(n/a)	(n/a)	*1492*
Median	476.0	316.1	1413	90.68	41.80	(n/a)	(n/a)	
Std. Dev.	812.0	952.0	44.08	6600	3421	(n/a)	(n/a)	
Maximum	4032	4860	20840	31250*	31250*	(n/a)	(n/a)	

* Our resolution in timestamps was limited to 128 usec. A transaction which accessed a single disk block only resulted in a measured intensity of 4 K/128usec, or 31250 KB/sec. This represents the upper limit of reference intensity we could measure.

4.2 The data

In this study, we wanted traces of the disk references of the system under consideration. In some cases the system was a database program, and in other cases, an entire timeshared computing environment, including the operating system and user applications. Even the traces from a single type of system (e.g., the database traces) were gathered in different ways. We have three traces of IBM's DB2 database disk references, two of which were gathered using IBM's DB2PM performance monitoring package which records occurrences of DB2 events and passes the records to GTF, IBM's general tracing facility. The other was collected without GTF's considerable overhead by making use of a custom DB2 tracing package written by Ted Messinger, at that time a researcher at IBM's Almaden Research Center. Unlike the DB2 database traces, our IMS database traces were collected by interpreting IMS online system log files created while executing IMS with a trace option enabled. The timeshared system traces from the GCOS operating system, on the other hand, were collected by instrumenting the operating system itself. In the case of the GCOS system traces and the non-DB2PM DB2 trace, it is possible to differentiate between disk references by the operating system and those made on the behalf of users. More detailed trace descriptions are provided in the appendix, Section A4.

DB2 GTF Traces: The first of these traces is called **Bank** and represents both on-line and batch financial applications. Another trace, which we refer to as **Transport**, represents managing bill-of-lading data for a large shipping company. Note that these were collected on production systems, running production workloads. The **Telecom** trace, involving transaction-processing financial applications, was created using production workloads at an IBM customer's site.

IMS traces: The applications represented by the **Oil** and **Monarch** traces include administrative, personnel, payroll, accounts receivable, order entry and inventory control applications. Thus, both of these databases represent production workloads.

GCOS traces: **Manufacture** comprises a primarily transaction-processing workload representing the support of a manufacturing operation, as well as some of its business operations. **Timeshare** represents the timeshared use of a development system for which the workload consists mainly of electronic meetings and mail, and source code editing and compilation.

Our traces are at the logical level —disk blocks are identified by a file identifier and a block offset into that file.

The traces are described in Table 1, which includes the ratio of the number of reads to the number of writes for each trace. The trace format is described in Section A5.

The second phase of the study involved using these traces of disk I/O references to drive models of disk caches of various sizes managed with different policies. In addition to this, however, we also characterized the traces to help us understand the results of our simulations, and allow others interested in using our results to determine the extent to which our traces represent the behavior of their systems.

5 Trace Characterization

5.1 Assumptions

Since our traces come from a number of different environments, we needed to make some assumptions and analogies to facilitate the comparison of results. We assume that a transaction (in the context of a database trace) and a process (in the context of a timesharing system trace) are analogs. Likewise, we assume a database tablespace corresponds to a file in a timeshared system. We also converted references of variable-sized amounts of disk data in **Timeshare** and **Manufacture** to 4-KB disk blocks, to match our other traces, which contained references to 4-KB disk blocks. We discuss these assumptions further in Section A6. When the original traces had sequential references to a single block, the separate references were not coalesced into a single reference.

5.2 Transactions

We characterized the traces in terms of transactions (processes.) For each transaction, we measured its duration (if timestamps were available,) its size (in terms of the volume of data it referenced,) and its *intensity*, which we define to be the ratio of the amount of disk data it referenced to the transaction's duration.

Duration: The average duration of a transaction is shown in Table 2. **Telecom** is notable for including four very long transactions which appear at the beginning of the 15-minute trace, and continue beyond the end of the trace.

Because the raw DB2 traces did not include records which explicitly represent the beginning of a transaction, we detected the beginning of the trace by encountering a reference with a transaction ID we hadn't previously encountered. Therefore, it is probable the long **Telecom** transactions had actually begun prior to the beginning of the trace period.

Disk Blocks Referenced: We also calculated the average, median, and standard deviation of the sizes of transactions in terms of the number of disk blocks they referenced during their duration. As this statistic did not depend on timestamps, we could determine these statistics for **Oil** and **Monarch**.

Intensity: We characterized the transactions in terms of the amount of data they referenced per second.

5.3 Objects

Next we characterized the traces in terms of the files referenced. For each file, if timestamps were available, we measured the interval of time between the time a transaction first referenced it, and the time of that transaction's last access to any file/database (which we called the open duration,) the file's size (in terms of the maximum-offset 4-KByte disk block referenced in it,) and its intensity, which we defined to be the number of disk block requests per microsecond of open duration directed to that file. The results appear as Table 3

We define an *ongoing open* as one in which a file is opened by a transaction which had not ended by the time the trace ended.

Size: We measured weighted averages, medians, and standard deviations for distributions of file size using different weights. We also include the number of unique files referenced, opens of files, references and file blocks for each trace.

We created a distribution corresponding to that seen by an observer picking an arbitrary file from the set of unique files referenced. The GCOS traces, **Timeshare** and **Manufacture**, are notable for the relatively small unique file sizes measured.

We weighted file sizes by the number of times each file was opened in the trace. This distribution corresponds to that seen by an observer picking an arbitrary file from the set of file opens. The distribution for **Bank** is notable for the preponderance of opens of the largest file in that trace. The file, 2.03 GB in size, was opened 711 times, accounting for 11.0% of the opens.

We weighted file sizes by the number of times they were referenced in the trace. This distribution corresponds to that seen by an observer picking an arbitrary file from all references.

In general, the average size of a file referenced was larger than that of a file opened. This is not surprising, since large files have more data to reference. **Bank**, however, is anomalous in that the average size of a referenced file is significantly smaller than the size of the average file opened.

We also weighted file sizes by the number of disk blocks in each file. The resulting distribution corresponds to that seen by an observer picking an arbitrary file block from the files referenced.

We calculated the averages, medians and standard deviations of the lengths of the open durations of the files, the activity (which we define to be the number of references to a file during the time a transaction has it open, and the intensity (disk blocks referenced per unit time.) The results of these measurements appear as Table 4.

5.4 Reference Patterns

5.4.1 Locality

It is when the principle of locality is exhibited in the reference behavior of workloads that caching is able to improve performance. In this section, we present some measures of locality and characterize our traces in terms of these measures.

LRU Miss Ratio

The miss ratio of a cache managed with the *Least Recently Used (LRU)* replacement policy is a good measure of the degree of temporal locality exhibited by a reference string. LRU replacement is also typically used as the "standard" replacement algorithm, against which others are measured. The miss ratio and the traffic ratios are shown in Figure 1; all simulations in this paper present warm start results, calculated from the point at which the largest cache simulated was filled. Table 5 shows the number of references required to reach the warm start point. Figure 1 also shows the miss ratio versus the cache capacity as a percentage of the total amount of data touched. **Bank** , **Timeshare** and **Manufacture** 's minimum miss ratios of more than 20% at a cache size equal to the amount of the data touched represent the theoretical limit to a demand-fetched cache's effectiveness for those traces. We plot the cumulative amount of data touched with respect to the cumulative number of references and percentage of trace length which indicate whether a trace has reached steady state; the curves appear in Figure A1 and Figure A2, respectively. By steady state, we mean that the proportion of new references (first reference to a block within the trace) has become a small percentage of all references. By this measure, none of the **Bank** , **Timeshare** , or **Manufacture** traces reached steady state during its duration. **Transport** , **Oil** , **Telecom** and **Monarch** , on the other hand, have relatively few new references occurring toward the end of the trace. Among the traces, **Bank** is notable in that its miss ratio starts high and drops only slowly with increasing percentage of data in the cache. This implies that, although steady state is reached in the trace, there is little temporal locality, at least for the cache sizes considered and the trace length available.

Figure 1: LRU miss ratios for various disk cache sizes; block size in all cases is 4 KB.

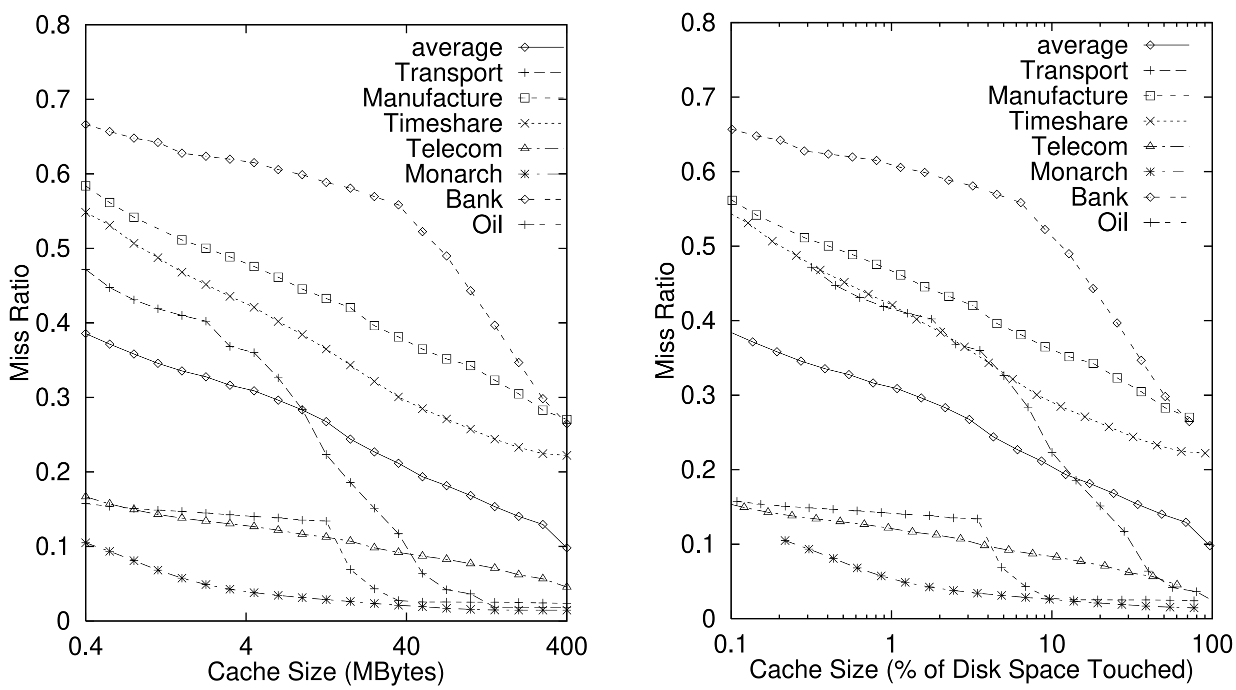


Table 3: File Characterization "Unweighted File Size" represents a distribution of the unique files in each trace. "File Size Weighted by Opens" represents a distribution of all files opened in the trace (that is, a file opened twice would appear twice in the distribution,) "File Size Weighted by References" represents a distribution of the sizes of files corresponding to each file block referenced during the trace, "File Size Weighted by Size" represents a distribution of the sizes of files corresponding to each unique file block in each trace.

Statistic	Transport	Bank	Telecom	Time-share	Manu-facture	Oil	Monarch	*average*
Unweighted File Size (MB)								
# unique	150	82	125	6389	3297	84	214	*10350*
Average (percentile)	85.55 (86.9%)	83.54 (87.8%)	58.51 (80.0%)	0.2400 (93.3%)	3.964 (92.5%)	122.2 (92.9%)	7.262 (86.9%)	*5.236*
Median	0.5000	0.7500	0.6250	0.01172	0.01953	0.5313	0.07813	
Std. Dev.	332.54	304.2	167.1	2.020	16.86	445.6	24.65	
Minimum	0.01172	0.01172	0.01172	0.003916	0.003916	0.003916	0.003916	
Maximum	2303	2081	1064	40.07	91.43	2032	181.94	
File Size Weighted by Opens (MB)								
# opened	23095	6447	51764	11032	6287	8604	136212	*243441*
Average (percentile)	174.6 (79.1%)	487.7 (69.1%)	133.3 (74.8%)	0.2259 (93.5%)	7.430 (86.4%)	141.1 (88.8%)	13.77 (83.5%)	*70.72*
Median	0.6484	148.4	19.34	0.01172	0.03916	0.5430	0.3828	
Std. Dev	403.9	674.1	258.1	1.848	22.38	455.6	34.14	
ave. opened / ave unique	6.326	5.838	2.278	0.9476	1.874	1.155	1.896	
File Size Weighted by References (MB)								
# references	552864	454227	2460032	570214	548721	688362	2540951	*7815371*
Average (percentile)	467.0 (64.3%)	347.3 (76.9%)	386.5 (50.5%)	3.738 (73.1%)	47.98 (49.3%)	148.6 (76.1%)	8.976 (89.1%)	*194.5*
Median	55.72	66.36	326.4	0.5938	36.69	22.18	0.4219	
Std. Dev.	590.9	525.3	333.5	7.363	42.77	320.5	29.65	
ave. referenced / ave. opened	2.675	0.7121	2.899	16.55	6.458	1.053	0.6518	
File Size Weighted by Size (MB)								
# file blocks	167343	1753626	1872454	195231	3344901	2627560	397821	*7998935*
Average (percentile)	1363 (51.2%)	1192 (32.7%)	535.6 (51.8%)	21.46 (49.2%)	75.72 (31.4%)	1747 (41.4%)	90.95 (53.8%)	*1026*
Median	1245	1203	378.7	26.45	94.94	1984	80.26	
Std. Dev	900.4	727.0	386.7	15.46	30.54	491.5	59.23	

Table 4: File Characterization, cont.

Statistic	Transport	Bank	Telecom	Time-share	Manu-facture	Oil	Monarch	*average*
File Open Durations (sec.): Complete Opens								
#	23080	6444	55285	3252	5993	8599	13696	*238849*
Average (percentile)	6.106 (95.6%)	3.958 (98.8%)	1.616 (93.1%)	23.48 (52.2%)	530.9 (64.5%)	(n/a)	(n/a)	*37.36*
Median	0.2927	0.2598	0.1443	21.21	134.7	(n/a)	(n/a)	
Std. Dev.	66.51	68.83	16.69	604.2	668.0	(n/a)	(n/a)	
File Open Durations (sec.): All Opens								
#	28672	6449	55548	11032	6287	8604	136212	*25804*
Average (percentile)	373.4 (81.3%)	5.628 (99.0%)	4.795 (97.2%)	1527 (51.2%)	542.0 (63.7%)	(n/a)	(n/a)	
Median	0.4148	0.4148	0.1459	1435	156.8	(n/a)	(n/a)	
Std. Dev	156.8	97.00	544.9	1272	667.9	(n/a)	(n/a)	
File Reference Activity (number of References): Complete Opens								
Average (percentile)	23.00 (92.9%)	47.49 (98.2%)	21.23 (92.2%)	59.02 (90.0%)	87.30 (91.2%)	79.99 (95.0%)	18.65 (84.1%)	*24.93*
Median	3	3	3	2	3	5	3	
Std. Dev.	370.7	2099	646.9	426.3	705.2	2043	195.8	
File Reference Activity (number of references): All Opens								
Average (percentile)	23.94 (92.9%)	70.46 (98.1%)	45.59 (95.4%)	51.44 (88.5%)	87.28 (91.2%)	80.00 (94.5%)	18.65 (84.1%)	*31.72*
Median	3	3	3	5	3	5	3	
Std. Dev	348.2	2561	4171	484.7	698.2	2043	195.8	
File Reference Activity (number of references per second): Complete Opens								
Average (percentile)	40.17 (80.9%)	76.44 (89.1%)	94.55 (79.8%)	69.00 (96.0%)	11.59 (91.7%)	(n/a)	(n/a)	*73.79*
Median	11.78	12.87	18.30	0.1506	0.0546	(n/a)	(n/a)	
Std. Dev.	119.1	237.7	279.6	590.5	101.4	(n/a)	(n/a)	
Maximum	1860	1822	5208	7812	2604	(n/a)	(n/a)	

Table 5: Warm start statistics

Warm Start Point	Transport	Bank	Telecom	Timeshare	Manufacture	Oil	Monarch
references	140000	150000	500000	200000	150000	220000	450000
(% references)	23.52	33.02	20.32	35.07	27.34	31.96	17.71
(% trace time)	40.64	32.36	26.59	43.93	31.71	(n/a)	(n/a)

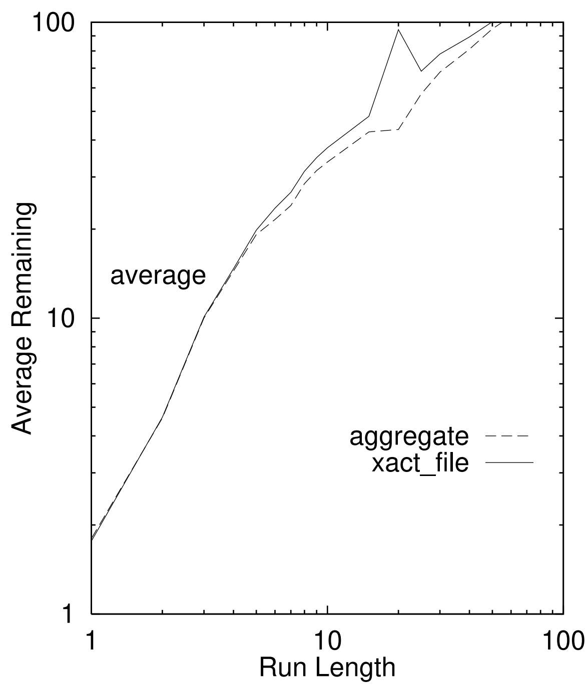
5.4.2 Sequentiality

It is common in both file and database systems to reference data sequentially—entire files are often read or written, and databases are often scanned sequentially for a target record. This section considers sequential reference patterns, presents measures of sequentiality, and then discusses the effect of this characteristic on the effectiveness of disk cache.

Sequential Run Length

When a sequential reference pattern is known or suspected, system performance can often be improved by prefetching into the disk cache blocks ahead of the current point of reference. More generally, we would like to prefetch block $i+1$, at the time of reference to block i, if the probability of referencing block $i+1$ before it is replaced from the cache is greater than some threshold probability P. We note two things about that statement: (a) We don't care if block $i+1$ is referenced immediately after block i, or at some later time, provided it is referenced before it is replaced, and provided it arrives before it is actually referenced. (b) Prefetches have performance costs (disk busy, I/O path busy, memory pollution [Smit78b],) so prefetching a block only makes sense if the expected performance gain exceeds the expected performance cost; this latter concept is captured by the probability P.

Figure 2: Average remaining run length for runs measured on an aggregate and per xact_file (transaction-file) basis. For a point *(x, y)* on the graph, when a run of length *x* has been observed, *y* represents the average of the remaining lengths of runs of *x* and greater.



Because we are interested in whether block $i+1$ is referenced before it is replaced, rather than whether it is referenced immediately after block i, as has been considered previously (e.g., [Smit78a], [Kear89],) we have used various metrics for evaluating and recognizing sequentiality. We measured sequential runs using two methods. The first, *aggregate*, kept run-length statistics on an aggregate basis only—i.e., a reference was sequential if and only if it was logically sequential (in the address space) to the preceding reference in the aggregate address stream (trace.) The second recorded run length statistics on a per-transaction and -file or per-process and -file basis, i.e., a reference was sequential if it was sequential within the reference stream for that single process or transaction to that file. We call this *xact_file*.

We calculated run lengths both ignoring immediate re-references and increasing the length of the run on re-references.

Results

Run length distributions for the various traces appear in Table A2. In Figure 2, we show our computation of the empirical expected remaining run length as a function of the length of the run so far; curves for each of the traces appear as Figure A3. From these figures, we can conclude that the longer the run observed so far, the longer the run is likely to continue, and the more useful prefetching would be.

Sequential Working Set

The problem with both the per-process-file and aggregate basis for determining run length is that neither actually measures the utility of prefetching. What we really want to do is prefetch block $i+1$, on a reference to block i, if block $i+1$ will be referenced before being replaced in the cache; we don't actually care if the reference to $i+1$ is the next reference or won't occur for the next one thousand references. As an aid to determining the utility of this type of prefetching, we compute the miss ratio for both the working set and what we call the *sequential working set*. The *working set* is the set of blocks referenced in the preceding τ time units [Denn68]. We define the sequential working set as all of those blocks in the working set plus all of those blocks sequentially following (in the disk address space) those in the working set. We can compute the miss ratios of both the working set and sequential working set as a function of cache size by varying τ, and computing the average [sequential] working set size and miss ratio for each value of τ. The extent to which the sequential working set miss ratio is lower than the working set miss ratio is a strong indication of the maximum extent to which sequential prefetching can decrease the miss ratio. In particular, this ratio is almost entirely insensitive to the interleaving of references between processes/transactions.

In more mathematical terms, we define the working set of reference string s, at time t, with working set parameter τ, to be:

$$WS(s, t, \tau) = \{ B_{t-\tau} , B_{t-\tau+1} , \ldots , B_{t-1} \} \tag{1}$$

We can define the Sequential Working Set, *SWS*, to be:

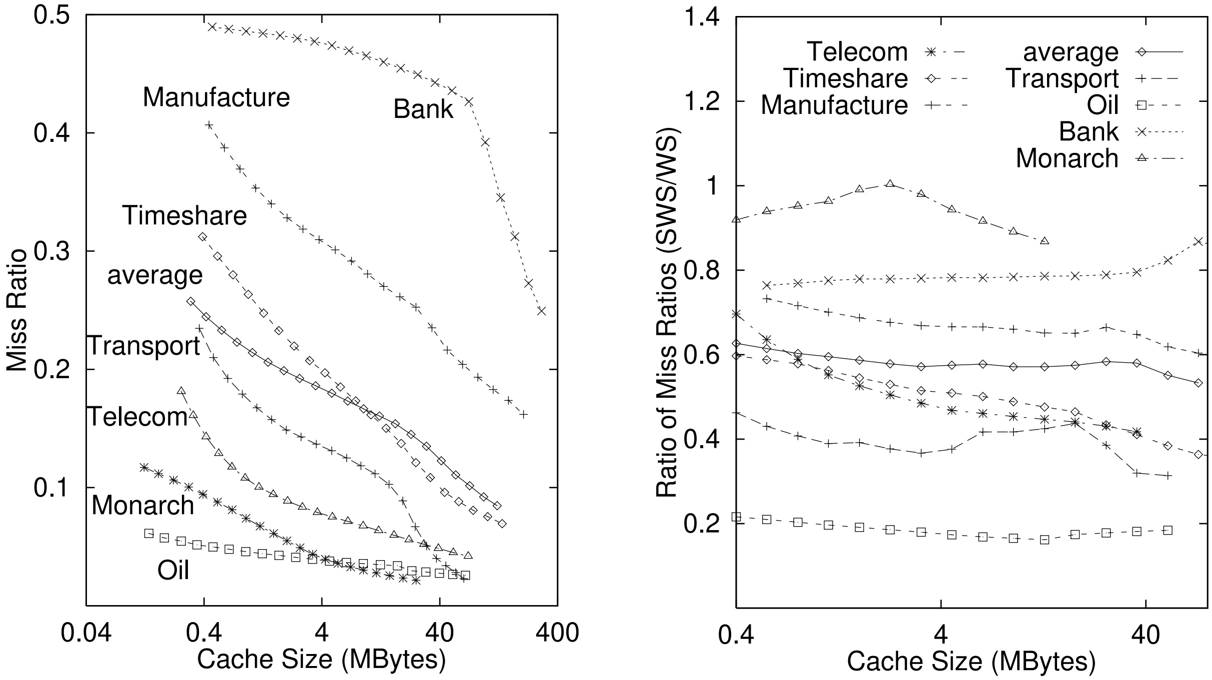
$$SWS(s, t_i, \tau) = \{ (B_{t-\tau}) + 1,\ (B_{t-\tau}) + 1, \ldots,\ (B_{t-1}) + 1 \} \cup WS(s, t_i, \tau) \tag{2}$$

We also tested for loops in the data reference pattern. Unlike instruction traces ([Koba84],) our file and database traces showed few loops.

Results

The Sequential Working Set miss ratio appears in the left in Figure 3 and the ratio of the sequential to regular working set miss ratio appears on the right. (Plots of miss ratio vs. τ appear as Figure A5.) As may be seen, the miss ratio for SWS is around 60% that for normal WS (or LRU); thus the maximum gain achievable from sequential prefetching should be around a 40% decrease in miss ratio.

Figure 3: Left: sequential working set miss ratio. Right: ratio of sequential working set miss ratio to regular working set miss ratio.



6 Disk Cache Management

6.1 Overview

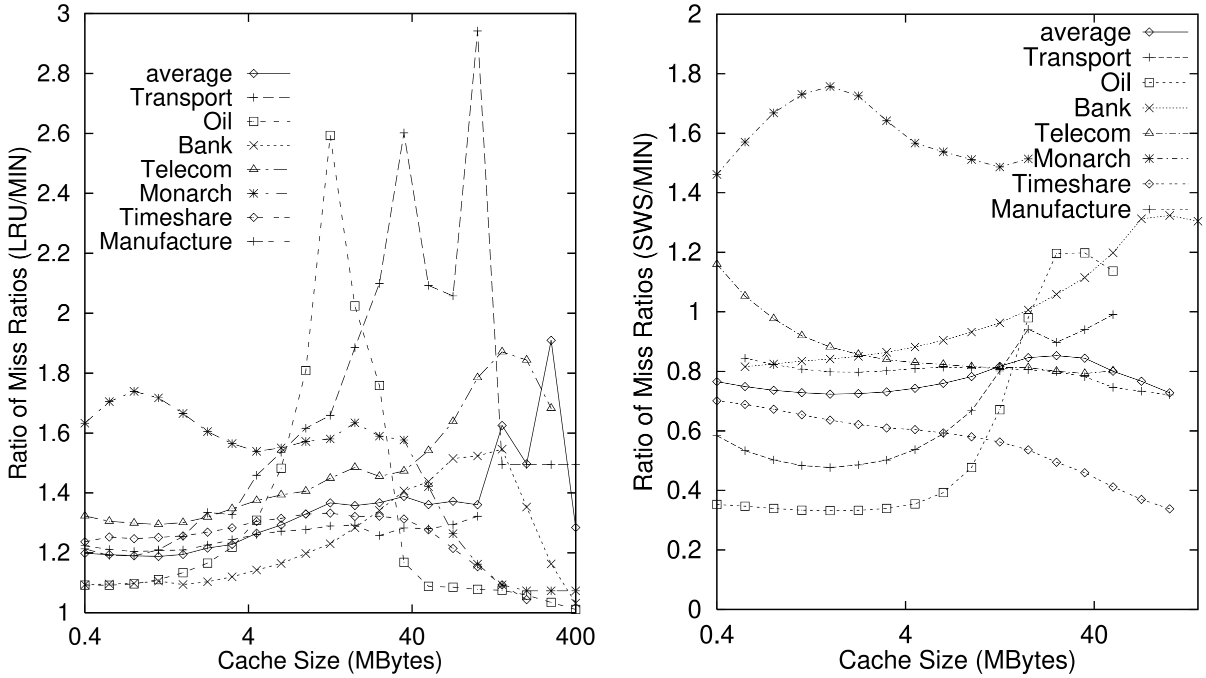
In this section we discuss the metrics we use to compare various buffering schemes, and the relative performance of these schemes under the differing workloads (i.e., traces.)

In most cases, we present our results using a graph of miss ratio versus some parameter of interest, typically the size of the disk cache. We note that while the miss ratio is indicative of performance, it is not a direct measure of system performance. The actual system performance depends only indirectly on the average disk access time; if the system never waits for disk I/O, then the miss ratio is irrelevant to performance. The average disk access time itself is affected not only by the miss ratio, but also by the physical disk access time, the queueing on the disk, the I/O channels, the disk cache, the data transmission rate, etc. We have chosen to use the miss ratio as our figure of merit because it is independent of system configuration, implementation details, and technology. A more detailed study of performance impact (such as was done in [Tse95] for CPU caches) would be needed to translate miss ratio to performance impact. In general, we believe that the average disk access time is closely proportional to miss ratio because of the very large ratio of access time to the disk relative to the disk cache. We note that most other studies of disk caches have also used miss ratio as the figure of merit, which facilitates comparisons between our results and those appearing earlier.

6.2 LRU and MIN Replacement Policies

For purposes of establishing a baseline (standard) level of performance, we performed simulations of caches managed with the LRU replacement policy, and also with the optimal, lookahead MIN (or OPT) replacement policy. Because MIN represents the unrealizable best that a demand-fetch replacement policy can do, it gives the researcher an idea of the scope of improvement possible from modifications of the replacement algorithm. We compare LRU to MIN and Sequential Working Set (SWS) to MIN in Figure 4.

Figure 4: Ratio of LRU to MIN miss ratios and ratio of SWS to MIN miss ratios.



As may be seen in Figure 4, LRU shows a miss ratio around 30% higher than MIN, except for very large cache sizes, for which little replacement occurs. This is consistent with the difference between what can be obtained with the optimal realizable algorithm and the optimal lookahead algorithm [Smit76]. The right side of Figure 4 shows that SWS is around 20% better than MIN—i.e., prefetching is better than optimal replacement. MIN and VMIN miss ratios appear as Figure A6. Working Set miss ratios [Denn68] appear in Figure A7.

6.3 Prefetch

Our good results with SWS suggest further exploration of prefetch algorithms. Here we consider prefetching (a) block(s) (logically) sequentially following the one being referenced, and (b) only at the time of a miss.

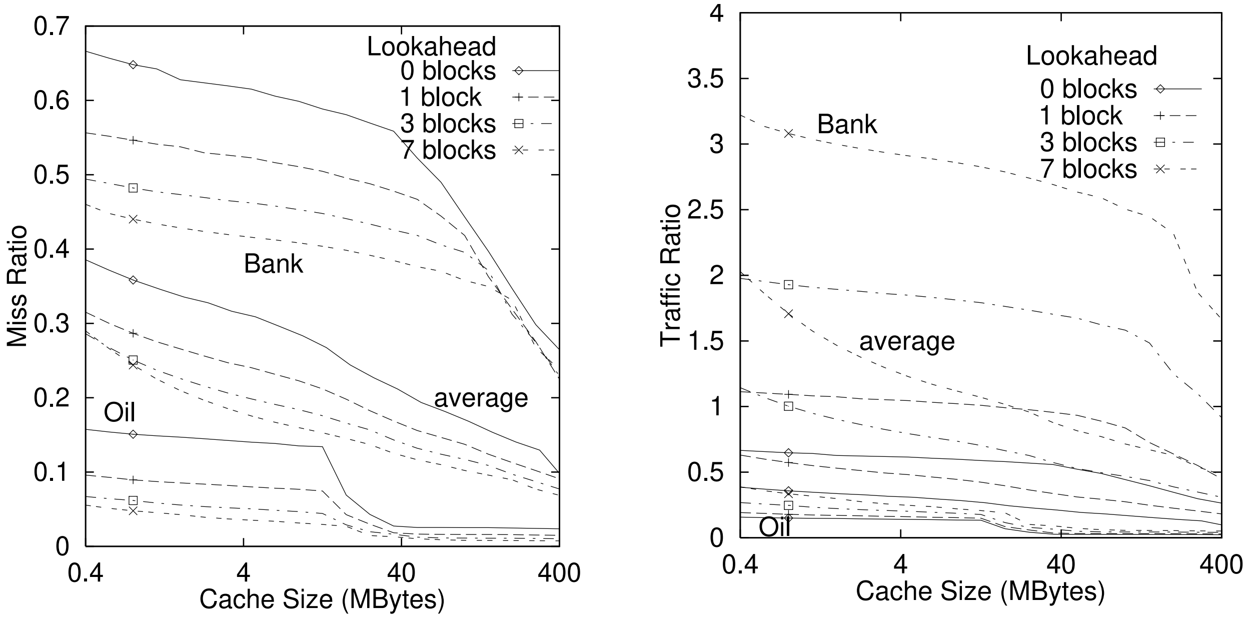
Results

We simulated two general forms of prefetch. Lookahead prefetch fetches a constant number of blocks ahead of the target block at the time of a miss. Figure 5 shows the average miss ratio and traffic ratio for lookahead prefetch of differing numbers of disk blocks. Also shown are the results for the **Bank** and **Oil** traces, which have the maximum and minimum miss and traffic ratios for a given cache configuration, respectively; the curves for all of the traces appear as Figures A9 and A10. Note that the IBM 3380 disk has a track size of about 50 KB, so a 7-block prefetch is slightly more than half a track. From the data shown in Figure 5, we can conclude that a cache block size greater than a track would yield very little improvement in miss ratio while significantly increasing the traffic ratio.

Variable Prefetch

From the results in Figure 5, it would be reasonable to try to prefetch a variable number of blocks, depending on the length of the sequential reference pattern seen. We experiment here with one such scheme, which we refer to as *variable prefetch*. Variable prefetch prefetches k-1 disk blocks when a run of length k (k<5) has been observed thus far, up to a maximum of 4 blocks.

Figure 5: Prefetch miss ratios and traffic ratios, when prefetching a constant number of blocks ahead of the target block at the time of a miss.



This strategy was first suggested in [Smit78a], where it was called *Strategy(0,1,2,3,4)*. In our experiments, run length was detected on a per-process-file (*xact_file*) basis, as in our sequentiality studies of Section 5.4.2. We refer to the strategy as *per-process-file variable prefetch*. Unlike the studies in [Smit78a], in our experiments, prefetch only occurred on misses.

We simulated two types of variable prefetch, both of which ignore immediate re-references to blocks. The first considers a reference part of a run if it is to the disk block following the last one referenced in that run, while the second considers a reference part of a run if its predecessor is in the cache, regardless of whether references are strictly sequential. We refer to the second type as *sequential working set variable prefetch*. In all cases, prefetches were done only on misses.

Results

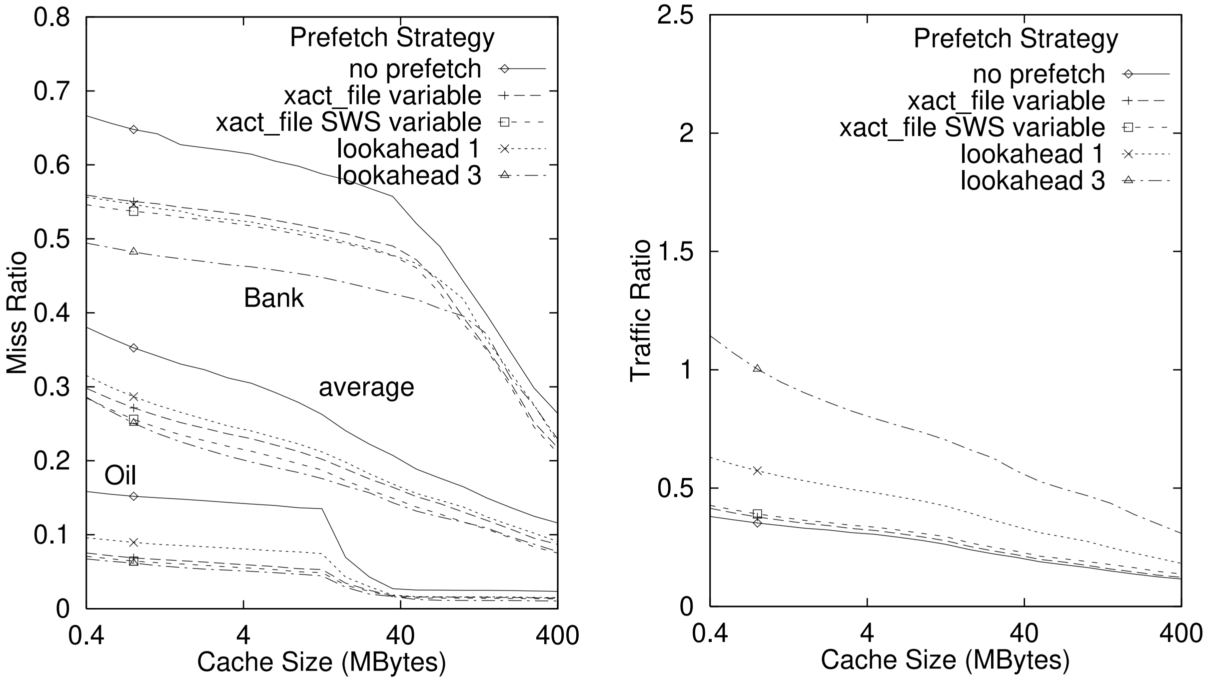
Figure 6 compares miss and traffic ratios for constant lookahead and variable prefetch; miss and traffic ratio curves for all of the traces appear as Figure A11 and Figure A12. As may be seen, variable prefetch has slightly better miss ratios than lookahead 3 and is significantly better than lookahead 1. SWS variable prefetch is slightly better than plain *xact_file* variable prefetch. Both forms of variable prefetch, however, have substantially lower traffic ratios than constant lookahead prefetch, and are accordingly preferable.

Figure 7 shows the average fraction of prefetched blocks that were referenced before they were replaced; per-trace curves appear as Figure A13 and Figure A14. As may be seen, variable prefetch is far more likely to fetch useful blocks, *i.e.*, blocks that will be referenced, than a constant lookahead algorithm.

Implications of Prefetch

Note that in some operating systems, notably Unix, the implementation of prefetch may have an impact on the way in which the file system should be managed. The 4.2 BSD Unix fast file system [McKu84], for example, uses parameters including the rotational speed of the disks and the speed of the CPU to determine the “rotationally optimal” distance between the placement of consecutive disk blocks. Optimizing this distance can result in logically consecutive disk blocks being placed in physically adjacent disk blocks in the same cylinder group, or they may be placed a set number of blocks apart, depending on the expected time to service an interrupt and generate another request. If prefetch is frequently used, however, the system timing may require a different file layout.

Figure 6: Average variable-prefetch miss and traffic ratios for various cache sizes. Prefetches were done only on misses.



6.4 Block Size

Another cache parameter that can be varied to improve performance is that of the transfer size between the disk and the disk cache; we refer to that unit of information as the *block size*. By doubling the block size (from the standard 4 KB,) we should get results similar to prefetch with a constant lookahead of one. The differences are: (a) prefetch always gets the next sequential block, whereas a double block size may result in fetching the preceding (sub)block; (b) an unused prefetched block may be replaced independently of the block that caused it to be prefetched, whereas a larger block must be transferred as a unit; (c) the process requesting a block can restart before any subsequent prefetched blocks have arrived, whereas for the larger block, the transfer isn't complete until the entire block has arrived; (d) the larger block size means smaller tables to manage the cache; (e) a prefetch would only take place if the subsequent block is missing, whereas the double block is always transferred as a unit. On the whole, we would expect that performance should be slightly better for prefetching than for a double block size.

Results

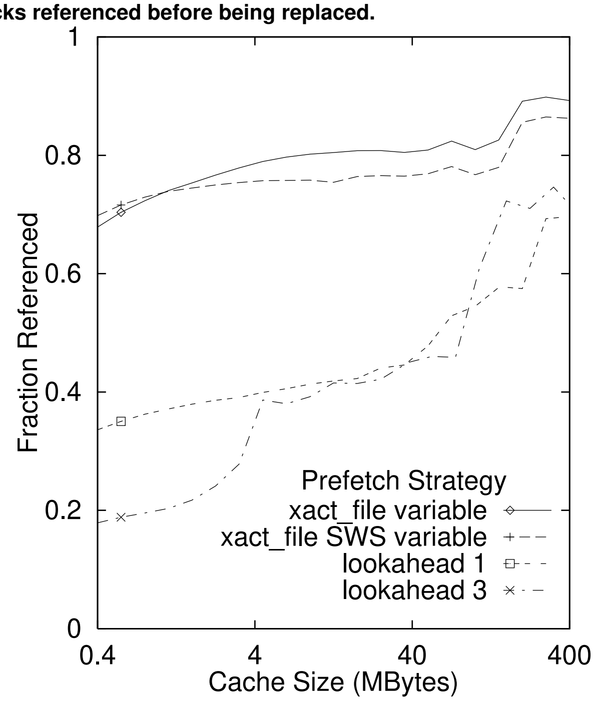
We performed experiments with a variety of cache block sizes; the resulting miss and traffic ratio curves appear as Figure 8; miss and traffic ratio curves for all of the traces appear as Figures A15 and A16, respectively. Variable prefetch results are plotted for comparison. As we can see, miss ratios for constant prefetch are slightly better than for the corresponding increased block size. Performance for the variable prefetch algorithms is considerably better than either. It is particularly worth noting that for larger block sizes, the traffic ratio increases sharply while the miss ratio decreases by a much smaller amount.

6.5 Write Handling

When the CPU writes to the disk, the write can be immediately propagated to the disk (*write-through*,) or the data can be altered in the cache only, and then later copied back to disk (*copy-back* or *write-back*.) A write-through policy maximizes reliability by synchronously propagating writes to the disk as they occur. However, it minimizes performance in the sense that write hits cost the same as write misses, incurring the full disk access cost of seek, latency and transfer times (plus a potential RPS miss delay in some IBM architectures.) A further distinction may be made between write-through caches which employ a *write-allocate* policy and those which employ a *no-write-allocate* policy. Write-allocate policies are useful when dirty data are likely to be referenced before they are replaced, such as in the case of intermediate program files in compilations, and shared data.

For write-back, a disk block write is not propagated to disk until the block is to be replaced. Writes can be avoided entirely if data are rewritten or removed before they are to be replaced. If, however, there is a failure which prevents the flushing of the disk cache to disk before the system is halted, data which were modified but not written to disk (dirty disk blocks) may be lost. As caches increase in size, dirty blocks can reside in the cache for long periods of time before being replaced. In a 1985 study of a Unix system, about 20% of the blocks in a 4-Mbyte cache remained longer than 20 minutes, implying that in the absence of battery backup of the disk cache, system crashes could result in the loss of significant amounts of data [Oust85].

Figure 7: Percentage of cache blocks referenced before being replaced.

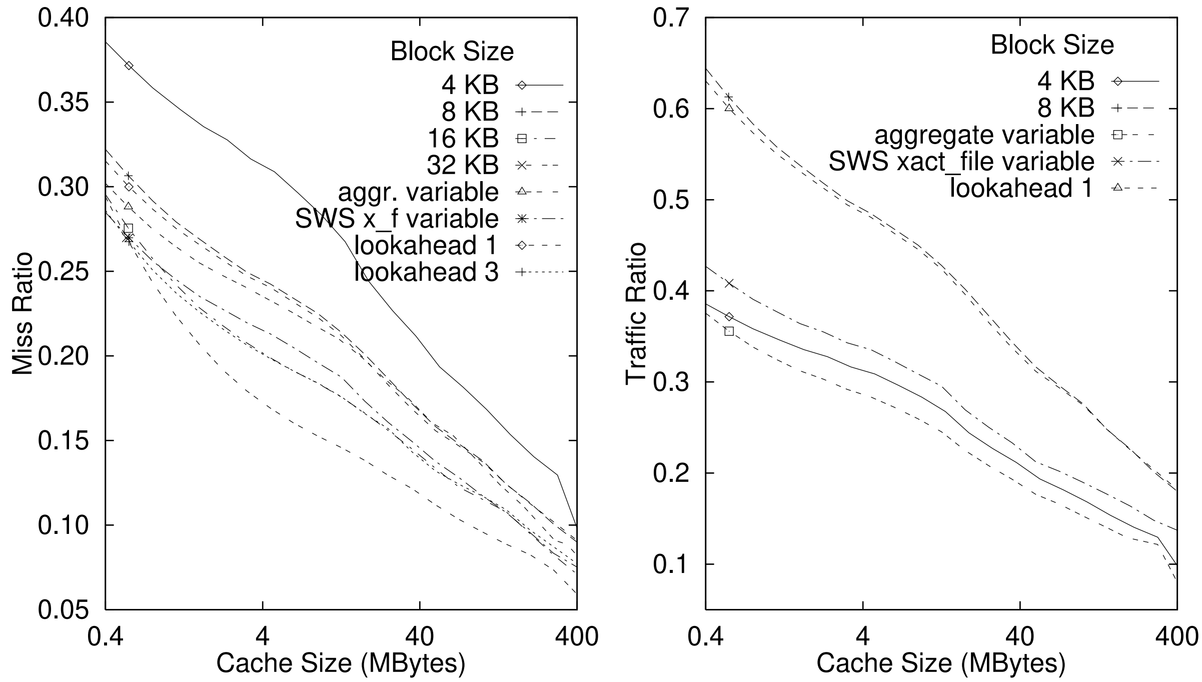


Results

We performed simulations of write-allocate and no-write-allocate policies for a write-through policy and calculated the resulting ratio of read miss ratios; the results appear in Figure 9;read miss ratio curves for all of the traces appear as Figure A17. We see that for a write-through write policy, not allocating a cache buffer on a write has little effect on the number of read misses, although the data written tend to be re-used before they are replaced. The reason is that for most of our traces, writes account for only a small fraction of the references. We compared write-through and write-back write ratios and traffic ratios for each of the traces; the curves appear as Figure A18 and Figure A19. The degree of decrease in the number of writes for each policy varied quite a bit for the traces, though, on average, the number of writes about halved for a write-back policy as opposed to a write-through policy. Implementing a write-through policy increased disk traffic modestly for all of the traces except **Bank** and **Transport**, for which there was negligible effect. We also calculated how many writes there were, for each block that was written in the trace. On average 15% of writes were to blocks written only once, and almost 60% of writes were to blocks written 100 or fewer times. The **Telecom** trace was notable in that many of its writes were to blocks which were written a large number of times.

We also calculated how many writes there were, for each block that was written in the trace. The results appear as Figure A24. On average, 15% of writes were to blocks written only once, and almost 60% of writes were to blocks written 100 or fewer times. The **Telecom** trace was notable in that many of its writes were to blocks which were written a large number of times.

Figure 8: Average miss and traffic ratios of caches with various block sizes. Compare with caches managed with aggregate variable prefetch, per-process-file Sequential Working Set variable prefetch and 1- and 3-block lookahead.



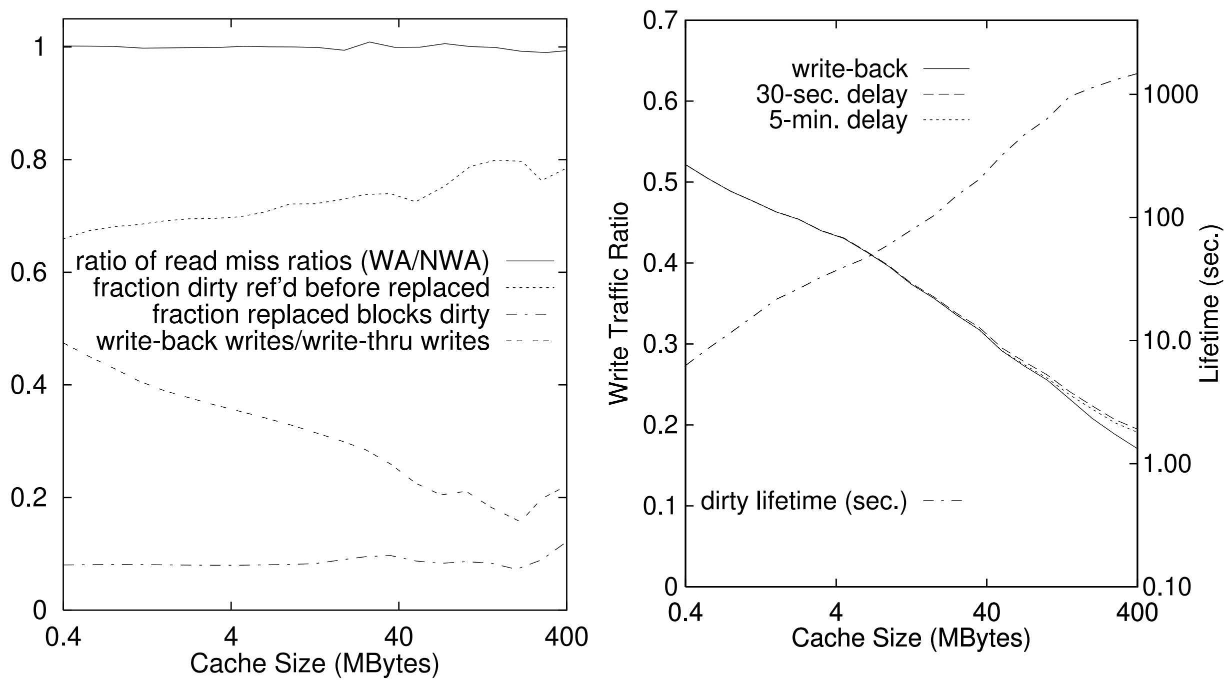
Delayed Writes

An approach which attempts to trade off performance for reliability batches up writes, delaying them until a specific amount of time has passed. The Unix file system buffer implements such a policy, flushing file system buffers every 30 seconds via the *sync* system call [Bach86]. This has the effect of limiting the amount of data lost due to system crashes (assuming no battery backup for the disk cache,) and eliminates the negative impact of writes by performing them asynchronously. Note that a delayed-write policy does not eliminate the reliability problem—it merely reduces it relative to that of a write-back policy.

Results

We performed experiments on those traces which included timestamps: **Bank**, **Telecom**, **Manufacture**, **Timeshare** and **Transport**, simulating 30-second and 5-minute delayed write policies; the curves appear as Figure A20. We found that, on the average, batching up writes and periodically flushing dirty cache buffers to disk had only a small effect on the write traffic, and then only for large cache sizes, as may be seen in Figure 9. One interpretation is that, in a pure write-back policy, most disk blocks were written back within 30 seconds due to replacement. However, when we measured the average duration of time between when a block was written and the time it was replaced, we found that, at large cache sizes, this time exceeded 5 minutes for all of the traces, and exceeded 30 seconds at cache sizes between 0.5 MB and 40 MB; the curves appear as Figure A21 and the average is included in Figure 9. However, only for **Timeshare** and **Manufacture** do these dirty blocks represent more than 5% of all blocks replaced, which explains the small effect on the traffic ratio for the other traces; these curves appear as Figure A23, and a curve of the average dirty lifetime is included in Figure 9. A positive effect of batching up writes to perform multiple writes at a time is to allow them to be written in an optimal manner, rather than requiring a random seek and rotational latency per write when they are performed separately. This is the idea behind log structured file systems.

Figure 9: Ratio of average write-allocate (WA) to no-write-allocate (NWA) read miss ratios, average fraction of dirty cache blocks referenced before they are replaced, average fraction of cache blocks dirty upon replacement and ratio of average write-back writes to write-through writes (left) and delayed write average write traffic ratios (relative to write-through) and average lifetime of dirty cache blocks in seconds (right.) Write-allocate places copy of written-through block in cache; no-write-allocate does not.



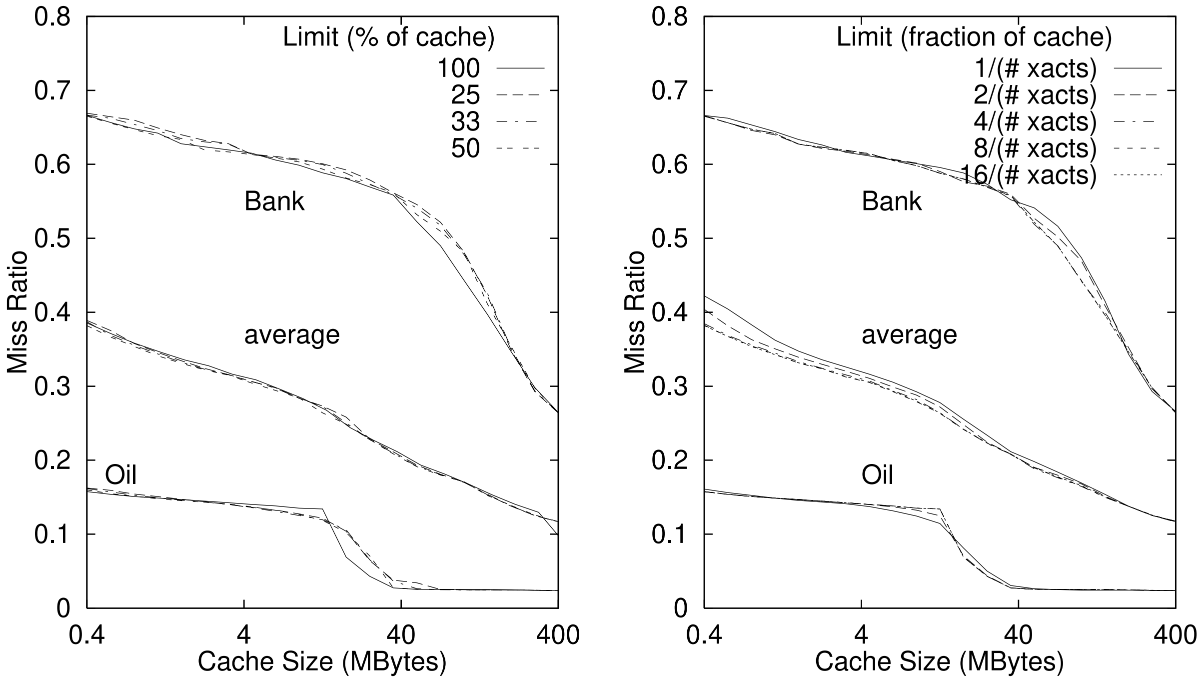
Mixed policy (Write-through/Write-back)

One approach which should minimize the reliability problems with write-back, while still obtaining most of the performance advantages, is to mix write-through and write-back. Write-back should be safest when used for temporary files, and should also have the largest performance impact in that case, since such files are typically written and immediately reread, and then erased. Write-through can be used for permanent files. This approach has been suggested previously ([Toku80].) The problem with this approach is to know which are the temporary files. In MVS and Unix, such files are usually easy to identify. In other operating systems it may not be so easy to determine the reliability requirements for a given disk block.

Results

We performed simulations which implemented the mixed write policy, for those traces (**Timeshare** , **Manufacture** and **Telecom**) for which we could distinguish temporary from permanent; the curves appear as Figure A25. The results show that the traffic ratio resulting from a mixed write policy varied, depending on the trace, between that of a full write-back policy or a full write-through policy. The **Telecom** trace mixed write policy was closer, in terms of traffic ratio, to a write-back cache, while **Timeshare** and **Monarch** ended up with traffic ratios closer to a write-through policy. When we measured the relative proportions of writes to temporary files versus those to permanent files, the reason became obvious: 72.2% of the **Telecom** writes are to temporary files, which are treated in a write-back manner. However, only 7.7% and 16.4% of **Manufacture** and **Timeshare** writes, respectively, were to temporary files. Therefore, for those traces, the mixed policy behaved more like a write-through cache. In general, however, the difference between the write-back and write-through policy was small for the traces we simulated.

Figure 10: Average miss ratios for various caches in which each process or transaction was limited in its use of cache capacity. On the left, the limit is a fixed percentage of the cache capacity. On the right, the fraction limit is *min[N/(# of active transactions),1.0]*.



6.6 Limiting a Transaction's Cache Allocation

A transaction which references large numbers of disk blocks may hinder other transactions' performance if it flushes their disk blocks from the cache. We experimented with a cache which limited the fraction of total cache capacity that a single transaction/process could occupy.

In these simulations, each active transaction maintained an LRU-sorted list of the blocks it had referenced which currently resided in cache, which we refer to as its *occupancy list*. When the length of the occupancy list was smaller than a maximum, expressed in terms of a percentage of the total cache capacity, it serviced a miss by taking the least-recently used block from the global LRU stack. When, however, the length of the transaction's occupancy list reached the maximum, it was forced to replace the block at the bottom of its own occupancy list instead of that at the tail of the global LRU list. In this manner, the transaction was prevented from flushing blocks of other transactions once it had reached a given allocation of cache space. When the transaction ended, its blocks were returned to the global LRU list.

We experimented with both fixed and variable maximum allocations. The fixed maximum allocations were expressed as a fraction of the total cache capacity. The variable allocation experiments limited each transaction to a fraction of the cache equal to $n/$ number of current transactions, where n was a constant number between 1 and 16. If the number of currently active translations rose and caused the limit to fall to a point at which a transaction occupied more than its allocation based on that limit, blocks were removed from its occupancy list and placed at the end of the global LRU stack, where they would be preferentially replaced. On the next miss by that transaction, replacement took place from the bottom of its new occupancy list. Disk blocks shared by multiple active transactions “counted against” only the transaction which faulted the block in first. In the case of the variable maximum allocation scheme, a transaction's allocation could increase due to a decrease in the number of currently active transactions. If a transaction had fewer than its current allocation of buffers at the time of a miss, it replaced a page from the bottom of the global LRU stack and added the page to its occupancy list. In this manner, a form of load sharing was implemented.

Results

We simulated caches for which transactions were limited to a fixed fraction of the cache—25%, 33% and 50% of the total capacity; curves of the average miss ratio for fixed and dynamic limiting caches appear as Figure 10; curves for the individual traces appear as Figure A26 and Figure A27. Our normal LRU curves are labeled 100—a single transaction's blocks may fill the entire cache. Generally, the results of these simulations were disappointing. Limiting transactions' allocations did not consistently outperform a non-limiting cache. Especially disappointing were the results of the variable allocation scheme. For n of 8 and greater the variable scheme miss ratio approached that of the fixed limiting allocation. For n of less than 8, variable allocation resulted in higher miss ratios than the fixed allocation caches at smaller cache capacities.

7 Conclusions and Suggestions for Future Work

In this paper, we presented the results of simulations of disk cache on traces from a variety of workloads, from differing systems. This section presents our major conclusions and suggests directions for future work.

- The main conclusion we reached was that the most effective way to decrease miss ratio was to implement a larger cache. Nothing, including implementing aggressive prefetch, worked as well as merely increasing the size of the cache.
- The traces exhibited varying degrees of sequentiality, as measured by average run length and average remaining run length. We also measured sequentiality in a novel way—the ratio of the miss ratio of the Sequential Working Set to the miss ratio of the Working Set. This relative measure appears to be a good predictor of a workload's performance with the implementation of prefetch.
- Prefetching did yield significant decreases in the miss ratio—around 20% on the average, when prefetching on misses. Sequential working set was quite useful as a way of detecting sequentiality.
- Variable prefetch outperformed constant prefetch in terms of miss ratio, with only a modest increase in disk traffic. Variable prefetch also outperformed caches with increased cache block size.
- Given a write-through write policy, allocating a cache buffer on a write tends to have little effect on the resulting read miss ratio compared to a no-write-allocate write-through write policy.
- Delaying writes and batching them up resulted in no detectable decrease in write traffic, although batching writes may result in decreasing the amount of time required to perform the writes, due to more efficient ordering.
- Implementing a mixed write policy (using a write-back policy for temporary data and a write-through policy for permanent data) increased system reliability over a pure write-back cache by minimizing unrecoverable losses in the event of a power loss in the absence of a battery backup for the cache. It had very little effect on write traffic, however.
- Limiting transactions' cache allocation did not outperform a non-limiting scheme, and in the case of one trace, resulted in a significant decrease in performance.

The previous conclusions lead to the following suggestions for future research:

- The miss ratio experiments presented here need to be extended with detailed timing simulations, in order to better understand the relevant performance tradeoffs.
- Better prefetch algorithms need to be developed, since sequential prefetching seems to be limited to around a 20% (on misses) or around 40% (prefetch at any time.) In particular, sophisticated additions to prefetch such as the pattern recognition of [Kroe96], which prefetches upon recognition of a previously-seen pattern of references, and the cost-benefit analysis of [Patt95], which is used to make prefetch decisions, show promise. They should be investigated further.
- How do new mass storage technologies and file system architectures affect the way that the data are buffered? Log-structured file systems, CD-ROM and flash EEPROM exhibit different characteristics which may lend themselves to new caching techniques.
- It appears from our and others' research that making use of the knowledge of data types (e.g., index versus data, user versus system) in making policy decisions can increase the performance of disk cache. We have investigated this in terms of write policy only. Our traces can be used to investigate this area further.

Acknowledgments

The authors would like to thank Steven Viavant, Ted Messinger, Samuel DeFazio and Rod Schultz for collecting the traces on which the experimental work was based.

References

[Arti95]: H. Pat Artis, “Record Level Caching: Theory and Practice,” *CMG Transactions* (Winter 1995,) pp. 71-76.

[Arun96]: Meenakshi Arunachalam, Alok Choudhary and Brad Rullman, “Implementation and Evaluation of Prefetching in the Intel Paragon Parallel File System,” *Proc. 10th Intl. Parallel Processing Symposium,* (April 1996,) Honolulu, HI, pp. 554-559.

[Bach86]: Maurice J. Bach, *The Design of the UNIX Operating System* , Prentice Hall, 1986.

[Bhid93]: Anupam K. Bhide, Asit Dan and Daniel M. Dias, “A Simple Analysis of the LRU Buffer Policy and Its Relationship to Buffer Warm-Up Transient,” *Proc. 9th Intl. Conf. on Data Engineering* (April 1993,) Vienna, Austria, pp. 125-133.

[Bisw93]: Prabuddha Biswas, K. K. Ramakrishnan, Don Towsley, “Trace Driven Analysis of Write Caching Policies for Disks,” *Proc. ACM SIGMETRICS Conf. on Measurement & Modeling of Computer Systems* , (May 1993,) Santa Clara, CA, pp. 13-23.

[Bolo93]: Jean-Chrysostome Bolot and Hossam Afifi, “Evaluating Caching Schemes for the X.500 Directory System,” *Proc. 13th Intl. Conf. on Distributed Computing Systems,* (May 1993,) Pittsburgh, PA, pp.112-119.

[Bozi90]: G. Bozikian and J. W. Atwood, “CICS LSR Buffer Simulator (CLBS,)” *CMG Transactions* , (Spring 1990,) pp. 17-25.

[Bran88]: Alexandre Brandwajn, “Modeling DASDs and Disk Caches,” *CMG Transactions* , (Fall 1988,) pp. 61-70.

[Buze83]: Jeffrey P. Buzen, “BEST/1 Analysis of the IBM 3880-13 Cached Storage Controller,” *Proc. SHARE 61* (Aug. 1983,) New York, NY, pp. 63-74.

[Cars92]: Scott D. Carson and Sanjeev Setia, “Analysis of the Periodic Update Write Policy For Disk Cache,” *IEEE Trans. on Software Eng., 18, 1* , (January 1992,) pp. 44-54.

[Casa89]: R. I. Casas and K. C. Sevcik, “A Buffer Management Model for Use in Predicting Overall Database System Performance,” *5th Intern. Conf. on Data Engineering* , (Feb. 1989,) Los Angeles, CA, pp. 463-469.

[Ciga88]: John Francis Cigas, *The Design and Evaluation of a Block-level Disk Cache Using Pseudo-files* , (1988,) Ph.D. dissertation, University of California, San Diego.

[CRAY94]: CRAY, *Advanced I/O User’s Guide,* SG-3076 8.0, (1994,) CRAY Research, Inc., Eagan, MN.

[Dan90a]: Asit Dan, Daniel M. Dias and Philip S. Yu, “The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment,” *Proc., 16th Intern. Conf. on Very Large Data Bases* (1990,) Brisbane, Australia, pp. 419-431.

[Dan90b]: Asit Dan, Daniel M. Dias and Philip S. Yu, “Database Buffer Model for the Data Sharing Environment,” *Proc. 6th Intl. Conf. on Data Engineering,* (Feb. 1990,) Los Angeles, CA, pp. 538-544.

[D’Aze95]: E. F. D’Azevedo and C. H. Romine, *EDONIO: Extended Distributed Object Network I/O Library* , Tech. Rept. ORNL/TM-12934, (March, 1995,) Oak Ridge Natl. Lab., Oak Ridge, TN.

[Denn68]: P. J. Denning, “The Working Set Model for Program Behavior,” *Commun. ACM, 11* , (May 1968,) pp. 323-333.

[Dixo84]: Jerry D. Dixon, Gerald A. Marazas, Andrew B. McNeill and Gerald U. Merckel, *Automatic Adjustment of Quantity of Prefetch Data in a Disk Cache Operation,* U.S. Patent 4,489,378.

[Effe84]: Wolfgang Effelsberg and Theo Haerder, “Principles of Database Buffer Management,” *ACM Transac-*

tions on Database Management, 9, 4, (Dec. 1984,) pp. 560-595.

[Falo91]: Christos Faloutsos, Raymond Ng and Timos Sellis, "Predictive Load Control for Flexible Buffer Allocation," *Proc. 17th Intern. Conf. on Very Large Data Bases* (Sept. 1991,) Barcelona, Spain, pp. 265-274.

[Frie83]: Mark B. Friedman, "DASD Access Patterns," *Proc. 1983 CMG Intern. Conf.* (Dec. 1983,) Washington, DC, pp. 51-61.

[Grim93]: Knut Steiner Grimsrud, James K. Archibald and Brent E. Nelson, "Multiple Prefetch Adaptive Disk Caching," *IEEE Trans. on Knowledge and Data Engineering, 5, 1,* (Feb. 1993,) pp 88-103.

[Hosp92]:Andy Hospodor, "Hit Ratio of Caching Disk Buffers," *Proc. Spring CompCon 92* (Feb. 1992,) San Francisco, CA, pp. 427-432.

[IBM83a]: IBM, IBM 3880 Storage Control Model 13 Introduction, GA32-0062-0, (1983,) IBM Corp., Tuscon, AZ.

[IBM83b]: IBM, IBM 3880 Storage Control Model 13 Description, GA32-0067 (1983,) IBM Corp., Tucson, AZ.

[IBM85a]: IBM, IBM 3880 Storage Control Model 23 Introduction, GA32-0082, (1985,) IBM Corp., Tuscon, AZ.

[IBM85b]: IBM, IBM 3880 Storage Control Model 23 Description, GA32-0083, (1985,) IBM Corp., Tuscon, AZ.

[IBM87a]: IBM, IBM 3990 Storage Control Introduction, GA32-0098 (1987,) IBM Corp., Tucson, AZ.

[IBM87b]: IBM, IBM 3990 Storage Control Reference for Models 1, 2, 3, GA32-0099 (1987,) IBM Corp., Tucson, AZ.

[IBM95]: IBM, IBM 3990 Storage Control Reference for Model 6, GA32-0274 (1995,) IBM Corp., Tuscon, AZ.

[Kame92]: Nabil Kamel and Roger King, "Intelligent Database Caching Through the User of Page-Answers and Page-Traces," *ACM Trans. on Database Systems, 17,4* (December 1992,) pp. 601-646.

[Kare94]: Ramakrishna Karedla, J. Spencer Love, Bradley G. Wherry, "Caching Strategies to Improve Disk System Performance," *IEEE Computer*, (March 1994,) pp. 38-46.

[Kear89]: J. P Kearns and S. DeFazio, "Diversity in Database Reference Behavior," *Performance Evaluation Review, 17*, (May 1989,) pp. 11-19.

[Kimb96]: Tracy Kimbrel, Pei Cao, Edward W. Felten, Ann R. Karlin and Kai Li, "Integrated Parallel Prefetching and Caching," *Performance Evaluation Review, 24, 1/Proc. 1996 SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems* (May 1996,) Philadelphia, PA, pp. 262-263.

[Koba84]: Makoto Kobayashi, "Dynamic Characteristics of Loops," *IEEE Trans. on Comp., C33, 2,* (Feb. 1984,) pp 125-132.

[Kotz90]: David F. Kotz and Carla Schlatter Ellis, "Prefetching in File Systems for MIMD Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems, 1, 2*, (Apr. 1990,) pp. 218-230.

[Kroe96]: Thomas M. Kroeger and Darrell D. E. Long, "Predicting File System Actions from Prior Events," *Proc. 1996 Winter USENIX Tech. Conf.* (Jan. 1996,) San Diego, CA, pp. 319-328.

[Lazo86]: Edward D. Lazowska, John Zahorjan, David R. Cheriton and Willy Zwaenepoel, "File Access Performance of Diskless Workstations," *ACM Trans. on Computer Systems 4,3*, (Aug. 1986,) pp. 238-268.

[Lee88]: T. Paul Lee and Rebecca Wang, "A Performance Study on UNIX Disk I/O Reference Trace," *Proc., CMG International Conf. on Management and Performance Evaluation of Computer Systems* (Dec. 1988,) Dallas, TX, pp. 121-127.

[Levy95]: Hanoch Levy and Robert J. T. Morris, "Exact Analysis of Bernoulli Superposition of Streams Into a Least Recently Used Cache," *IEEE Trans. on Software Eng., 21, 8*, (Aug. 1995,) pp. 682-688.

[Maka90]: Dwight J. Makaroff and Dr. Derek L. Eager, "Disk Cache Performance for Distributed Systems," *Proc. 10th Intern. Conf. on Distributed Computing Systems* (May 1990,) Paris, France, IEEE, pp. 212-219.

[Mank83]: P. S. Mankekar and C. A. Milligan, "Performance Prediction and Validation of Interacting Multiple Subsystems in Skew-Loaded Cached DASD," *Proc. CMG XIV* (1983,) Washington, DC, pp. 383-387.

[Mara83]: G. A. Marazas, Alvin M. Blum and Edward A. MacNair, "A Research Queuing Package (RESQ) Model of A Transaction Processing System with DASD Cache," *IBM Research Report RC 10123*, (Aug. 1983).

[McKu84]: Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems 2, 3* (Aug. 1984,) pp. 181-197.

[McNu93]: Bruce McNutt, "A Simple Statistical Model of Cache Reference Locality, and its Application to Cache Planning, Measurement and Control," *CMG Transactions* (Winter 1993,) pp. 13-21.

[Memo78]: Memorex, *3770 Disc Cache Product Description Manua* l, Pub. No. 3770/00-00, (1978,) Memorex Corp., Santa Clara, CA.

[Mill95]: Douglas W. Miller and D. T. Harper III, "Performance Analysis of Disk Cache Write Policies," *Microprocessors and Microsystems, 19,3,* (Apr. 1995,) pp. 121-130.

[Minn93]: Ronald G. Minnich, "The AutoCacher: A File Cache Which Operates at the NFS Level," *Proc. 1993 Winter USENIX Tech. Conf.* (Jan. 1993,) San Diego, pp. 77-83.

[Moha95]: C. Mohan, "Disk Read-Write Optimization and Data Integrity in Transaction Systems Using Write-Ahead Logging," *Proc. 11th Annual Conf. on Data Engineering* (March 1995,) Taipei, Taiwan, pp. 324-331.

[Munt92]: D. Muntz and P. Honeyman, "Multi-level Caching in Distributed File Systems—or—Your Cache ain't nuthin' but trash," *Proc. 1992 Winter USENIX Tech. Conf.* (Jan. 1992,) San Francisco, CA, pp. 305-313.

[Nels88]: Michael N. Nelson, Brent B. Welch and John K. Ousterhout, "Caching in the Sprite Network File System," *ACM Trans. on Computer Systems 6,1*, (Feb. 1988,) pp. 134-154.

[Ng96]: Raymond T. Ng and Jinhai Yang, "An Analysis of Buffer Sharing and Prefetching Techniques for Multimedia Systems," *ACM Multimedia Systems 4,* *2,* (1996,) pp. 55-69.

[Ordi93]: Joann J. Ordille and Barton P. Miller, "Distributed Active Catalogs and Meta-Data Caching in Descriptive Name Services," *Proc. 13th Intl. Conf. on Distributed Computing Systems,* Pittsburgh, PA (May 1993), pp. 120-129.

[Oust85]: John K. Ousterhout, Herve' Da Costa, David Harrison, John A. Kunze, Mike Kupfer and James G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proc. 10th Symp. on Operating System Principles* (Dec. 1985,) Orcas Island, WA, pp. 15-24.

[Palm91]: Mark Palmer and Stanley B. Zdonik, "Fido: A Cache That Learns to Fetch," *Proc. of the 17th Intern. Conf. on Very Large Databases* (Sept. 1991,) Barcelona, Spain, pp. 255-264.

[Patt93]: R Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan, "A Status Report on Research in Transparent Informed Prefetching," *Operating Systems Review, 27, 2,* (April 1993,) pp. 21-34.

[Patt95]: R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky and Jim Zalenka, *Informed Prefetching and Caching*, Carnegie Mellon University Tech. Rept. CMU-CS-95-134 (May 1195,) Carnegie Mellon University, Pittsburgh, PA.

[Phal95]: Vidyadhar Phalke and B. Gopinath, "Program Modelling via Inter-Reference Gaps and Applications," *Proc. 3rd Intl. Conf. on Modeling, Analysis and Simulation of Computers and Telecommunications Systems*, Durham, NC, (Jan. 1995,) pp. 212-216.

[Pura96]: Apratim Purakayastha, Carla Schlatter Ellis and David Kotz, "ENWRICH: A Compute-Processor Write Caching Scheme for Parallel File Systems," *Proc. 4th Annual Workshop on I/O in Parallel and Distributed Systems* (May 1996,) pp. 55-68.

[Rich93]: Kathy J. Richardson and Michael J. Flynn, "Strategies to Improve I/O Cache Performance," *Proc. 26th Annual Hawaii Intern. Conf. on System Sciences*, (1983,) Hawaii, pp. 31-39.

[Robi90]: John T. Robinson and Murthy B. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. ACM SIGMETRICS Conf. on Measurement & Modeling of Computer Systems* (May 1990,) Boulder, CO, pp. 134-142.

[Ruem94] Chris Ruemmler and John Wilkes, "An Introduction to Disk Drive Modeling," *IEEE Computer, 27, 3*, (March 1994,) pp. 17-28.

[Schr85]: Michael D. Schroeder, David K. Gifford and Roger M. Needham, "A Caching File System for a Programmer's Workstation," *Operating Systems Review 19, 5*, (Dec. 1985,) pp. 25-34.

[Shih90]: F. Warren Shih, Tze-Ching Lee, Shauchi Ong, "A File-Based Adaptive Prefetch Caching Design," *Proc. 1990 IEEE Intl. Conf. on Computer Design: VLSI in Computers and Processors* (Sept. 1990,) Cambridge, MA, pp. 463-466.

[Smit76]: A. J. Smith, "A Modified Working Set Paging Algorithm," *IEEE Trans. on Comp., C 25, 9*, (Sept. 1976,) pp. 907-914.

[Smit78a]: A. J. Smith, "Sequentiality and Prefetching in Database Systems," *ACM Transactions on Database Systems 3, 3*, (Sept. 1978,) pp. 223-247.

[Smit78b]: Alan Jay Smith, "On the Effectiveness of Buffered and Multiple Arm Disks," *Proc. 5th Annual Symp. on Comp. Architecture*, (Apr. 1978,) pp. 242-248.

[Smit85]: A. J. Smith, "Disk Cache—Miss Ratio Analysis and Design Considerations," *ACM Transactions on Computer Systems 3, 3*, (Aug. 1985,) pp. 161-203.

[Smit87]: A. J. Smith, "Remark on 'Disk Cache—Miss Ratio Analysis and Design Considerations'," *ACM Trans. on Computer Systems, 5, 2*, (Feb. 1987,) p. 93.

[Smit94]: Alan Jay Smith, "Trace Driven Simulation in Research on Computer Architecture and Operating Systems," *Proc. Conf. on New Directions in Simulation for Manufacturing and Communications* (Aug. 1994,) Tokyo, Japan, ed. Morito, Sakasegawa, Yoneda, Fushimi, Nakano, pp. 43-49.

[Solo96]: Valery Soloviev, "Prefetching in Segmented Disk Cache for Multi-Disk Systems," *Proc. 4th Ann. Workshop on I/O in Parallel and Distributed Systems* (May 1996,) Philadelphia, PA, pp. 69-82.

[Solw90]: Jon A. Solworth and Cyril U. Orji, "Write-Only Disk Caches," *Proc. Intl. Conf. of ACM SIGMOD* (May 1990,) Atlantic City, NJ, pp. 123-132.

[Stev68]: D. A. Stevenson and W. H. Vermillion, "Core Storage as a Slave Memory for Disk Storage Devices," *Proc. IFIP '68*, (1968,) pp. F86-F91.

[Thek92]: Chandramohan A. Thekkath, John Wilkes and Edward D. Lazowska, *Techniques for File System Simulation*, Hewlett Packard Laboratories Tech. Rept. HPL-92-131 (Sept. 1992,) Hewlett Packard Corp.

[Thom87]: James Gordon Thompson, *Efficient Analysis of Caching Systems*, UC Berkeley Tech. Rept. UCB/CSB 87/374, (Oct. 1987,) Univ. of Calif. at Berkeley.

[Toku80]: T. Tokunaga, Y. Hirai and S. Yamamoto, "Integrated Disk Cache System with File Adaptive Control," *Proc. IEEE Computer Society Conf. '80,* (Wash. DC., Sept. 1980,) pp. 412-416.

[Tse95]: John Tse and Alan Jay Smith, "Performance Evaluation of Cache Prefetch Implementation," U. C. Berkeley Tech. Rept. UCB/CSD-95-873, June, 1995, submitted for publication

[Varm95]: Anujan Varma and Quinn Jacobson, "Destage Algorithms for Disk Arrays with Non-Volatile Caches," *Proc. 22nd Intl. Symp. on Computer Architecture,* (June 1995,) Santa Margherita Ligure, Italy, pp. 83-95.

[Welc91]: Brent B. Welch, "Measured Performance of Caching in the Sprite Network File System," *Computing Systems*, 3, 4, (Summer 1991,) pp. 315-341.

[Will93]: D. L. Willick, D. L. Eager and R. B. Bunt, "Disk Cache Replacement Policies for Network Fileservers," *Proc. 13th Intl. Conf. on Distributed Computing Sys.,* (May 1993,) Pittsburgh, PA, pp. 2-11.

[Wu94]: Yuguang Wu, "Evaluation of Write-back Caches for Multiple Block-Sizes," *Proc. 2nd Intl. Workshop on Modeling, Analysis and Simulation of Computers and Telecommunications Systems*, (Jan.-Feb. 1994,) Durham, NC, pp. 57-61.

[Yim95]: Emily Yim, "Fortran I/O Optimization," *NERSC Buffer, 19, 12,* (Dec. 1995,) Natl. Energy Research Supercomputer Center, Livermore, CA. pp. 14-23.

A1 Motivation for Disk Cache

Until a dense, inexpensive, fast, nonvolatile memory is developed, computer systems will continue to implement storage as a hierarchy of memory devices, with price per bit, access speed, and volume per bit generally decreasing with increasing distance from the CPU.

The classic hierarchy consists of 5 levels: registers in the CPU, CPU cache memory, main memory, disk memory and mass storage. The ratio of the amount of time needed to access two adjacent levels is called the *access gap*. Typically the largest access gap in a computer system is the ratio of the speeds of main memory and disk memory, which is on the order of 10^6. And while the access gaps between the other levels of the memory hierarchy are tending to remain more or less constant, the gap between main memory and disk storage continues to increase [Katz89].

Since disk memory is the first nonvolatile memory in the hierarchy, and since some data (both input and output) are too large to fit in any higher level, it is common for programs to transfer large volumes of data between disk and main memory. And although main memory continues to increase in size quickly, as CPU's increase in speed, users are taking advantage of increased computing power by operating on ever-larger volumes of data, continuing to stress the limits of amounts of data which can be memory-resident, and continue to be forced to store data on disk. Thus, we expect disk access time to remain a limiting factor in computer system performance.

While there are several ways to deal with the large memory-disk access gap (examples being multiprogramming and buffering of disk I/O in applications,) the addition of another level in the storage hierarchy between main memory and disk called a *disk cache* has become common in computer systems ranging from departmental mainframes to personal computers.

A2 Terms and Concepts

The principle of locality: The principle of locality can be thought of in terms of *temporal locality* and *spatial locality*. Temporal locality is the tendency for data used in the recent past to be used again in the near future. Spatial locality is the tendency for data references to be unevenly distributed in address space: data close in address are more likely to be used together.

Access time: Memories have a cost associated with information storage or retrieval. The *access time* is typically broken down further into two components: a fixed *latency* and a *transfer time* which depends on the size of the transfer and the bandwidth of the medium involved. The relative sizes of the latency and bandwidth vary: for semiconductor memories such as those which make up the main memory, they may be considered to be of the same order of magnitude for typical data transfers. Disk access' latency component consists of two components, *seek time*, the time it takes for the head to move radially to the correct cylinder, and *rotational latency*, the time until the desired data rotate below the head. For transaction processing I/O workloads, disk access times tend to be less sensitive to the amount of data transferred than for scientific workloads [Katz89].

Miss/hit ratio: A disk cache is a fast, small memory located between the CPU and the magnetic disks which holds a subset of the information stored on disk. When the CPU requests disk data, the cache is first accessed to see if it contains the desired data. If so, a *hit* has occurred, and the data can be provided from the cache at a much lower access time than that of the disk. If the data are not in the cache, a *miss* has occurred, and the disk must be accessed. The fraction of requests which require a disk access is referred to as the *miss ratio*. The lower the miss ratio, the fewer disk accesses are necessary, resulting in a lower average access time, and higher performance. Similarly, the *hit ratio* is the fraction of requests which can be serviced out of the cache.

Replacement policy: When a disk cache miss occurs, the desired data are copied into the cache. This operation displaces some data previously in the cache. The *replacement policy* identifies the victim data. A replacement policy tries to determine which of the cached data are least likely to be re-used in the near future. The decision is often based on past history; some examples of policies which use past history are the Working Set replacement policy and the Least Recently Used replacement policy (both of which are described in Section A7.) An important class of replacement policies are based on knowledge of the future, and are used to determine the theoretical limit of a cache's performance. An example of a replacement policy based on future knowledge is called Optimum, and is also known as MIN.

Write policy: The way writes are handled can have a major impact on the performance of a disk cache. *Write policies* typically trade off reliability for speed: as the cache is usually implemented with volatile semiconductor memory, a power interruption between the time the data are written to cache and when they are written to disk can cause a loss of modified data.

A *Write-through* policy maximizes reliability by forcing writes to the disk immediately, typically synchronously, at the cost of the disk access time. With this write policy, writes cost as much in terms of time as cache misses, so this approach minimizes the write performance of the cache. With a write-through policy, a further policy decision must be made regarding whether or not to allocate cache storage to the written data. A *write-allocate* policy stores the written data in the cache as well as writing it to the disk, assuming that it will be read soon. A *no-write-allocate* policy bypasses the cache entirely, leaving the cache to store only those data which have been read.

A *Write-back* policy maximizes performance by delaying writes as long as possible until data which have been modified in the cache but not written to disk (so-called *dirty blocks*) are to be replaced by new data on a miss. The writes may be made asynchronous to have little effect on I/O performance. Physical disk writes may be avoided entirely if the data are written in cache multiple times before they are replaced.

Some *mixed write* policies choose a performance/reliability trade-off between those of the previous two. Writes to disk may be performed asynchronously and may be delayed up to a specified maximum amount of time.

Block size: The size of the minimum amount of data transferred between the disk and the cache is called the *cache block size*. In some caches, this is *not* equal to the minimum amount of data which can be transferred between the CPU and cache. The cache block size may also be larger than the smallest allocatable volume of disk storage.

Fetch policy: Data need not be loaded into the cache only after having been requested. While a *demand fetch policy* loads data into the cache from disk only on the miss, a *prefetch policy* loads data into the cache assuming that they will soon be referenced, either synchronously with the loading of data due to a miss, or asynchronously.

A3 Previous Work

In this section we discuss in greater detail the literature we reviewed in Section 3, as well as discussing literature not covered in the text.

A3.1 Data-driven Studies

The following seven papers concern general aspects of disk cache such as its effect on system-level performance, and design considerations of disk cache.

In [Smit85], Smith performs the first in-depth analysis of disk cache using trace-driven simulation. Examining the effect of capacity on miss ratio, Smith finds that a modest cache, 2-8 MB, captures the bulk of references. Considering the location of cache, Smith recommends placing the cache at the main memory in order to minimize overhead. If that is not practical for commercial or technical reasons, Smith recommends caching at the storage controller. Smith also finds a single-track disk cache block size to be optimal, since larger block sizes can be effectively implemented via prefetching. Analyzing the behavior of the various types of data referenced, Smith finds little locality in paging data, mixed results for the effectiveness of buffering batch versus buffering system data sets, and the suggestion of a great deal of sequentiality in the batch and temporary file reference patterns. Smith's analysis also suggests the usefulness of dynamically turning on or off caching on a per-device basis. On the qualitative side, Smith recommends that storage controllers implementing disk cache must have multiple data paths to operate effectively. Based on performance/reliability trade-offs, Smith suggests that a write-back policy be implemented flexibly for those data which can easily be recreated and a write-through policy for those which require higher reliability. Smith also discusses the cache consistency problem, error recovery in the face of system failure, trade-offs between software and hardware control of the cache, and the implications of disk cache on operating systems.

Hospodor develops a model for determining the point at which disk cache management overhead outweighs the decrease in access time due to hit ratio and the difference in access time in [Hosp92]. The author bases the resulting function on measured access time overhead of cache misses, and finds that no improvement in access time performance will be found unless a hit ratio of at least 14% can be maintained.

In [Frie83], the researcher presents cache performance statistics from medium-sized IBM-compatible computer systems with disk cache, and qualitatively analyzes disk access methods' behavior in a cached environment. Friedman concludes that, with the exception of hashed access methods, file organizations which minimize the number of I/Os necessary to access data tend to exhibit locality to a great degree and perform well in the presence of a disk cache.

Makaroff and Eager use traces of disk block reads and writes from a single-user Unix workstation to study the performance of disk cache in [Maka90]. The authors simulate a distributed system consisting of a single file server and up to ten client workstations to determine miss ratios for various sizes of the client and/or server cache(s). The authors discuss the tradeoffs involved in minimizing both the overall cache miss ratio and server congestion.

Treiber and Menon consider the performance of cache in the context of a Redundant Array of Inexpensive Disks in [Trei95]. The authors use long traces of production DB2/MVS I/O activity, finding performance benefits in allocating the majority of the disk cache to reads (the traces were read-dominated,) ordering the blocks read and written by destage (writing dirty cache blocks to disk) to minimize the disk rotation, and adding disk segmented buffers which can perform prefetch. Turning to large caches, the authors find performance benefits in retaining sequentially accessed blocks, despite the conventional wisdom that sequentially accessed blocks tend not to be re-used.

In [Bozi90], the CICS database and the VSAM Local Shared Resource (LSR) disk caches are simulated using traces from an IBM electronic mailing system to determine the best allocation between LSR's pools of different-sized buffers.

Traces of reference behavior are used to drive simulations of the HP3000 disk cache in [Busc85]. Unfortunately, most of the effort in the study is devoted to using the resulting miss ratios to drive analytical models of the HP3000 system to calculate system performance in terms of throughput, and very little miss ratio data are presented. The assumptions the authors make to calculate the performance are system-specific, moreover, and the resulting numbers are relevant only to the enterprise under study.

The next six papers study sequentiality; four propose prefetch schemes.

Using traces of supervisor calls of MVT/Hasp, vintage 1971-1972, Smith studies the effectiveness of disks with multiple arms and those with buffers in [Smit78]. He concludes that disk seeks can almost entirely be eliminated with three disk arms and that three cylinder-sized buffers can achieve hit ratios of around 96% and can eliminate not only seek time, but latency as well.

Kearns and DeFazio study the disk access behavior of an IMS hierarchical database in the context of a medium-sized IBM-compatible system in [Kear89]. The authors look at sequentiality and locality at the level of individual transactions or individual databases, finding that the degrees of sequentiality and locality exhibited by the transactions/databases was highly skewed, but tended to be quite stable over the period traced.

Shih, Lee and Ong study a file-based variable prefetch policy in [Shih90]. A prefetch status table stores statistics representing the observed sequentialities of files during program execution. A table entry consists of a 3-bit hit/miss history for the last three references to disk blocks in the file, the file ID, and access pointers for the file. On each access to a prefetched block, the associated history is updated (accesses to demand-fetched blocks do not modify the history.) While the authors call the prefetch policy file-based, it corresponds to the prefetch method we call *xact_file*, since a separate entry is created per file-use. The amount to be prefetch is based on the current hit/miss history state (the authors do not indicate whether prefetching is performed only on misses, or on any access.) Using traces of personal computer applications to drive a simulator, the authors find reasonable decrease in average disk I/O time over demand fetch, and performance close to a static prefetch amount of two or three, depending on workload.

Grimsrud, Archibald and Nelson present a variable prefetch strategy in [Grim93]. For each cache block, an entry in the “adaptive table” records one or more pairs consisting of a predicted next disk block and an associated confidence level. The table is dynamically updated on each cache access. Depending on the confidence level and other operating parameters, the cache initiates a prefetch of the predicted block(s). The authors find a substantial decrease in average disk access time using traces from several Unix-based installations.

Kroeger and Long present a variable prefetch algorithm inspired by the data compression technique *Prediction by Partial Match* in [Kroe96]. Using traces of Unix *open* user-level system calls, the authors model a disk cache which caches whole files. Therefore, prefetch occurs in the context of which whole file to prefetch based on which files have already been accessed. The algorithm makes prefetch decisions based on a dynamic data structure whose current state reflects common sequences of file accesses and the frequency of the sequences’ occurrence. When a file is accessed, the structure is consulted to determine the most-probable next file(s) accessed. Depending on a probability threshold, one or more files will be prefetched and placed at the top of the LRU list. The authors experiment with prefetch threshold, and find that prefetching on a surprisingly low probability of reference results in the best performance, measured in terms of hit ratio. The authors also find that limiting the dynamic data structure to “remember” sequences of three or fewer file references resulted in substantial performance benefits at a reasonable cost in terms of the memory required to hold the data structure.

Kimbrel, Cao, Felten, Karlin and Li briefly describe prefetch for a single cache and multiple disks in [Kimb96]. First studying off-line algorithms, those which make use of knowledge of all future requests, the authors describe an algorithm that exploits idle disks by prefetching the first missing block residing on that disk, provided that it will be referenced before its intended victim, the block in cache which is referenced farther in the future. The authors call this algorithm *regular aggressive*. The authors also present a more aggressive off-line algorithm which performs the above prefetching on the reverse of the reference stream, then derives a schedule of prefetches, and applies them to the original stream. This algorithm is referred to as *reverse aggressive*. The authors then use trace-driven simulation to evaluate the algorithms’ performance and find that disk block placement has a profound effect on the performance of the algorithms. Reverse aggressive performs nearly optimally, even under unbalanced loads. Regular aggressive performs well only under balanced loads, such as those resulting from the striping of file data on disks. Limiting lookahead decreased the performance of both algorithms, and seemed to have eliminated the reverse aggressive algorithm’s performance edge over the regular aggressive algorithm’s.

The next seven papers look at I/O reference behavior and disk cache in the context of file systems or database buffer management systems.

Willick, Eager and Bunt use trace-driven simulations to compare the locality-based LRU replacement policy with the frequency-based LFU replacement policy in [Will93]. The authors find that for a stand-alone workstation with disk, the locality-based LRU replacement policy performed well. However, for a network fileserver, which sees only those requests not satisfied by a client's cache, the frequency-based LFU replacement policy outperforms LRU, since most, if not all, of the temporal locality in a reference stream has been filtered out by the client cache. LRU stack depth histograms show that at the client, most hits were at the recently-used end of the stack. However, at the server, hits were more or less evenly distributed across all of the stack depths. Applying a similar analysis to the LFU policy, LFU stack depth histograms shows that at the server, cache hits were concentrated at the most frequently used end of the stack.

A single-user Unix system was the source of a disk I/O reference stream analyzed in [Lee88]. Unlike most Unix disk I/O traces, the data contain references to indirect disk blocks, those which implement the hierarchical structure of the Unix file system. Lee and Wang study the burstiness and locality of requests in terms of the distributions of the interarrival block distances and the LRU stack distances. The *metadata* disk blocks are found to exhibit a very great degree of locality, and respond quite well to a very small number of cache buffers, while the data blocks themselves require a much greater cache capacity to achieve the same degree of performance improvement.

In [Rich93], Richardson and Flynn suggest algorithms to improve cache hit rates by using type information to allocate a fixed portion of the cache to filesystem metadata and caching each 128-bit index in a separate, smaller cache block, resulting in higher hit ratio than the baseline cache. It is not, however, reported whether the increase in hit ratio resulted in increased performance by minimizing "expensive misses"—misses to inodes or program data.

Ousterhout, et. al., trace the *open/create*, *close*, *seek* (reposition within a file,) *unlink* (delete,) *truncate* and *exec* (program load) user-level system calls on timeshared Unix systems, investigating typical file access patterns to determine effective disk cache organization and management in [Oust85]. The authors found that a majority of files were accessed in their entirety, that files tended to be opened for short periods of time, and that 20-30% of all newly-written data were deleted within 30 seconds, and 50% within 5 minutes. A 4-MB cache of disk blocks eliminated between 65 and 90% of all disk accesses. However, the authors' exclusion of read and write records result in a pessimistic estimate of miss ratio by eliminating immediate re-references. Based on the short lifetime of newly-written data, the authors recommend the use of a "flush back" policy in which the disk cache is flushed of dirty data at a fixed interval. Data which are written and then destroyed within an interval are never flushed back, so writes to disk are decreased. However, data lost during a system failure are guaranteed not to be older than the flush interval, so the loss may be limited. We simulate such a policy (which we call "delayed write") in Section 6.5.

In [Thom87], Thompson studies networked file system client caching, finding that the choice of a proper consistency protocol can cut the miss and transfer ratio in half. Thompson's observation that delaying writes significantly reduced write traffic echoes that in [Oust85], with the additional discovery that delaying write-backs of temporary files only (those which can be easily re-created in the event of a system crash) produced nearly the same savings. A more aggressive assumption of sequential file access which implements prefetch as a default outperformed standard Unix prefetch which is activated only upon detection of sequentiality.

Muntz and Honeyman use trace-driven simulation to study an intermediate cache-only server in a network of workstations in [Munt92]. They find that the server was not effective in terms of miss ratio (20-MB caches achieved hit ratios of less than 19%,) but that it did reduce the peak request rate to upstream file servers.

Database buffer management is discussed at length by Effelsberg and Haerder in [Effe84], including qualitative statements about the reference behavior of database systems concerning disk block sharing, locality and predictability. Specifically, the authors state that sharing should be frequent (due to the database being a shared resource,) locality is due not only to the behavior of single transactions (e.g., sequentiality,) but also to the concurrent execution of multiple transactions, and that the reference behavior of transactions can to some extent be predicted with knowledge of access path structures such as indices. Using reference strings from CODASYL database systems, they find that the best performance is realized when database cache buffers are allocated to an individual transaction in a global manner—considering the reference behavior of concurrent transactions as well as that of the transaction itself. LRU and CLOCK replacement algorithms were found satisfactory, although the more general GCLOCK and a scheme based on reference density, the frequency of reference to a buffer page over a particular interval, also performed well.

The following thirteen papers propose modifications to existing disk cache schemes.

Cigas uses traces of Unix file system behavior to drive simulations of a disk cache which buffers "pseudo-files," groups of logically-related disk blocks, using them to guide replacement and prefetching decisions in [Ciga88]. Pseudo-files are formed when arithmetic progressions in disk reference strings are detected. Using a space-time working set approach with pseudo-files tended to outperform the space-time working set alone in terms of miss ratio at equal average cache capacities, but was dependent on workload, and was not feasible for small cache sizes (on the order of less than 1 MB.)

In [Kare94], Karedla, Love and Wherry describe a modification to the LRU replacement strategy which divides the stack into two sections: a protected segment, where disk blocks are placed after a reference to them hits in the cache and from which no replacement occurs, and a probationary segment, from which a victim is chosen for replacement. Disk blocks age out of the protected segment when they are not accessed for a number of references equal to the size of the protected segment in blocks. Thus, a simple frequency-based algorithm is implemented. The authors find that for small cache sizes, the segmented LRU algorithm results in miss ratios as small as those of a LRU-managed cache of twice the size, and that allocating 60-80% of the total cache capacity to the protected segment maximizes the hit ratio.

Kamel and King propose a method to improve the utilization of database caches in [Kame92]. Instead of caching disk blocks containing database pages, the authors advocate pre-storing in cache memory partial predicate results of queries, called "page answers", the set of which is chosen heuristically, and "page traces," which are relevant intermediate page-answers. Correctly choosing the set of page answers and page traces allows queries to be resolved without accessing disk. The authors contend that caching derived data instead of the data itself represents a more efficient use of main memory.

[Ordi93] describes another application of metadata caching in a very large, distributed environment, a name service. The *distributed active catalog* contains information used to constrain the search space for a name service query. Portions of the active catalog are cached at the query site in order to exploit locality in the queries, eliminating the overhead of repeatedly accessing the active catalog for query constraint information. The authors find reductions in query response time and improved performance in multi-user workloads.

Biswas, Ramakrishnan and Towsley study write policies in the context of using non-volatile storage for caching write data in [Bisw93]. They find that even a simple write-behind policy for such a cache is effective at reducing the number of disk writes by more than half. Implementing hysteresis, two thresholds expressed in terms of the percentage of dirty blocks in the write cache, the higher of which initiates write-backs to disk and the lower of which triggers the end to write-backs, reduces disk writes even further. Writes were reduced even further, to about 15% of the original number, by "piggybacking" them with read operations. The authors found little difference in implementing a single non-volatile cache for both reads and writes versus two separate caches (although the read cache may be implemented with volatile storage.)

In [Mill95], Miller and Harper analyze write policies of caches in disk controllers in the presence of the Unix file buffer cache. File buffer caches in main memory tend to decrease the read/write ratio as they absorb a large fraction of read requests. The cache in the disk controller sees only the read misses of the file buffer cache. The Unix system on which Miller and Harper collected traces acts as an NFS file server for twelve diskless clients. Because NFS implements a write-through write policy at the server's file buffer cache and requires clients to write modified data to the server within 30 seconds, the read/write ratio at the controller cache is relatively low. Simulating the entire I/O file system, the authors analyze both write-through and write-back write policies, each with either a write-allocate policy or no-write-allocate policy. The authors note a problem with write-back policies: there is essentially no locality between the disk block being loaded and the block being replaced. This leads to a phenomenon known as "disk thrashing" in which the increased number of seeks degrade average response time to one worse than that of a system with no disk cache at all. Write-through policies uniformly outperformed a non-caching system in the authors' simulations. The authors propose a modification to the write-back policy called "scheduled write-back" (SWB,) in which the disk scheduler maintains two queues: one a queue of requests from the cache, and one a queue of write-back requests. Instead of delaying write-back until a cache block is replaced, SWB uses disk idle times to schedule write-backs. By not replacing blocks until they have already been written back (a replacement policy the authors refer to as "LRU over clean lines,") the authors further improve the performance of SWB, especially under heavy loads.

Traces of the VM, Unix and MVS operating systems drive simulations of a disk cache implementing a replacement algorithm called *frequency-based replacement* in [Robi90]. Robinson and Devarakonda report performance improvement relative to LRU replacement for the Unix and MVS traces, but not the VM traces, measured in terms of hit ratio.

A file cache which operates as an NFS server, is described by Minnich in [Minn93]. The "AutoCacher" runs as a user-level process and provides partial emulation of NFS (READ operations) while caching read-only files on a local disk. Developing the AutoCacher allowed little-used files to be removed from local disks and cached locally when necessary.

In [Thie92], Theibaut, Stone and Wolf present a variable algorithm for managing a cache shared by multiple reference streams (which the author call "processes.") In their experiments, reference streams correspond to accesses to different disks, and the algorithm partitions a disk cache between the reference streams. Building on the work of Stone, Wolf and Turek in [Ston92], which shows that the optimal partition between two streams is that which results in equal miss ratio derivatives for both processes, the authors use a shadow directory to hold a histogram of cache hits in order to drive the partitioning algorithm. They find a relative improvement in the overall and read-hit ratios of 1-2% over conventional LRU, suggesting that conventional LRU results in a near-optimal partitioning of the cache between competing reference streams.

Mann, Birrell, Hisgen, Jerian and Swart implement a write-back write policy on both file and directory data in the context of a distributed file system called Echo in [Mann93]. Echo specifies ordering constraints on the write-behind, allowing consistency to be maintained across server or client crashes or network partitions. The authors find that the speedup an application sees with directory write-back depends on the amount of work it performs between directory writes. Directory write-back made some operations faster in Echo than in Unix.

Bolot and Afifi study caching in the context of the X.500 distributed directory service in [Bolo93]. The X.500 directory service provides mappings between a set of names and a set of associated attributes. As it is intended to service very large and diverse namespaces, the X.500 directory can be though of as a distributed database; the information for a given name may reside on one of several, possibly remote, servers. The authors use traces to drive simulations addressing the effects of cache size, replacement policy and partitioning on miss ratio and average access time. Determining the interreference interval distribution for names in the traces, the authors find that requests for names exhibit a great degree of locality. Therefore, not surprisingly, a modest amount of cache resulted in a significant reduction on average access time. With respect to replacement policy, LRU outperformed both a frequency-based policy and a random policy at small cache sizes; at larger cache sizes, LRU did not perform significantly differently, however. Noting the nonuniform cost of servicing misses for the different classes of request (geographically locally-resolved requests, requests including names on the Internet, requests to names on the French X.25 public networks and other requests including requests to names on other European X.25 networks,) the authors propose dividing the cache into four partitions, each for a given class of request. Each partition was guaranteed a portion of the cache, and competed with one of the other partitions for unused space in the cache; each class' guaranteed partition was based on its access cost relative to its "competition." The authors find that even though the partitioned cache exhibited a higher miss ratio than a LRU cache, the overall performance was better, since it tended to minimize misses to the "expensive" classes.

In [Pura96], Purakayastha, Ellis and Kotz propose ENWRICH, a compute-processor write-caching technique that collects a large number of writes and flushes them to disk intermittently using a high-performance disk-directed I/O technique. By limiting the caching to write-only files, cache consistency is easily maintained. Batching writes allows them not only to be performed off-line with respect to computation, but also allows for their optimization, and can result in the elimination of writes. Trace-driven simulations of ENWRICH showed throughput two to three times that of traditional cache, which forwards I/O requests to an I/O processor containing two buffers per compute processor.

The last seven studies extend the concept of trace-driven simulation.

McNutt presents a novel model of cache behavior based on the "hierarchical reuse probability" concept used in the study of fractals in [McNu93]. The concept is used to predict cache miss ratios with larger caches given miss ratios for a cache of a particular size, and to extend trace-based measurements to cache sizes for which the trace is not large enough to fill the cache. The model correctly predicts miss ratios with respect to an average cache residency time curve. In [Frie95b], Friedman refutes McNutt's claims, noting that using McNutt's equation for estimating cache requirements predicts needing less cache to achieve higher hit ratios for the same workload. Friedman states that McNutt's equation provides a lower bound for cache requirements, and fails to take into account the cache necessary to hold the working set.

Thekkath, Wilkes and Lazowska discuss techniques for performing simulation-based evaluations of file and disk systems in [Thek92]. They propose a technique called "bootstrapping", a technique borrowed from statistics, for generating new traces from old ones. They also present a more realistic disk model with submodels of the controller, the disk mechanism and track buffers. "Scaffolding" between the traces, a snapshot of the file system, and disk and file system simulators use lightweight threads to simulate the independent progress of each process making file system calls.

Levy and Morris use derived trace data (in the form of stack depth distributions) of individual address streams and use a simple Bernoulli switching assumption to predict an overall hit ratio in [Levy95]. An I-way Bernoulli process chooses from which of the I independent streams the next access will come. The authors post-processed traces of the logical I/O requests made by a database management system to generate two stack depth distributions of requests for data (i.e., table) pages and requests for index pages. The hit ratio resulting from the combination of the two streams was predicted using Bernoulli superposition and was found to closely approximate the actual hit ratio of the database buffer. The authors contend that Bernoulli mixing appears to be a good approximation, provided that the buffer is not so small that its size is comparable with the lengths of bursts of one type of data.

In [Phal95], Phalke and Gopinath present a program modeling technique which quantifies temporal locality as a Markov chain on a per-address basis. Thus, for each address in a trace, inter-reference gap information is kept and later used to make cache replacement decisions using Markov chain predictors to identify the address used farthest in the future. The authors see up to a 35% improvement in miss ratio over LRU, but with significant space-time costs for making the replacement decision.

Wu extends stack-based trace-driven simulation of cache memories to perform one-pass simulations of LRU cache memories with write-back write policies that evaluates write ratios of multiple block sizes in [Wu94].

In [Varm95], Varma and Jacobson combine I/O traces to create a workload to drive simulations of a Redundant Array of Inexpensive Disks (RAID) with a non-volatile write cache. The authors study the effects of the algorithm used to make decisions to destage write data from the write cache. The authors also propose an algorithm which dynamically varies the rate of destage based on the occupancy of the write cache, which results in minimal effect on host reads, while providing reasonable insensitivity to bursts of write traffic which may overwhelm other algorithms.

A3.2 Analytical Studies

The following five papers analyze the effects of hit ratio and other cache performance parameters on I/O subsystem performance.

An IBM 3880 cached storage controller is modeled by Buzen in [Buze83] to determine the effect of hit ratio, read/write ratio, access rate and channel utilization on the performance of the I/O subsystem.

Cached disks in an IBM I/O system with Dynamic Path Reconnection are modeled in [Bran88] by Brandwayn in order to determine the effect of miss ratio on queuing delays and server time variance. For some workloads the assumed hit ratios of 79-95% may be unrealistic. For example, for one of our traces, the theoretical minimum demand-fetch hit ratio only reached 75% at a cache size of 400 MB.

Marazas, Blum and MacNair model an IBM 4967 disk control unit with cache to determine its performance in a transaction processing system in [Mara83], and find that a 75% hit ratio results in a 40% decrease in response time.

Carson and Setia use queuing theory to study the "periodic update" policy which writes dirty cache blocks to disk on a periodic basis (we call such a policy "delayed write") in [Cars92]. They find that the periodic update policy fails to outperform a write-through policy with respect to average response time for read requests in all but a narrow range of operating conditions (large read/write ratios, high write hit ratio.) The authors propose more robust policies: "periodic update with read priority," which pre-empts bulk writes for read requests, and "individual periodic update" policy which delays write operations on an individual basis, rather than presenting them to the disk system in bulk. Both policies are shown to avoid the bottleneck that periodic update may cause.

Approaches to analytically modeling cached disk systems with uneven loading are developed and verified in [Mank83]. The system is divided into multiple subsystems depending on the skewness of the load analyzed separately, allowing the response time of the entire disk system to be calculated.

The remaining papers discuss disk cache modeling in general and other aspects of modeling disk caches.

Lazowska, Zahorjan, Cheriton and Zwaenepoel study the performance of single-user diskless workstations that access files remotely over a local area network in [Lazo86]. The authors use a parameterized queuing network of a number of workstations, a file server, and the LAN which connects them. Users of workstations were monitored in order to characterize the models of the customers of the file server, the diskless workstations. The authors then parameterize the queuing network and study several design alternatives for improving the performance of remote-file access, including buffering smaller blocks at the server (to reduce latency at the client,) increasing disk block size (to amortize seek and rotational latency costs over a larger block size,) optimizing the file server code and reducing server CPU costs for remote file access, increasing the packet size on the local area network, prefetching disk blocks to the CPU, adding another file server, and implementing client file block caching. The authors find that of all the design alternatives, implementing file block caching at the client performs the best, as hits decrease contention for both the network and the server.

Casas and Sevcik present a performance model of database buffer management in [Casa89]; a hypothesis of the model is that database page references are Bradford-Zipf distributed. The authors collect database block reference strings and frequencies from five System 2000 databases, and find for each, behavior which closely approximates a B-Z type of distribution. The authors then derive expressions which estimate the miss ratio achieved by a buffer manager implementing a Random replacement policy and compare them to measured miss ratios for LRU and Random replacement policies. The B-Z estimation delivered an 8.7% average error over all measured cases, leading the authors to recommend using the B-Z distribution to drive studies of systems in the design stage, before any measurements are possible. The authors also propose a strategy, which they call FIX, which fixes a number of the most frequently-used blocks in cache and uses the remaining buffer to read in other blocks, and find a miss ratio that outperforms LRU and Random at small cache sizes (LRU outperforms it at larger cache sizes.)

Dan, Dias and Yu refine the modeling of skewness of reference (degree of locality) and examine its effect on the database buffer hit ratio and the probability of data contention in a multi-system data sharing environment in [Dan90a]. Studying the case in which the database consists two partitions, one which has a high access frequency (the *hot set*) and one which has a low access frequency (the *cold set*,) the authors model the buffer using the LRU policy and the Optimistic Coherency Control (OCC) protocol (which invalidates buffer blocks for conflicting transactions on other systems when a transaction commits its data.) The authors find that the hot set is more susceptible to buffer invalidation due to its higher probability of existing in multiple buffers at a given time. For re-run transactions (those which were previously aborted by the OCC and whose buffer blocks had been invalidated,) the cold set hit ratio was generally 100% (cold set buffers are not invalidated by the OCC protocol,) while the hot set hit ratio generally decreased with increasing numbers of nodes in the system. The authors also examine transaction response time as a function of buffer size, and find three regions: one in which data referenced during the first run of a transaction are flushed before the transaction is re-run (for buffer sizes smaller than the working set size,) a region in which the invalidation rate for the hot set is less than the re-reference rate (the response time decreases sharply in this region with increasing buffer size) and a third region in which additional buffer capacity cannot be used by the hot set, and the response time decreases only slowly. Varying the degree of skew results in two competing effects: first the level of data contention increases, resulting in increased probability of hot set buffer block invalidation and a decrease in hot set hit rate; second, the overall buffer hit rate increases due to a larger fraction of the buffer being occupied by the hot set. For very small and very large buffer sizes, the effect of increased contention predominates, while for intermediate buffer sizes, the increased hit rate due to a larger proportion of hot blocks predominates.

In [Falo91], Faloutsos, Ng and Sellis employ a simple queueing model and use *predictors*, estimates of expected throughput and effective disk utilization, to make buffer allocation decisions based on the estimates of whether a given buffer allocation will improve the overall performance of the system. Simulated results show the proposed method consistently outperforms DBMIN, an algorithm which makes allocations based on the size of the locality set of a job.

In [Palm91], Palmer and Zdonik discuss FIDO, a cache which adaptively prefetches data by employing an associative memory to recognize access patterns. A training phase adapts the associative memory contents to data and access patterns. A 1000-element cache plus a 500-element FIDO associative memory significantly outperformed a 2000-unit LRU cache. Learning and pattern recognition takes place on a per-file basis.

Soloviev considers disk- and main-memory prefetching in [Solo96]. The system under consideration employs disk striping, in which a given file's blocks are distributed over multiple disks. As with most modern disks, each includes a segmented local cache. The workload consists of a number of concurrent sequential scans of various files. The author finds that small variations in the disks' loads causes the more heavily-loaded disk to become a bottleneck, degrading system performance. Studying various ways of eliminating the bottleneck, the author finds that a small amount of main memory cache outperforms other approaches such as limiting the number of disks on which a given file's blocks could be stored, limiting the number of concurrent file scans, or dynamically varying the number of segments in the disk caches. Implementing request queues at each disk combined with disk cache-aware prefetching was the only disk-level prefetching method that provided performance close to that of main memory prefetching.

Solworth and Orji examine “Write-Only Disk Caches” in [Solw90], proposing a cache whose purpose is to hold blocks to be written to disk in order to minimize the cost of writes. The writes are “piggy-backed” onto reads, being written during the reads’ rotational delays. Two piggy-back schemes are studied, “zero-cost” piggy-backing in which only those blocks rotationally prior to the read request are written, and “track-based” piggy-backing in which also those blocks rotationally after the requested block are written. [Solw90] models a single-surface disk and the authors find that when the write-only cache is small, the track-based scheme outperforms the zero-cost scheme. The authors extend their analysis to multiple-surface disks in [Orji92], finding the multiple-surface model to be more effective for large cache sizes.

In [Bhid93], Bhide, Dan and Dias present a simple approximation of the LRU cache warm-up transient (the hit ratio that a cache displays after a cold-start or a discontinuity in reference behavior and before a steady state has been reached.) Assuming the Independent Reference Model for cache accesses, the authors validate approximations against the warm-up transient behavior of several database transaction workloads. The authors find that the expected steady-state hit probability is identical to the hit probability when the buffer becomes full. The authors’ method is generalized to handle repeated changes before a steady-state is reached.

Dan, Dias and Yu develop an analytic model for coupled systems sharing a common database in [Dan90b]. A database buffer model is integrated with a concurrency control (CC) model employing Optimistic Concurrency Control (OCC) and the interactions between the buffer hit ratio and CC invalidation are evaluated to estimate the resulting response time. In the OCC protocol, rerun transactions are more likely to find previously accessed data in the buffer, and, thus, tend to have lower miss ratios than first-run transactions. With increasing numbers of concurrent transactions, the resulting increase in invalidation from other nodes can reduce the effective buffer size; indeed, for a given number of nodes, beyond the size of buffer at which the invalidation rate equals the buffer replacement rate, addition buffer does not reduce response time.

Ng and Yang analyze the special buffering requirements of multimedia systems in [Ng96]. Multimedia systems are special cases of real-time systems for which there are time constraints for computations and I/O. Audio and video data are delay-sensitive, and maintaining continuity in playback necessitate the consumption of large amounts of system bandwidth and memory, part of which is allocated to buffers. Ng and Yang advocate sharing buffers between multiple multimedia streams, with dynamic allocation of buffer space to the different streams. According to the authors, sharing buffers between streams can decrease total buffer requirements by 50%. Furthermore, Ng and Yang propose three prefetching algorithms which are intended to improve disk utilization. The authors’ analyses show that “intelligent prefetching” based on a “marginal gain” analysis outperforms blindly prefetching as much data as possible for the stream at the head of the queue in terms of throughput, disk utilization and buffer utilization.

A3.3 Implementations and Commercial Products

The following papers describe commercial disk cache products and users’ experiences with them.

The first disk cache product was the Memorex 3770, described in [Memo78], which consisted of a 1-to-18-MB CCD cache controlled by a microprocessor, implementing a least-recently-used replacement algorithm, increasing transfer rate for the tracks contained in the cache from 806 KB per second to 1.2 MB per second.

The IBM 3880 Model 13 storage controller ([IBM83a], [IBM83b]) was IBM’s first caching storage controller for disk data. The 3880-13 contained either 4 or 8 MB of disk cache, and offered a version with two storage directors. The cache was managed using the LRU policy, and writes were propagated out to disk before device end was signaled. A major performance limiter in the 3880-13 was its implementation of *serial transfers* only; after a record was transferred to the processor, the DASD required repositioning in order to read the rest of the track into cache, increasing busy time for the disk and controller.

The IBM 3880 Model 23 storage controller ([IBM85a], [IBM85b]) overcame this limitation by implementing *branching transfers*, allowing data to simultaneously be transferred from DASD to cache and from cache to channel. Cache size was increased to a maximum of 64 MB, and the 3880-23 added two new caching modes: *sequential mode* instructed the controller to automatically stage the next track to cache, (which we refer to as one-block lookahead,) while *bypass-cache* mode takes the cache out of the path—data are transferred directly from DASD to processor.

The IBM 3880 and 3990 storage controllers are discussed in [Gros85], [Gros89], and [Meno88].

Similar disk cache products from other manufacturers include the Sperry Univac Cache/Disk System [Sper81], the STC 8890 Disk/Cache Buffer Control Unit [STC82], and the Amperif Cache Disk System [Fuld88].

Users of IBM's caching storage controllers (and its compatibles,) however, have noted that the presence of the cache does not universally improve I/O performance system-wide. Channel utilization may increase due to cache overhead and the ability of caching controllers to absorb higher rates of I/O operations for a given performance than non-caching controllers. This may exacerbate a known problem with the IBM disk architecture called "RPS miss". Devices employing RPS (Rotational Position Sensing) disconnect from the channel between the time the seek completes and the time the desired data rotate under the head ("rotational latency.") If all of the channels or directors are busy, a reconnect miss occurs and the data transfer is delayed by at least one revolution (typically 16.7 milliseconds.) Because channel utilization may increase (due to cache overhead and the ability of caching controllers to absorb higher rates of I/O operations for a given performance than non-caching controllers,) devices which are not cached, but share channels and directors with cached devices (as well as cache read miss disk accesses,) may experience increased incidence of RPS misses in the presence of a cache, possibly leading to degraded system-wide I/O performance. Documentation of this phenomenon includes [Mill82] and [Buze83] (as mentioned in [Frie83],) and [Kova86]. As an aside, Hoffecker, Walsh and Cunard propose the use of actuator-level buffers to eliminate the rotational latency component of RPS misses in [Hoff89].

In [Arti95], Artis discusses a recent feature of the IBM 3990-6 called *record level cache one* (RLC 1) aimed at improving the performance of online transaction processing (OLTP) workloads. OLTP workloads are characterized by a read of an essentially random record followed by a write of that record. The traditional mode of operation in the 3990 stages the record requested and all records following it in the track. Since records in OLTP workloads tend to be a relatively small fraction of the track size, most of the disk data staged into the cache is never used, and results in excess utilization of the control unit's "back end," the paths of the disks. At high I/O rates, the excess utilization may lead to severe degradation of performance to below that of "bypass cache" mode. To address this issue, in 1994, IBM introduced the RLC 1 facility, which stages only the record requested. Using a program which randomly reads and then writes a single record, the author shows that the RLC 1 facility does indeed improve performance over that of either normal operation or bypass-cache operation.

Friedman discusses the use of a general-purpose Unix-based multicomputer to emulate an IBM 3990 cached controller in [Frie95a]. The author argues that the use of general-purpose (i.e., programmable) components into a hardware solution such as a 3990 disk controller results in lower cost and quicker development time. The design approach involves modifying Unix' disk cache software, which supports SCSI disks employing a fixed-block architecture, to emulate the 3990, which uses a disk-geometry -based architecture. Designers of such systems face such issues as how to map between the fixed-size blocks and variable-sized blocks. Differences between the rates of revolution of the emulated disks and the actual disks require special handling of synchronous channel programs.

Workstation-class computer systems typically function in a distributed computing environment consisting of file servers and clients. An early caching file system for a workstation is described in [Schr85]. The distributed file system, called CFS, provides workstations with a hierarchical naming environment which includes all files local to the workstation itself and on all file servers (but not other clients.) The complexity of the consistency problem is drastically reduced by the fact that all remote (that is, located on a server) files in CFS are read-only. The most difficult aspect of the client cache involves the management of the cache. A program which allocates sectors on the local disk triggers cache flushing when fewer than 1000 pages are free on the disk, so most cache flushes occur asynchronously. The CFS cache makes it possible to operate a workstation with approximately 20 MB of local disk storage, and lowers the load on the file servers significantly.

A survey of caching techniques for distributed file systems is presented in [KSmit90].

Based on the observations made in [Oust85] that written data tend to be short-lived, Nelson, Welch and Ousterhout describe the Sprite network operating system's file cache in [Nels88], implementing a delayed-write policy for both clients and servers. Sprite also allows diskless clients and servers of the file system to use their main memories to cache data, negotiating with the virtual memory subsystem for its allocation of memory. Servers guarantee client data consistency by issuing writeback commands to clients caching written data requested another client.

In [Welc91], Welch reports on the measured performance of Sprite's file cache, finding that, indeed, the delayed write policy eliminates 40% to 60% of all client writeback traffic, and that write traffic ratios as low as 30% had been observed during large compilations (which create significant amounts of temporary files.) The hourly average read miss ratios ranged from 20% to 60%, and averaged 35% to 39%. Consistency-related actions such as calls for dirty data writeback due to the detection of serial sharing of written data and invalidates due to concurrent write sharing occurred on less than 2% of all file open operations. The adaptive nature of the dynamic allocation of main memory to file cache and virtual memory was observed, with clients tending to devote a smaller fraction of their memory to the file caches than file servers.

In [Bake93], Baker and Ousterhout attempt to reduce Sprite file server recovery time to less than 90 seconds by taking advantage of the distributed state present in the clients. When a client detects a reboot of a server, it transfers its pertinent file system state to the cache. An early bottleneck with respect to the number of simultaneous RPC (Remote Procedure Call) requests a recovering server can handle was overcome by the addition of negative acknowledgment to the RPC system so an overloaded service can inform a client of its inability to service the client's transferral of state. A client receiving the negative acknowledgment can then back off and retry at a later date. The negative RPC acknowledgment reduced 40-client state recovery time from around 15 minutes to approximately 100 seconds.

Dixon, Marazas, McNeill and Merckel receive a patent for a disk cache with variable prefetch in [Dixo84]. The number of disk blocks prefetched (one or two) depends on the location of the requested record within the 8-record disk block; requests to blocks located near the end of the record, beyond a threshold location result in two blocks being prefetched. Requests for records beyond the location corresponding to another threshold, but preceding the location corresponding to the first threshold result in a single block being prefetched, and requests to records preceding the location corresponding to the lower threshold result in no prefetch. The two thresholds are dynamically varied in response to the measured read hit ratio.

Prefetching in file systems for MIMD multiprocessors is studied by Kotz and Ellis in [Kotz90]. The system under consideration consists of a network of processors, each with its own channel and disk. File blocks are assigned to processors in a round-robin fashion. This arrangement differs from a distributed file system in that a file's blocks will be spread over multiple disks, and it differs from a disk striping arrangement in that disks are connected to separate processors through separate channels. The authors use a testbed consisting of a Butterfly Plus multiprocessor with the interleaved file system modeled in each processor's memory by a disk image and artificial delays and artificial workloads representing combinations of observed file access behavior. The authors find that the hit ratio metric is an overly optimistic indicator of the resulting performance, since prefetched blocks might not arrive at a processor by the time they are needed. Still prefetch is found to generally contribute to a decrease in overall execution time of a parallel program, provided that the benefits of prefetch are well-distributed between the processors. Otherwise, prefetching can increase total execution time even as it reduces average block read time.

[Yim95] contains some hints for optimizing Fortran I/O in a Cray supercomputer environment. In addition to making general suggestions like using unformatted (non-ASCII) files wherever possible to avoid conversion, and resizing I/O buffers from their defaults to more closely match the amount of data requested at a given time, Yim also recommends specifying that files use caching buffers. The cache size, cache block size and the number of blocks to read-ahead upon the detection of sequentiality are all configurable. Additionally, larger caches (up to 100 MB) can be defined on the Solid-State Storage Device. Users can also specify I/O operations to take place asynchronously, perform computations which the I/O takes place, and perform explicit synchronization when the data are needed.

D'Azevedo and Romine describe EDONIO, a library of I/O routines for Intel iPSC/860 and Paragon supercomputers which allows users to create a disk cache on a per-application basis in [D'Aze95]. The library consists of special versions of Unix-style *read, write* and *seek* system calls, as well as a call to specify the size of the disk cache. The resulting cache will consist of two portions, one of which contains only read-only files. Another system call, *do_preload,* fills any empty blocks in the disk cache with disk data. *do_preload*, however, will not displace any blocks currently in the disk cache, and it performs a one-time prefetch only. The disk cache is managed using LRU. Using EDONIO routines resulted in significant decreases in I/O times relative to native, non-caching I/O routines for code simulating the behavior of a large finite-element application.

Tokunaga, Hirai and Yamamoto treat sequential files differently than other files by prefetching on cache misses and sending a sequential file cache block to the head of the LRU list when the last record in the block has been accessed in [Toku80]. Temporary files use a write-allocate strategy, while permanent files may use a no-write-allocate strategy. Both permanent and temporary files are written through to the cache unless they have been registered in "High Speed File Mode," which notified I/O completion after the record has been written to cache and before it has been written to disk. This improves write performance with a slight decrease in reliability. The authors implemented the disk controller and found improvements of a factor of two to ten in average disk access time.

Alonso, Barbara and Garcia-Molina discuss issues in data caching in an information retrieval system consisting of a single central node storing data and a large number of remote terminals which simply perform queries in [Alon90]. They present a form of caching at the terminals in which the data are allowed to deviate from the value currently stored at the central node in a controlled manner, allowing updates to the cached data to occur when communications costs are low.

Mungal describes the software provided by a cache supplier and its application to capacity planning and performance modeling in [Mung86]. Data from customers relating hit ratio to cache size are presented.

In [Patt93], Patterson, Gibson and Satyanarayanan exploit application-level "hints" about future I/O accesses. *Transparent informed prefetching* reduces measured file access read latency in both local and networked file systems by overlapping read latency with compute cycles. The authors measure reductions in execution time of up to 30%, and argue that greater performance benefits can be realized when such prefetching is integrated with low-level resource management. In [Patt95], the concept is refined to estimate the impact of prefetching a hinted block in terms of the cost of displacing an LRU block (one which has already been referenced,) a hinted block, or other prefetched blocks. The authors find that the more sophisticated prefetch decisions arising from the cost-benefit analysis results in decreased I/O access time even for an application looping on a dataset larger than the total cache capacity. Without the estimators which determine the estimated cost of a prefetch decision, each of the application's disk blocks had been flushed by the time it was re-referenced.

In [Berg93], Berger discusses a feature called "Dynamic Cache Management Enhancement," which measures hit ratios on a per-data set basis to drive caching decisions based on a hit ratio threshold derived from a measured overall cache hit ratio. Although the total number of cache hits did not change significantly, a prototype improved the overall I/O response time by about 25%, resulting in a 25% reduction in IMS transaction response time.

In [Moha95], Mohan implements a method for guaranteeing the detection of partial disk writes in a database system using write-ahead logging and multi-sector pages. Since sectors may be written out of order by SCSI disks, although SCSI disks can detect a partial write of a sector, they cannot detect partial page writes if none of the sectors was partial write. Mohan proposes a method, using *check_bits* , bits at the end of each sector. If the check_bits in each sector of a page are not all equal, a partial disk write has been detected, and the appropriate recovery action can then be taken. Unlike previously proposed solutions, Mohan's implementation works with common read/write optimizations such as avoiding the need for on-disk initialization of new pages during file extension operations, avoiding disk read of pages during reallocation, allows buffer pool purging of deallocated pages of a file and file name reuse, while accommodating ARIES-style recovery with minimal CPU and disk space overhead.

In [Arun96], Arunachalam, Choudhary and Rullman describe a prototype prefetching parallel file system in an Intel Paragon massively parallel supercomputer. Prefetch requests are implemented as asynchronous read requests to the Paragon File System (PFS,) making use of existing OS support for asynchronous I/O requests. The prefetched data are stored in a buffer associated with the file to which the request was made. Implementing prefetch had little effect on the read bandwidth for I/O-bound applications since the prefetched block was requested before the prefetch had a chance to finish. However, a balanced application which performed significant amounts of both I/O and computation was able to increase its read bandwidth significantly, with larger prefetch request sizes being better able to offset the overhead of implementing prefetch.

A survey of disk cache products for personal computers in presented in [Jali89].

A3.4 Improving I/O Performance Independently of Disk Cache

For completeness' sake, we present some well-known and proposed methods of improving disk I/O performance independently of a disk cache. As technologies change, these approaches may fall into or out of favor with respect to disk cache.

In [Smit81], Smith discusses various ways to optimize file systems, including file system disk block size, data set (file) placement, disk arm scheduling, rotational scheduling, compaction, I/O multipathing and file data structures. Smith also discuss disk spindle buffers in [Smit85].

In [McKu84], McKusick presents some improvements to the original Unix file system which include increasing the disk block size, allowing fragments of disk blocks at the ends of files, and improving the layout of file system metadata, as well as that of the data themselves. In [McKu90], McKusick and Karels propose radical changes, including dispensing with Unix's explicit disk cache and combining it with the free memory pool, thus allowing any free memory to be used to cache file blocks. A similar approach is taken in [Brau89].

Patterson, Gibson and Katz propose a high-performance file system in [Patt87], called *RAID* , which makes use of a technique called *disk striping* to store the data blocks of files on multiple disks, allowing parallel access of the data. The decrease in reliability due to the increased number of devices is addressed by adding redundancy in the form of multiple copies of disk data or parity information. This approach is also discussed in [Post88] and [Koel89]. Disk striping for parallel I/O is implemented in a supercomputing environment in [Naga90]. In [Jone93], Jones proposes a disk susbsystem with redundant components. Data and dual-redundancy information are recorded in a rotating fashion throughout a disk array. Data are maintained using a log-structured system.

Maintaining two copies of each disk block, either on different disks or on a single disk, is proposed in [Ng91], allowing the latency time to be reduced by accessing the disk block closest to the actuator. The results depended on the ratio of disk reads to writes, as maintaining multiple copies necessitated an extra write to the redundant copy. If the second write was performed synchronously, only at high read-to-write ratios did the latency and RPS miss reduction exceed the cost of the extra writes. Synchronizing the disks holding the copies of the data such that they were 180 degrees out of phase resulted in further improvements. Putting both copies on a single disk resulted in the best performance. This technique was first described by Scheffler in [Sche73].

The emergence of the distributed computing environment has changed the context of disk scheduling. File servers devote a large amount of their memory to disk buffer, and satisfy requests for large numbers of clients, resulting in longer disk queues than stand-alone systems. Seltzer, Chen and Ousterhout propose two algorithms which outperform traditional disk scheduling techniques by taking rotational latency into account in [Selt90]. Adding an age-based weighting factor to an algorithm which satisfies requests for the shortest seek and latency time first to maintain fairness resulted in performance improvements of three to four times, measured in terms of the average I/O time.

Van de Goor and Moolenaar investigate the transfer of some of the I/O functions of Unix to dedicated I/O processors in [Van88]. As the implementation was not complete, they were not able to determine the resulting performance.

McDonald investigates dynamic restructuring of the data blocks on the disk to reduce the effects of fragmentation in [McDo88]. In this approach, files which are referenced at a rate exceeding a threshold and that span multiple disk cylinders are marked and then restructured such that they are stored on the minimum number of cylinders necessary to hold them. Refining the marking technique by weighting the reference rate by a measure of the degree of a file's fragmentation was shown to improve performance even further. By restructuring a small number of files, performance in terms of file access time was improved significantly.

Mohan proposes schemes for fast page transfer between hosts for a multiple-system database management system in a shared-disk environment [Moha91]. The author eliminates disk I/Os by using a high-speed communications link to transfer dirty pages between hosts, and studies the scheme's implications on log management, recovery, lock management, buffer management and coherency-control protocols.

In [D'Gue92], D'Guerra and Deshmukh discuss the differences between RAM disk and disk cache, and describe considerations to use when selecting between the two solutions for reducing I/O bottlenecks. They find that RAM disk is good for temporary data (those which can be easily re-created in the event of a system crash.) Disk cache is preferred for applications with a high read/write ratio and those which use files which are too large to fit into a RAM disk.

In [Mars94], Marsh, Douglis and Krishnan forward the idea of inserting a flash memory (a modified EEPROM device that allows in-system programming) between the DRAM disk cache and disk in a mobile computer. In a mobile environment, disk accesses increase in relative expense due to aggressive power management techniques which spin down the disk at any opportunity. They find that the "FlashCache" can reduce the power consumption of the storage subsystem by 20-40% and improve overall response time by 30-70%.

Hunter describes buffering at the disk arm itself in [Hunt81].

A4 Detailed Trace Descriptions

DB2 GTF Traces: These traces came to us from Ken Kolence, Chairman of the Computer Measurement Group, as part of a study by Steven Viavant in [Viav89]. They represent the reference behavior of IBM's DB2 relational database in several environments. The enterprises traced were Security Pacific Bank (**Bank**) and Crowley Maritime Corporation (**Transport** .) These traces were collected using the DB2PM DB2 performance monitoring package and the Generalized Tracing Facility of the MVS operating system.

DB2PM couldn't directly trace the disk accesses of the database. Instead it recorded the locking of the database resources (disk blocks and files.) Given the status of these locks, and knowledge of the resource locked, it was possible for researchers to recreate the disk reference strings after making some assumptions (discussed in [Viav89].) The main limitation to these traces is the fact that the timestamps are therefore associated with the time the resource was locked, and not with its actual use. However, we make the assumption that the variance of the time from which a resource is locked until its use is small enough to allow us to ignore it in calculating and comparing durations (e.g., the duration of time a resource is in use.)

Other DB2 traces: We obtained a trace of DB2 database references from researchers at IBM. However, it was generated by instrumenting DB2 itself, and thus, represents the actual DB2 I/O references. This greatly cut down on the translation effort. In addition to some other records, this trace includes a record for every disk I/O. This trace is valuable in that it represents a long interval of system operation. **Telecom** was created from this trace.

IMS traces: We obtained two traces of IMS, a hierarchical database, from Sam DeFazio, who studied them in ([DeFa88].) DeFazio's Ph.D. thesis, [DeFa88], explains the trace generation process in detail, but briefly, by operating IMS with a trace option enabled, the database log file (normally used for recovery, auditing and monitoring) is augmented with reference information. The log file was then interpreted to create a trace that represents database reference activity. The shortcoming of this data is the lack of timestamps, which precludes us from performing some characterizations and some experiments that we were able to perform on other traces.

The first of these two traces is called **Oil** , and was collected at Gulf Oil Corporation's Harmarville Datacenter during 1985. About 100 IMS physical databases are represented, and the workload consists of a number of financial, modeling, inventory, technical and personnel systems. We refer to the other trace as **Monarch** , which represents the activity of an IMS system at the Monarch Marking Company's Dayton Datacenter during 1987.

GCOS traces: We obtained two traces of disk reference activity on timeshared Honeywell-Bull GCOS8 systems. Rod Schultz, a researcher in Honeywell-Bull's Phoenix, Arizona, Computer Performance Evaluation group, developed a monitor called STEM, the System Trace Event Monitor, which is a standard GCOS8 product. We are prevented from describing the enterprises which were traced, except in the most general terms. We refer to them only as **Timeshare** and **Manufacture** . The STEM traces include system and user references.

A5 Trace format

In order to facilitate our investigations, the traces were all translated into a common format. This appendix describes the structure of the records in the format we used. Each record type is of fixed length, but the record types have different lengths.

Begin record: The traces contain nine different record types, the first of which corresponds to the beginning of a transaction or process. The first byte of the *Begin* record indicates the record type, and is equal to the hexadecimal value 0x01. The next four bytes contain a number uniquely identifying the process/transaction within that trace. Following that are four bytes containing the time corresponding to the event, and the next twelve bytes identify the user corresponding to the transaction/process in ASCII characters.

Open record: The *Open* record corresponds to the first reference to a particular database or file by a transaction/ process. The byte indicating its type is equal to the hexadecimal number 0x0f. The next four bytes uniquely identify the database/file being opened. The timestamp is contained in the next four bytes. The following twelve bytes contain ASCII characters identifying the user opening the database/file. The next byte encodes the database/file (resource) type. Examples of types are database log, data, database index root block, database index leaf block, HDAM file, and temporary random access file. The encoding of the resource type field is contained in Table A1. The size of the database/file being opened is contained in the following four bytes and a number identifying the device on which the database or file resides is contained in the last four bytes, when that information is available.

I/O Record: This record corresponds to a reference made to a file or database. The first byte of this record (equal to the hexadecimal number 0x00) indicates its type. The transaction/process identifier is contained in the next four bytes. The following four bytes contain the number of references between this one and the next *I/O* record accessing the same file block. This field is used in simulations of the MIN replacement policy. The next byte contains the number of references to the file block to which this record corresponds. In previous investigations, this number was used to reduce trace length by combining successive references to a disk block into a single record. In the traces used in this study, this number is always one, but the field is included for backward compatibility with the earlier trace format. The next byte encodes the nature of the reference, (0x00 for read or 0x01 for write.) The following byte encodes the type of file being accessed (resource type,) an example being an index or data file. The next four bytes contain a file identifier, and the following four, the block being referenced. All references are expressed in terms of 4096-byte blocks. Immediate rereferences to the same 4096-byte block were not consolidated. The next four bytes contain the time at which the reference was made. The last four bytes also contain a timestamp corresponding to the amount of time between the time the I/O was requested and the time it was available. This field is empty for the traces in this study, since that information was unavailable for the traces we collected. The field is included for forward compatibility.

Close Record: The *Close* record corresponds to the last reference to a database/file in a transaction/process. Like all other records, it begins with a byte indicating its type which is equal to the hexadecimal number 0x10. The following four bytes identify the database/file being closed. The timestamp is contained in the next four bytes. The elapsed time since the database/file was opened is contained in the next four bytes. The transaction/process closing the database/file is identified in the next four bytes. The following four bytes contain the number of bytes transferred to and/or from the file by this transaction/process during the time it was open, and the number of references made by the transaction/process during this duration is contained in the next four bytes. The last byte indicates the mode in which the file was accessed—0x01 for read only, 0x02 for write only, 0x03 for read/write and 0x00 for unknown.

End record: The *End* record corresponds to the end of a transaction/process. The byte corresponding to its type contains the number 0x02. The following four bytes identify the transaction/process. The timestamp is contained in the next four bytes, while the elapsed time since the transaction/process began is contained in the last four bytes.

Max_Data record: For some of the traces, there is a *Max_Data* record for each of the databases/files referenced in the trace. The record contains a byte indicating its type (0x0d,) four bytes identifying the database/file, and four bytes containing the largest block offset referenced in the file over the duration of the trace. These records were used primarily in the translation process.

Max_Index record: The *Max_Index* record has the same format as the *Max_Data* record, but is identified by its type being equal to 0x0c. It corresponds to the largest block offset referenced in the file for an index.

End_of_Tape record: The *Eotape* record indicates the end of a tape on which the raw trace was stored. It consists of a single byte indicating its type, equal to the hexadecimal number 0x0a.

End_of_Trace record: The *Eotrace* record indicates the end of a trace and is the last record in a trace. It contains a byte indicating its type (0x0e) and a four-byte timestamp.

Timestamp format: Unlike the format of the other binary fields (for transaction/process identifier and database/file identifier, for example,) the four bytes for the timestamps are written least-significant byte first. They are written in this manner to maintain compatibility with earlier trace formats.

The units of this field are 128 usec. That is, if the field contains the number 1000, the time is 128 msec. The first record in a trace always corresponds to time 0.

Resource type encoding: Table A1 contains the encoding of the resource field. Some codes apply only to traces from a specific environment (e.g., the PRMSEQ resource type appears only in the STEM traces.)

Table A1 contains the encoding of the resource field.

Table A1: Resource Type encoding. This table contains an overview of the data types represented in the traces.

Encoding	Resource Type	Trace Type	Encoding	Resource Type	Trace Type	Encoding	Resource Type	Trace Type	Encoding	Resource Type	Trace Type
0x00	Log	DB2	0x08	HSAM MDSG	IMS	0x10	PRMSEQ	GCOS	0x18	TSSJOUT	GCOS
0x01	Data	DB2	0x09	HDAM	IMS	0x11	PRMRND	GCOS	0x19	BUFMGR	GCOS
0x02	Index Root	DB2	0x0a	HIDAM	IMS	0x12	TMPSEQ	GCOS	0x1a	TS8PRM	GCOS
0x03	Index Leaf	DB2	0x0b	HI Index	IMS	0x13	TMPRND	GCOS	0x1b	TS8TMP	GCOS
0x04	Index Node	DB2	0x0c	HISAM	IMS	0x14	BATCHPGM	GCOS	0x1c	PRMNCAT	GCOS
0x05	HISAM SDSG	IMS	0x0d	SHISAM	IMS	0x15	TSSPGM	GCOS	0x20	DB2IXTMP	DB2
0x06	HISAM MDSG	IMS	0x0e	Index Uniq	IMS	0x16	TSSTMP	GCOS	0x21	DB2TSTMP	DB2
0x07	HSAM SSGM	IMS	0x0f	Index Mult	IMS	0x17	TSSUSR	GCOS	*0xff*	*unknown*	*all*

A6 Assumptions

Transactions and Processes: We assume that the transaction (in the context of a database trace) and the process (in the context of a timesharing system trace) are analogs. We feel this is a natural analogy for the following reason: both a transaction and a process correspond to a "unit of work" performed by the system on behalf of the user. In addition, in the traces we studied, the average amount of data accessed by transactions was roughly equal to that accessed by processes (See Table A2.) Note that in timesharing systems, processes may be "detached" from terminals to behave more or less like batch transactions, or may operate in the more typical interactive mode.

Database/Tablespaces and Files: We assume a database tablespace (i.e., a single table, in a relational database) corresponds to a file in timeshared systems. Tables contain related data and may each be accessed in a number of ways. They may be searched, sorted, or read in their entirety, much like files.

Disk Blocks: Our database traces each make references to 4-KB disk blocks. Some of the original traces (**Timeshare** and **Manufacture**) make references to variable-sized amounts of data. Therefore, we have converted these traces to make references which are to uniform 4-KB disk blocks as well. For example, a read of 1 KB is translated to single read of an aligned 4-KB disk block. A write of 9000 bytes is translated to three writes to aligned 4-KB disk blocks. Since disk cache is typically implemented assuming constant-sized cache blocks, we believe this is a valid translation.

A7 Replacement Policies

A7.1 Least-Recently Used Replacement Policy

The LRU ("Least-Recently Used") replacement policy is the standard implementable analog of the MIN policy. LRU takes advantage of knowledge of past reference patterns to attempt to predict future ones. Specifically, it assumes that a disk block used in the recent past will be used again in the near future. Likewise, it makes the assumption that disk blocks which have not been used in a long time will not be used for quite some time in the future.

The contents of an LRU-managed cache are simply as many of the most recently used disk blocks as will fit into the cache. Thus, for reference string s, the contents of an LRU-managed cache of size b at time t (before reference (X_t , F_t, B_t) has been serviced,) denoted by LRU , is:

$$LRU(s, t, b) = \{B_{t-n}, B_{t-n+1}, \ldots, B_{t-1}\} \tag{1}$$

with n such that $|LRU| = b$.

LRU chooses a block to replace based on the priority ordering relation below. Note that in this discussion as in the discussion of MIN above, the block with the highest priority number is the one which is replaced.

$$priority_t(B) = (t-i) \text{ such that } (B_i = B) \text{ and } (B_{i'} \neq B) \text{ for all } (i < i' < t) \tag{2}$$

That is, the priority of a block can be thought of simply as the time since it was last referenced.

A7.2 MIN Replacement Policy

The MIN replacement policy (also referred to, variously, as "OPT" or "Optimum") is the standard "best-case" policy. However, although it yields the lowest miss ratio possible, it is unrealizable in the sense that it depends on knowledge of future reference patterns.

MIN chooses a block to replace based on the priority ordering relation below. At time t,

$$priority\,(B)_t = i \text{ such that } (B_i{=}B) \text{ and } (B_{i'} \neq B) \text{ for all } x < i' < i \tag{3}$$

That is, the priority number of a block can be thought of simply as the time until it is next referenced. Thus, the block in the cache with the highest priority number (used farthest in the future,) is the one chosen for replacement.

A7.3 VMIN Replacement Policy

The VMIN replacement policy ([Prie76]) is also a standard "best-case" policy, this time for variable-allocation strategies. Like MIN, it is unrealizable in the sense that it depends on knowledge of future reference patterns.

On a miss, VMIN removes from the cache those blocks which won't be used for τ references. Since multiple blocks may be discarded on a miss, the size of the buffer (i.e., the number of disk blocks cached) will vary with time. This is thought to make more efficient use of memory, assuming the memory freed can be effectively used elsewhere (e.g., for page frames, in a system implementing virtual memory.) Clearly, in a system with *de facto* fixed allocation, for example in a disk cache implemented in the disk controller, a fixed-allocation scheme is more appropriate. The parameter controlling which disk blocks are cached is based on an operating parameter called τ, which represents how far into the future the policy looks. The following equation gives the set of blocks to be replaced ("yanked") on a miss at time t:

$$\mathrm{Y}(\mathrm{t}, \tau) = \{B \text{ such that } (B_i = B) \text{ and } (B_{i'} \neq B) \text{ for all } (i < i' < t)\} \tag{4}$$

That is, the set of blocks replaced comprises those which will not be referenced in the next τ references.

VMIN is to variable-space allocation strategies as MIN is to fixed-allocation strategies: no other demand-fetch variable-allocation replacement policy can yield a lower miss ratio.

Results

We simulated the various reference strings using the VMIN replacement policy, for varying sizes of τ. They are plotted, however with respect to the average size, in megabytes, of cached blocks in the appendix, Figure A6. The curves are very similar to those for the MIN policy.

A7.4 Working Set Replacement Policy

The Working Set replacement policy ([Denn68]) uses the notion of the working set, the set of disk blocks which have been referenced in the last τ references, where τ is the operational parameter.

Generally, the Working Set policy stops caching those blocks which have not been referenced in the time window τ. Therefore,

$$C(t, \tau) = \{B \text{ such that } B_i \text{ and } \langle t - \tau < i < t \rangle\} \tag{5}$$

That is, the set of blocks cached comprises those which were referenced in the last τ references.

Results

Figure A4 shows the results of simulating the various reference strings using the WS replacement policy, for various sizes of τ. We plot the curves with respect to the average working set size.

An interesting observation, given the results of our informal looping study discussed in Section A8.3, is that **Timeshare** and **Transport** do not also exhibit the lowest miss ratio under LRU replacement policy. No overwhelming trend exists, either, with respect to the database program, although the IMS traces, **Oil** and **Monarch**, generally exhibited better locality measured in terms of LRU, MIN and Working Set miss ratios than the DB2 traces.

Contrary to Smith's findings in [Smit76], we did not find that WS consistently outperformed LRU with respect to the average WS cache size, as seen in Figure 4 in the curves of the ratios of miss ratios for the LRU and WS replacement policies. Note that **Oil**, **Telecom**, and, at larger cache sizes, **Transport** and **Monarch**, have lower LRU miss ratios than WS miss ratios. We note that these traces also have large ratios of maximum to average working set size (the curves appear as Figure A8.) In Smith's study, the ratios of maximum to average working set size does not exceed two. In our study, this ratio exceeded two for each of the traces for which LRU appears to exceed WS. We theorize that these large spikes in the WS size inflate the average WS size to a point which the corresponding Working Set miss ratio exceeds that of a LRU cache of the same size.

Working Set Miss Ratio

Another way to measure locality is to measure the miss ratio of a cache managed with a policy based on the Working Set memory management policy, a variable-space caching algorithm in which just the disk blocks referenced in the last τ references are buffered, where τ is the operating parameter. That is, for a reference string s,

$$WS(s,t,\tau) = \{B_{t-\tau}, B_{t-\tau+1}, \ldots, B_{t-1}\} \tag{6}$$

We define the Working Set Miss Ratio to be the ratio of references to a disk block in the working set to the total number of references. Formally, we define an indicator function, r_{WS} associated with references such that it equals one if the reference is a member of the working set at that time:

$$r_{WS}(s, t, \tau) = \begin{cases} 1 & if\ B_i \in WS(s, t, \tau) \\ 0 & otherwise \end{cases} \tag{7}$$

then the miss rate is

$$MR_{WS}(s, \tau) = 1 - \frac{1}{|s|} \sum_{i=1}^{|s|} r_{WS}(s, i, \tau) \tag{8}$$

Results

The Working Set miss ratios for the traces studied are presented in Figure A4. The figure shows the Working Set miss ratio as a function of the operational parameter τ, expressed in terms of number of references. Not surprisingly, since the working set policy is a variable-space analog of the Least-Recently Used policy, the curves look similar to the LRU curves.

A7.5 Prefetch Traffic Ratio

We calculated traffic ratios for caches managed with prefetch. We define the function representing the number of pages prefetched for reference string s at the time of a given page reference, t, to be $P(s, t)$. We further define an indicator function $r(s, t)$ to be equal to one if a reference from s at time t is not in the cache, and zero otherwise. We then define the traffic ratio to be the ratio of the sum of references to blocks not found in the cache (i.e., misses, represented by $r(s, t)$) and prefetched pages to the total number of references $P(s, t)$. Formally, the traffic ratio for a cache managed with prefetch function P is:

$$TR(s, P) = \frac{1}{|s|} \sum_{i=1}^{|s|} (r(s, i) + P(s, i)) \tag{9}$$

A8 Run Length

A8.1 Run Length Example

We represent a reference by the triple (X_t, F_t, B_t) where X represents the transaction making the reference, F, the file referenced, and B the file block referenced. Consider the following reference string: {(*X2, F1, 1*), (*X1, F1, 2*), (*X1, F2, 6*), (*X2, F1, 4*), (*X1, F2, 7*), (*X2, F1, 5*), (*X1, F1, 3*)}. If we retain the file information only, as in [Smit78], we can rewrite the reference string as {(*F1, 1*), (*F1, 2*), (*F2, 6*), (*F1, 4*), (*F2, 7*), (*F1, 5*), (*F1, 3*)}. If, for each file opened by any transaction, we keep track of the last page referenced and the current run length, we measure the following run lengths: *2, 2, 2, 1* . For this reference string, we measure even longer runs using this method, which we call *file* .

We can also retain both the file and transaction information and keep run length information for each transaction-file pair. The following run lengths result: *2, 2, 1, 2* . For this reference string, both this method, which we call *xact_file* , and the *aggregate* method result in the same run lengths.

We gathered statistics on runs using the methods we defined inSection 5.4.2; they appear in Table A2. Generally, the longest runs resulted by detecting runs on a per-file basis (that is, using the *aggregate* method above) or on a per-transaction-and-file basis (*xact_file* .) Detecting runs on a per-transaction-and-file basis does not always result in the longest runs since it has the effect of ending runs at the end of transactions. If, for example, two overlapping transactions share a single-block file, detecting runs on an aggregate basis will result in a longer run. **Transport** , **Telecom** and **Manufacture** are notable, therefore, in that the average *xact_file* run length is longer than the average *aggregate* run length. These traces have a much higher degree of file sharing than the other traces, with 38%, 9.6% and 1% of references, respectively, being to files which were at the time of the reference being shared by 10 or more concurrently-running transactions. Moreover, these shared files tended to be quite large, so measuring runs on a per-file basis only was likely to yield shorter runs as interleaved references by the concurrent transactions broke the *aggregate* runs.

To illustrate this with an example, consider the following sequential working set:

$$SWS(s, t_i, \tau) = \{1, 2, 4, 5\} \tag{10}$$

Assume that the reference string s and time window τ are such that the sequential working set contains these same 4 disk blocks for the next 3 references.

Now assume a reference string which is defined to be the reference string s, above, concatenated with the substring *{5, 6, 7}* .

None of the references in the resulting substring, which clearly displays a high degree of sequentiality, will fail to be a member of the sequential working set, since the reference to disk block *5* causes disk block *6* to become a member of the Sequential Working Set, and the reference to disk block *6* brings disk block *7* into the Sequential Working Set.

Now consider a new reference string, which we will define to be the original reference string s, as above, concatenated with a new substring, *{2, 5, 4}* .

Since we said the sequential working set would include blocks *1*, *2*, *4*, and *5* for the duration of the new substring, and since all of the disk blocks referenced in the new substring are members of the sequential working set, the reference string, while clearly not sequential in the strictest sense, does display a form of sequentiality which can be exploited by a disk cache.

We consider this flexibility of the Sequential Working Set to be a strength since the sequential working set clearly gives us a relative indication of the potential usefulness of prefetching; in this case, of a lookahead prefetch. Consider the fact that there need not be a requirement of absolute sequentiality for a prefetching strategy to be effective. There need only be a general sequentiality of the form measured by the sequential working set in which disk blocks sequentially following the disk blocks recently referenced are requested while the predecessor is still in the cache, regardless of their absolute order.

A8.2 Average Remaining Run Length

In this experiment, we calculated the average remaining run length after having observed a run of a given length by using the distribution of run lengths we generated in the previous experiment. Figure A3 shows the average remaining run lengths for the traces. For a point *(x, y)* on the graph, when a run of length *x* has been observed, *y* represents the average of the remaining lengths of runs of length *x* and greater. The labels of the curves are consistent with the labels used in Table A2.

A8.3 Looping

However well sequentiality responds to prefetching, long runs have the undesirable property of flushing caches, and unless prefetching is implemented, do not benefit at all by the presence of a cache if the disk blocks in the run are never rereferenced after a miss loads them into the cache. However, if the blocks are indeed rereferenced before being replaced, a performance improvement will be seen upon implementation of a cache.

Perhaps the simplest and most common case of reference patterns involving sequential runs and rereferencing is the *loop*. A loop is, in our context, a set of disk blocks which are referenced sequentially a multiple number of times. Therefore, it can be thought of as a series of sequential runs (which we define to be toward increasing logical block address) separated by negative jumps back to the first block in the loop. Kobayashi uses a similar definition of looping in [Koba84] in the context of an instruction execution string. He finds looping to be quite common in this context.

Clearly, if the blocks comprising a loop can fit into a cache, and if interference from the references by other transactions or to other files by the same transaction is kept to a minimum, the mean access time per disk block in the loop decreases with the number of times the loop is traversed: the disk blocks will be loaded into the cache on the first traversal of the loop, and future traversals of the loop will find requests for disk blocks to be serviced out of the cache.

We modified the program measuring run lengths by to measure the degree of looping behavior by adding an indication of what broke the run. Specifically, if a run was broken by a reference to a previous block in the run, we detected a loop. Note that this definition is relatively restrictive in the sense that it precludes any forward jumps in the body of the loop: skipping one block in a loop will not only end the sequential run, but will also break the loop. However, we measured this for each file in each transaction separately, so interleaving references by other transactions or to other files by the same transaction would not break the loop.

As a disk cache represents a cache of data references only, it would not be surprising to see little looping behavior. Loops are far more common in code references than in data references.

Results

Generally, we found looping behavior to be relatively rare in the traces, despite the fact that some them exhibited high degrees of locality as measured by the LRU miss ratio. **Transport** contained a file which was accessed repeatedly in loops of length 16 by 27 transactions, a single transaction having traversed the loop for 1955 of the 2650 total traversals of that loop. These traversals, however, accounted for only 6.2% of the total references. The other traces exhibited even less looping behavior. **Timeshare**, however, included several reference strings which, while not behaving as loops in the strictest sense, exhibited a predictable, generally increasing reference pattern. One string, for example, consisted of short runs of length 5. While the runs were, of course, sequentially increasing, a subsequent run would begin in the middle of a previous run. These strings were anomalous in the sense that they resembled something between a long run and a loop.

Table A2: Run Length Statistics *rows marked **aggregate** refer to runs measured on a per-file basis, and **xact_file** to runs measured on a per-transaction and -file basis. Numbers in parentheses correspond to measurements taken with immediate re-references ignored.*

	Method	Transport	Bank	Telecom	Timeshare	Manufacture	Oil	Monarch
Number of Runs	***aggregate***	223481	387188	1860571	265281	310712	87209	898960
	xact_file	218938	386902	1837529	267171	308023	89897	927056
Mean Run Length	***aggregate***	2.47 (1.92)	1.17 (1.15)	1.32 (1.16)	2.15 (1.51)	1.77 (1.25)	7.89 (2.48)	2.83 (1.64)
	xact_file	2.53 (1.96)	1.17 (1.15)	1.34 (1.17)	2.13 (1.51)	1.78 (1.25)	7.66 (2.43)	2.74 (1.62)
Median Run Length	***aggregate***	1	1	1	1	1	2	2
	xact_file	1	1	1	1	1	1	2
Maximum Run Length	***aggregate***	2743 (2743)	1155 (115)	3649 (2136)	830 (89)	14476 (4826)	1055 (213)	6700 (133)
	xact_file	2743 (2743)	1155 (1155)	4464 (4464)	827 (89)	14476 (4826)	1055 (213)	631 (133)

A9 Figures

Figure A1: Cumulative percentage of data touched. The symbols indicate the point chosen to represent warm start for each trace.

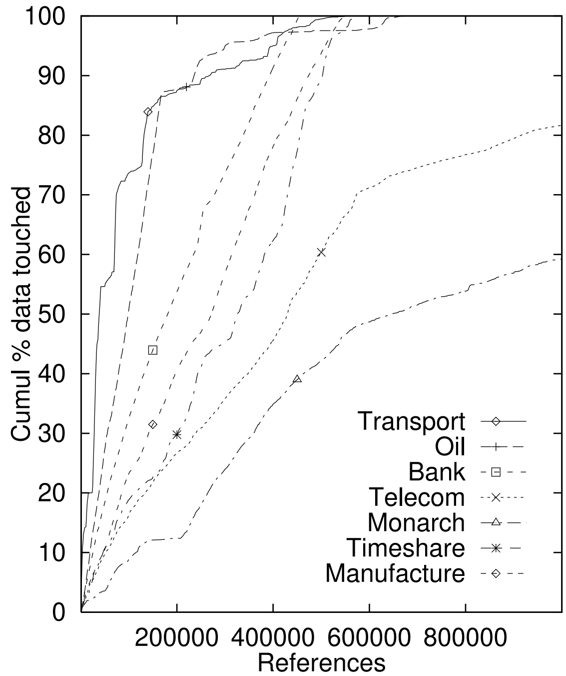


Figure A2: Cumulative percentage of data touched with respect to percentage of trace length.

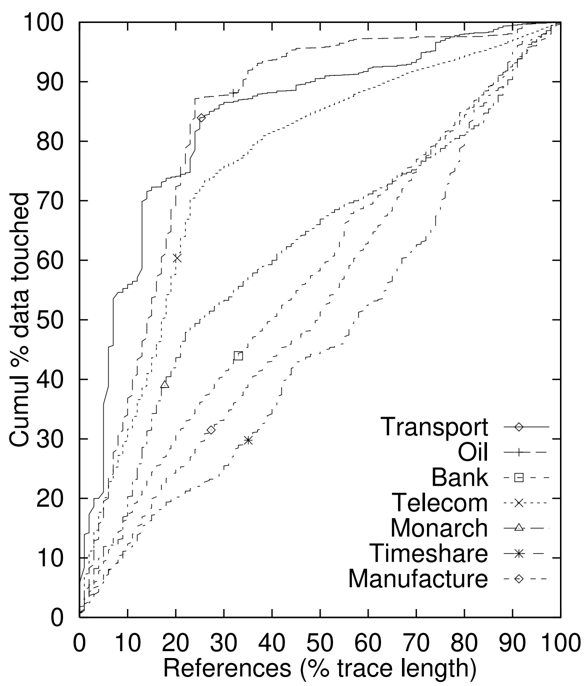


Figure A3: Average remaining run length for runs measured on an aggregate and per-file and -transaction basis. *For a point (x, y) on the graph, when a run of length x has been observed, y represents the average of the remaining lengths of runs of length x and greater.*

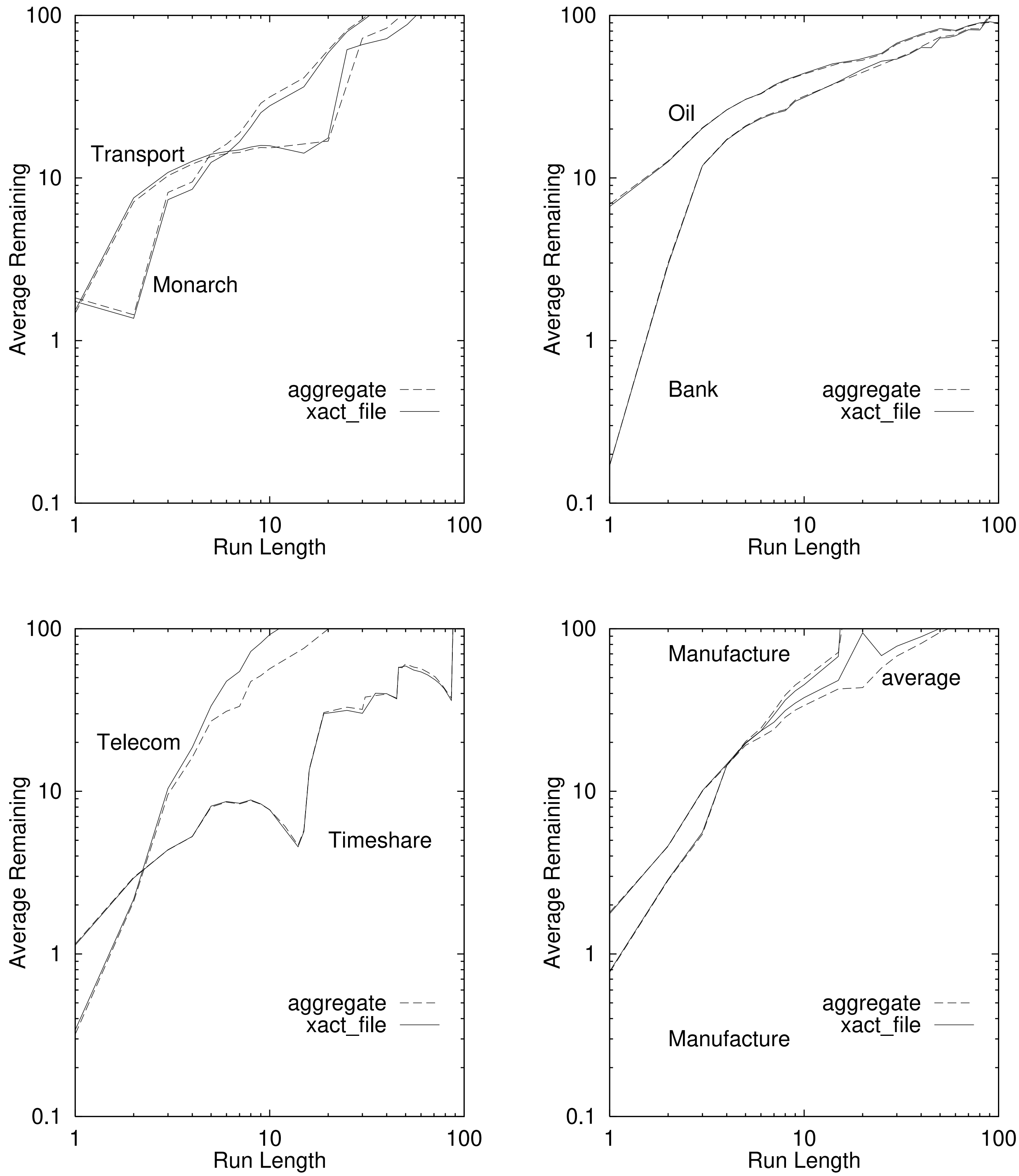


Figure A4: Working Set miss ratio for τ in terms of references and time.

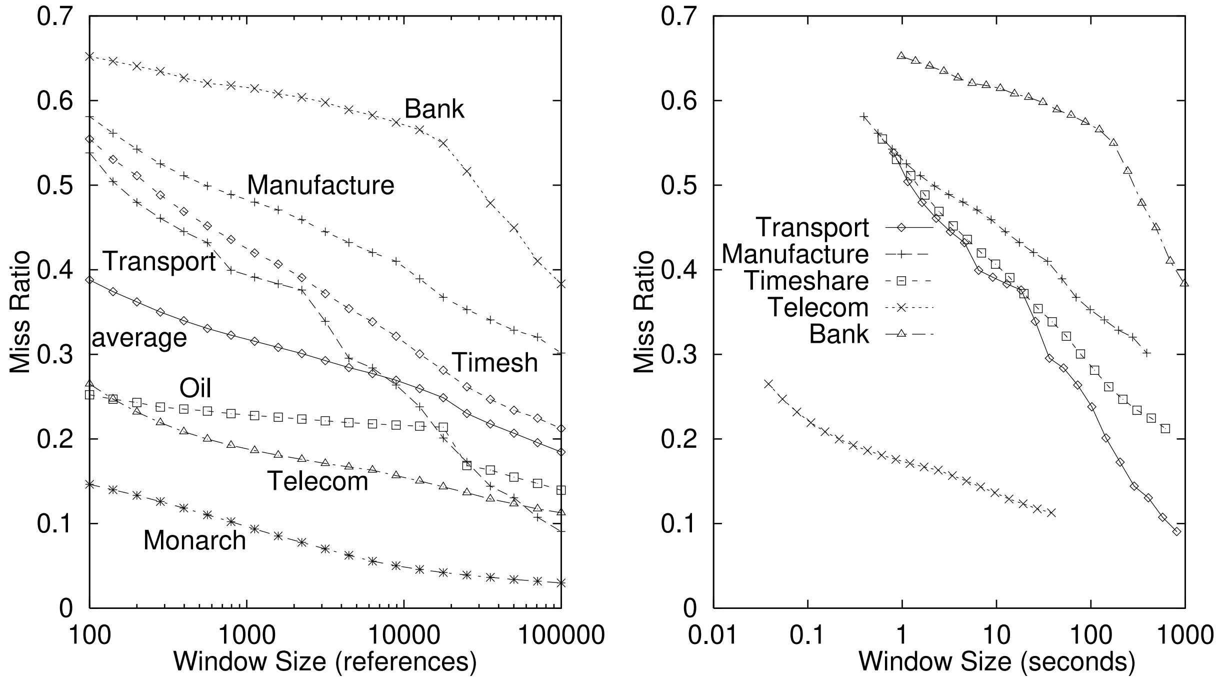


Figure A5: Sequential working set miss ratio for various values of working set window size, τ.

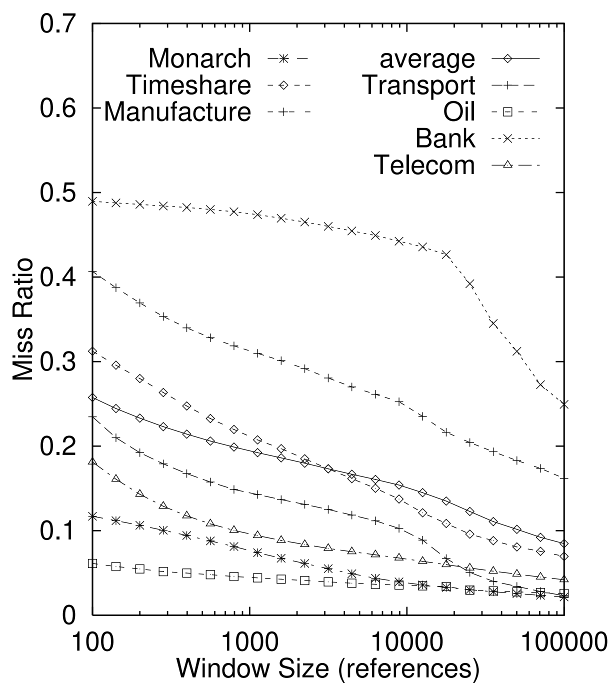


Figure A6: MIN (left) and VMIN (right) miss ratios for various cache sizes.

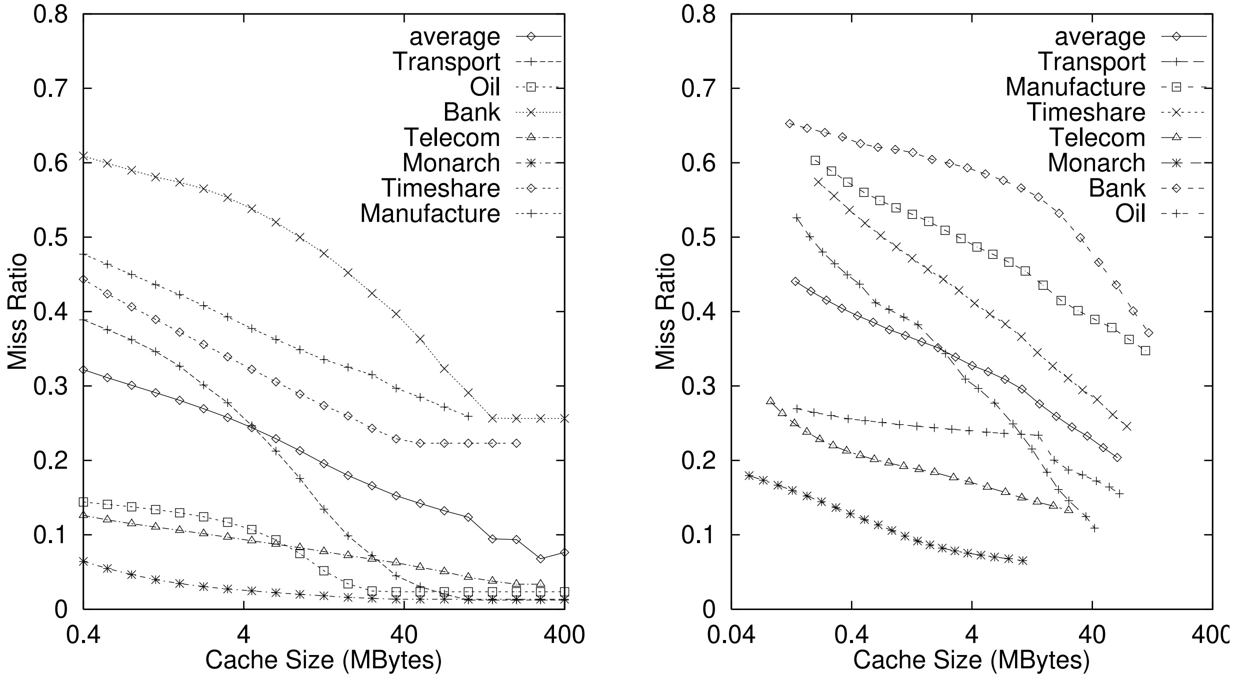
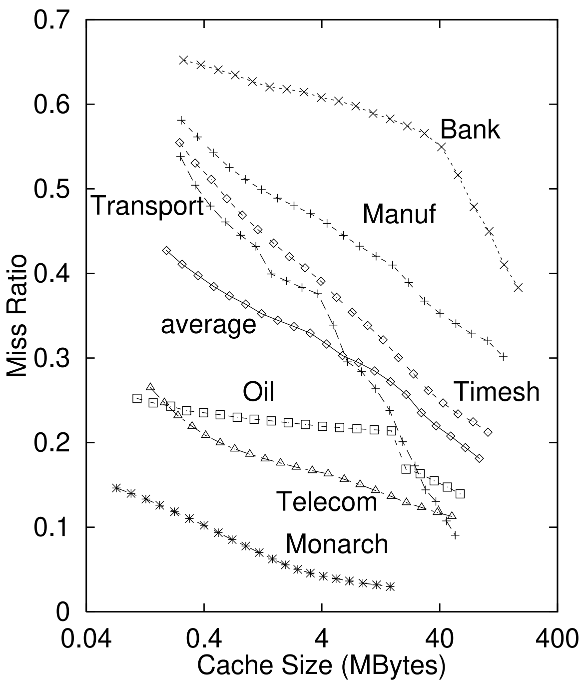


Figure A7: Working Set miss ratios for various cache sizes.



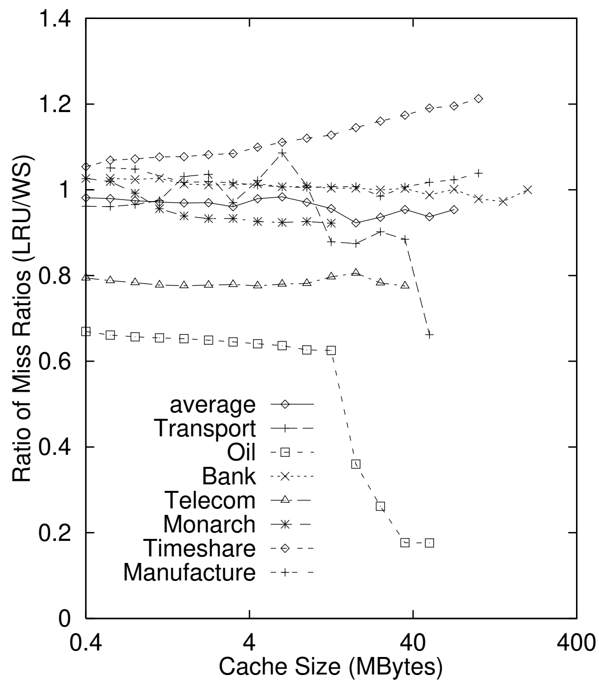


Figure A8: Ratio of maximum working set size to average working set size.

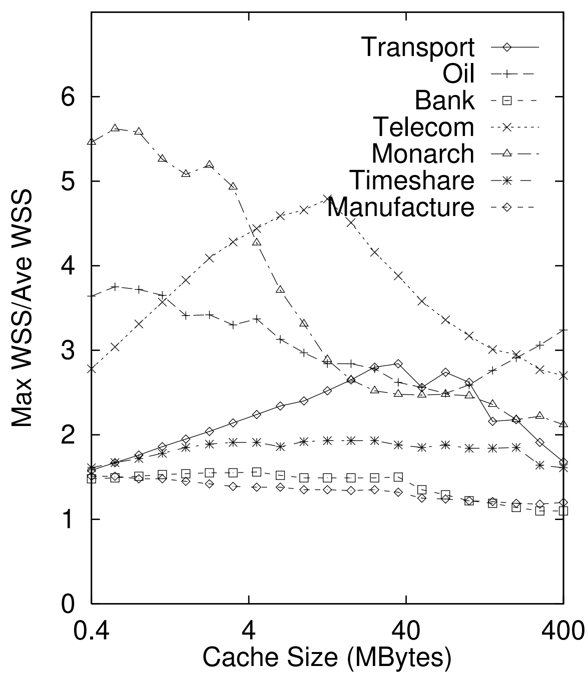


Figure A9: Lookahead prefetch miss ratios for various amounts of lookahead.

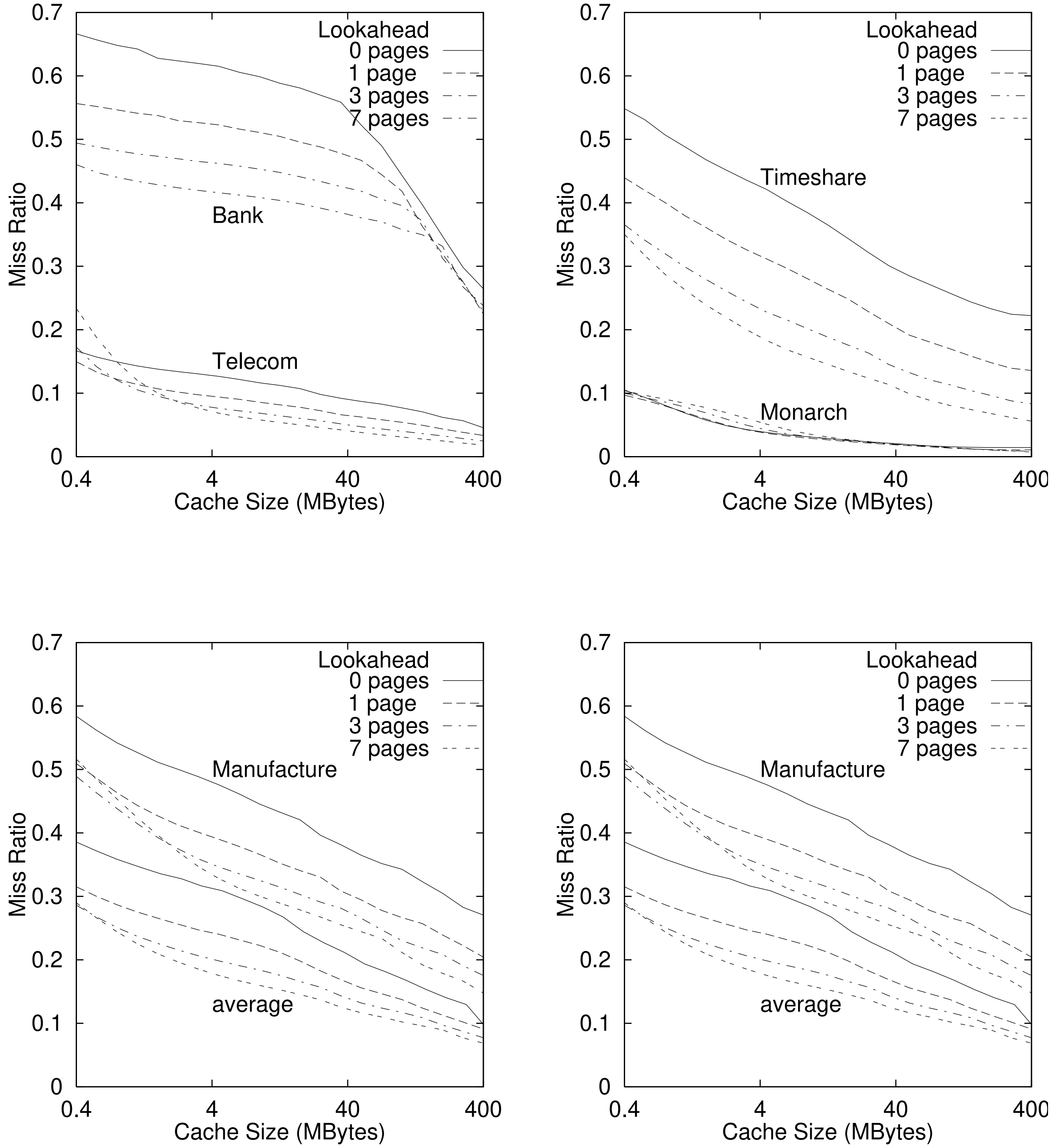


Figure A10: Lookahead traffic ratios for various amounts of lookahead.

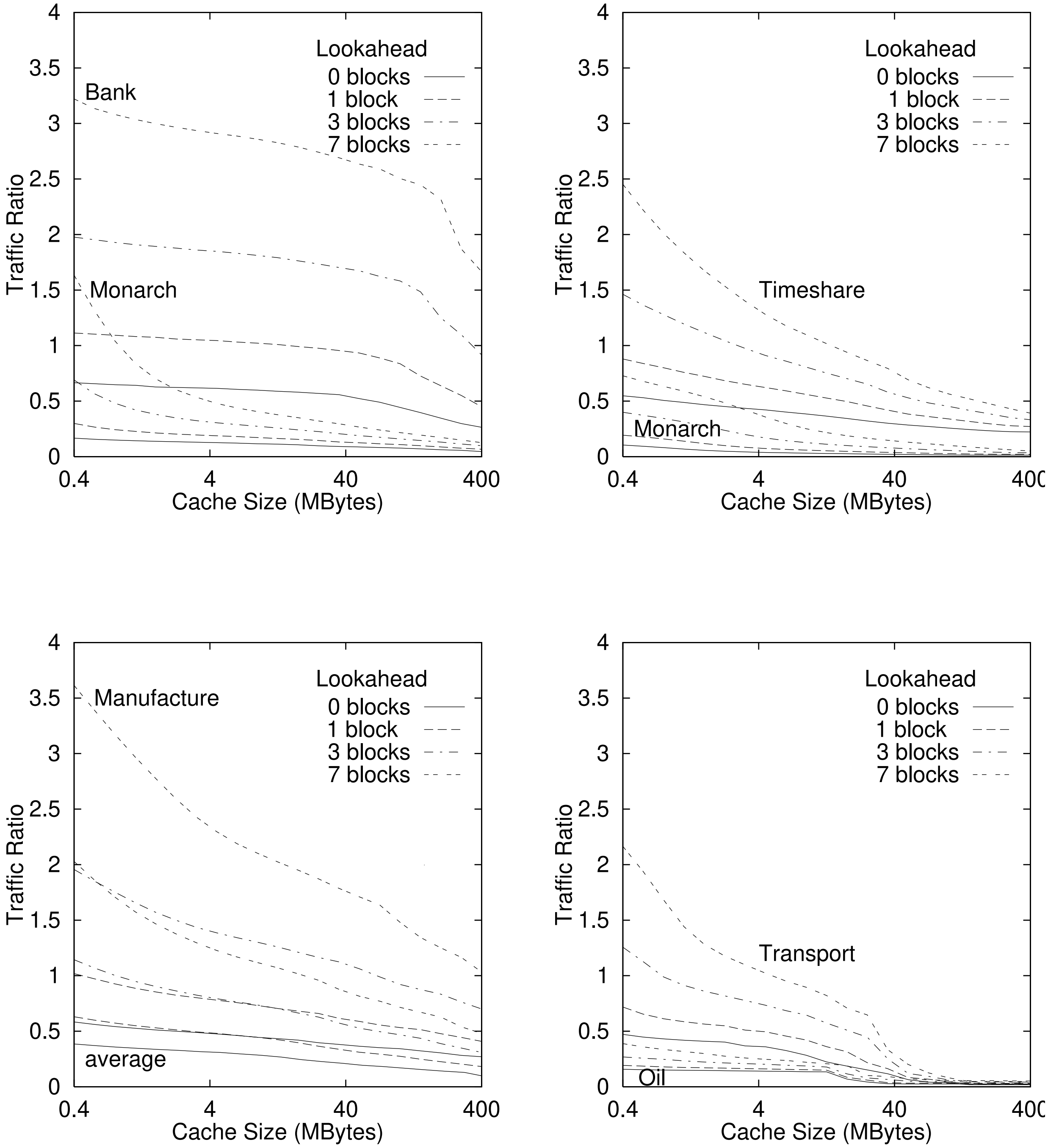


Figure A11: Variable prefetch miss ratios compared with no prefetch and lookahead prefetches of 1 and 3 disk cache blocks. Curves labeled *xact_file* refer to runs detected on a per-transaction and per-file basis. Curves labeled S*WS* refer to runs detected using the *sequential working set* method.

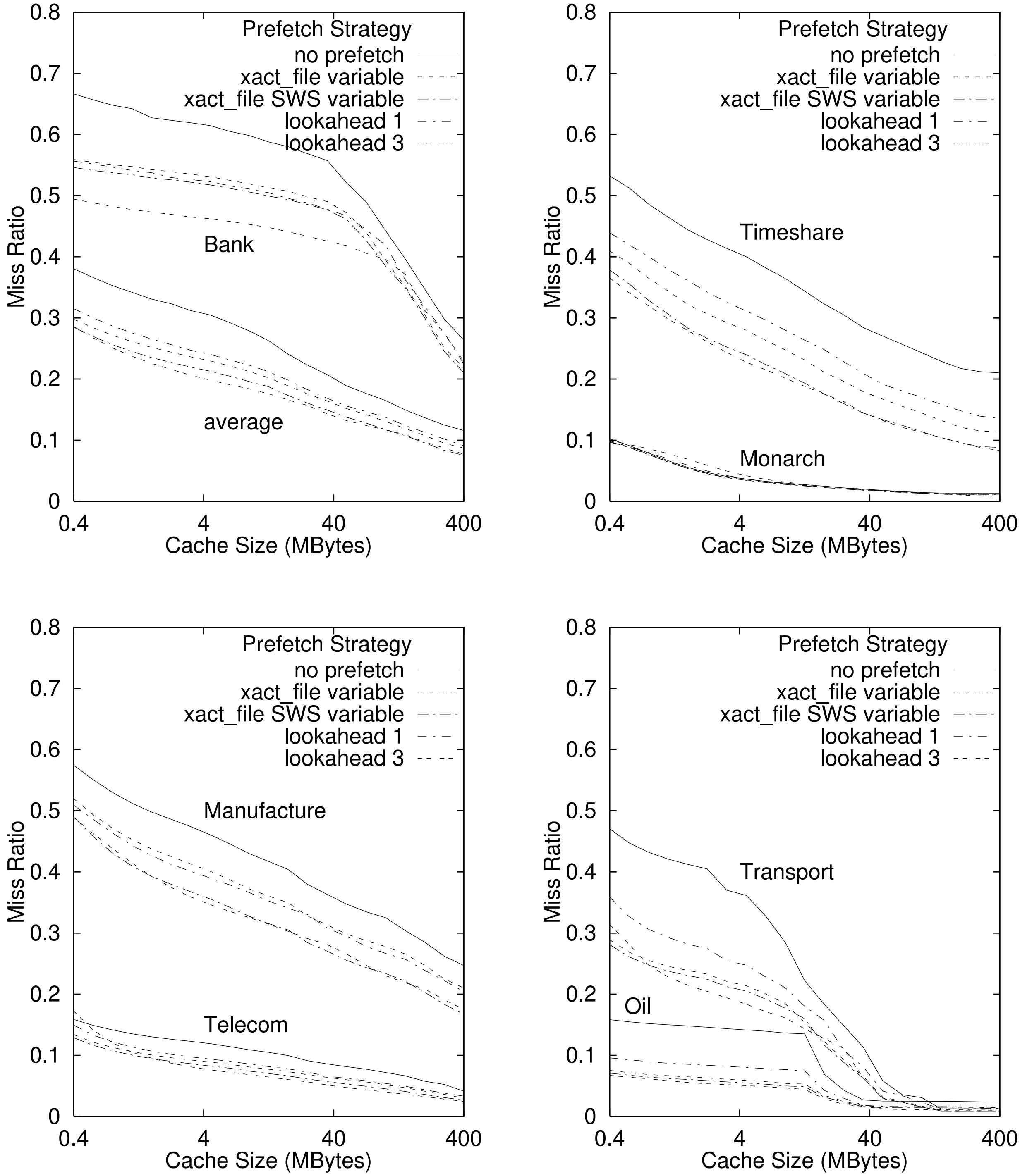


Figure A12: Variable prefetch traffic ratios compared with no prefetch and lookahead prefetches of 1 and 3 disk cache blocks. Curves labeled *xact_file* refer to runs detected on a per-transaction and per-file basis. Curves labeled *SWS* refer to runs detected using the *sequential working set* method.

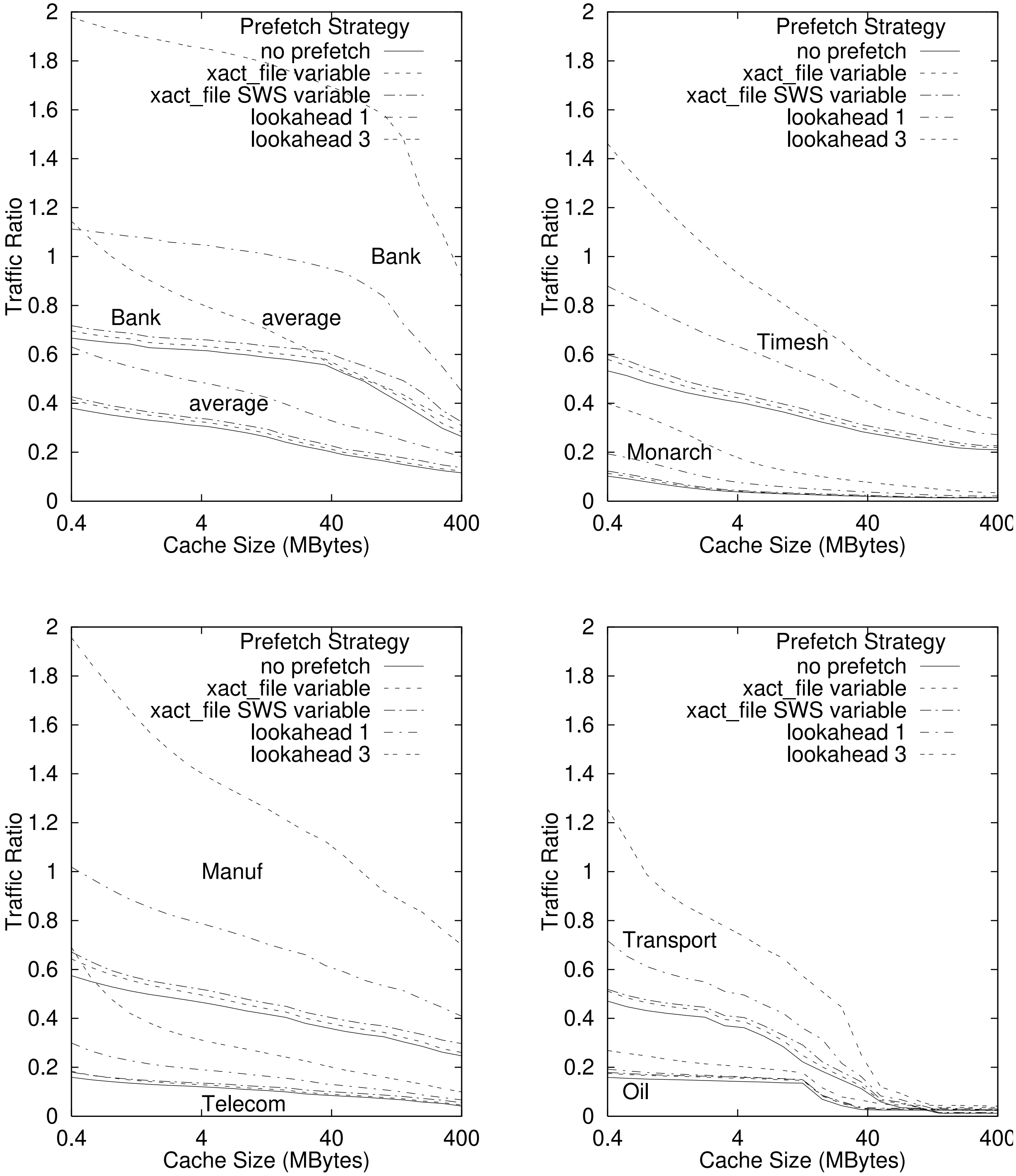


Figure A13: Percentage of prefetched pages referenced before being replaced. Curves labeled *SWS* refer to runs detected using the *sequential working set* method.

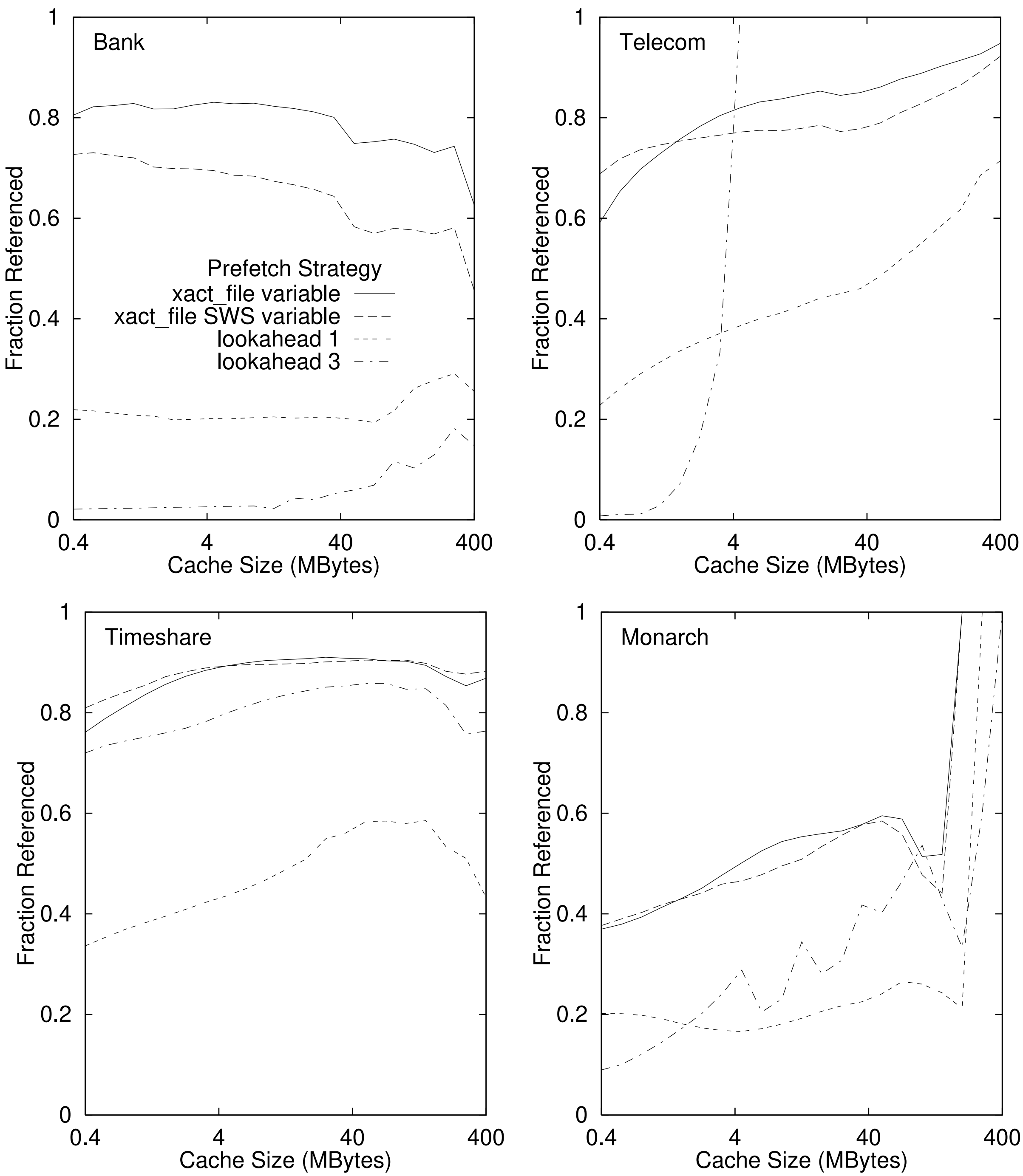


Figure A14: Percentage of prefetched pages referenced before being replaced, cont.

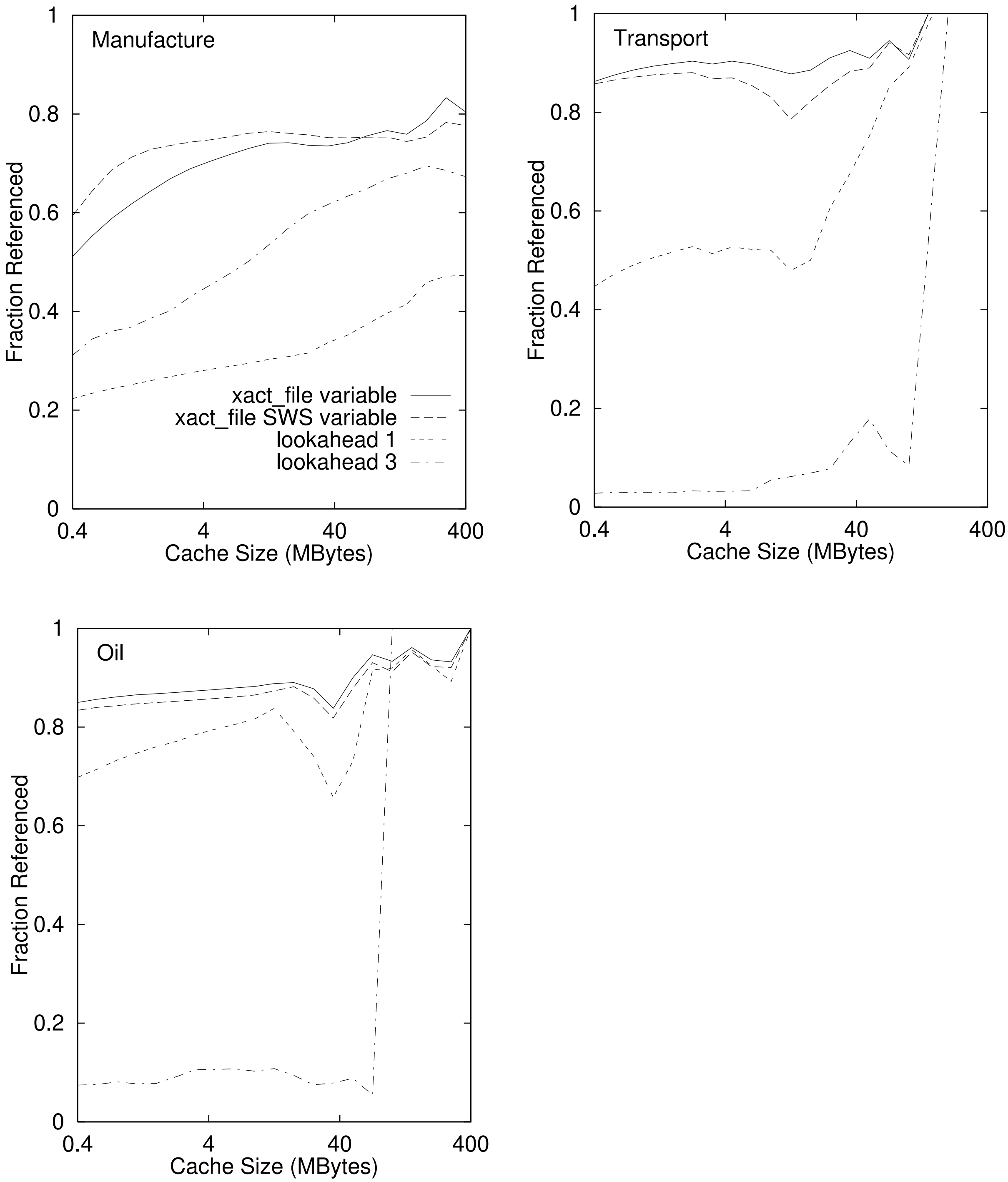


Figure A15: Miss ratios of caches with various block sizes. Compare with miss ratios of caches managed with per-file and -transaction variable prefetch (curves labeled *SWS* refer to the *sequential working set* method of detecting runs.)

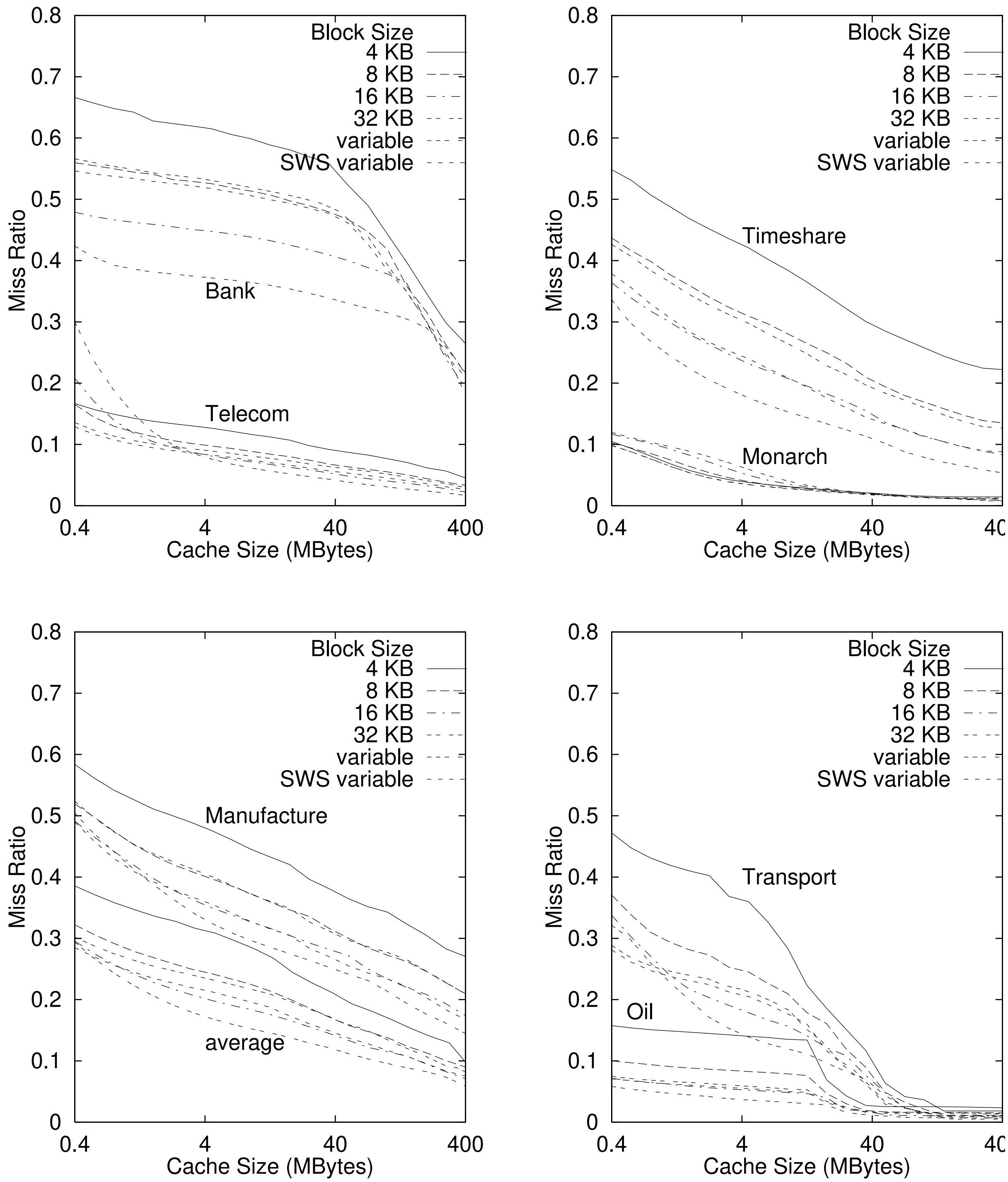


Figure A16: Traffic ratios of caches with various block sizes. Compare with miss ratios of caches managed with per-file and -transaction variable prefetch (curves labeled *WS* refer to the *working set* method of detecting runs.)

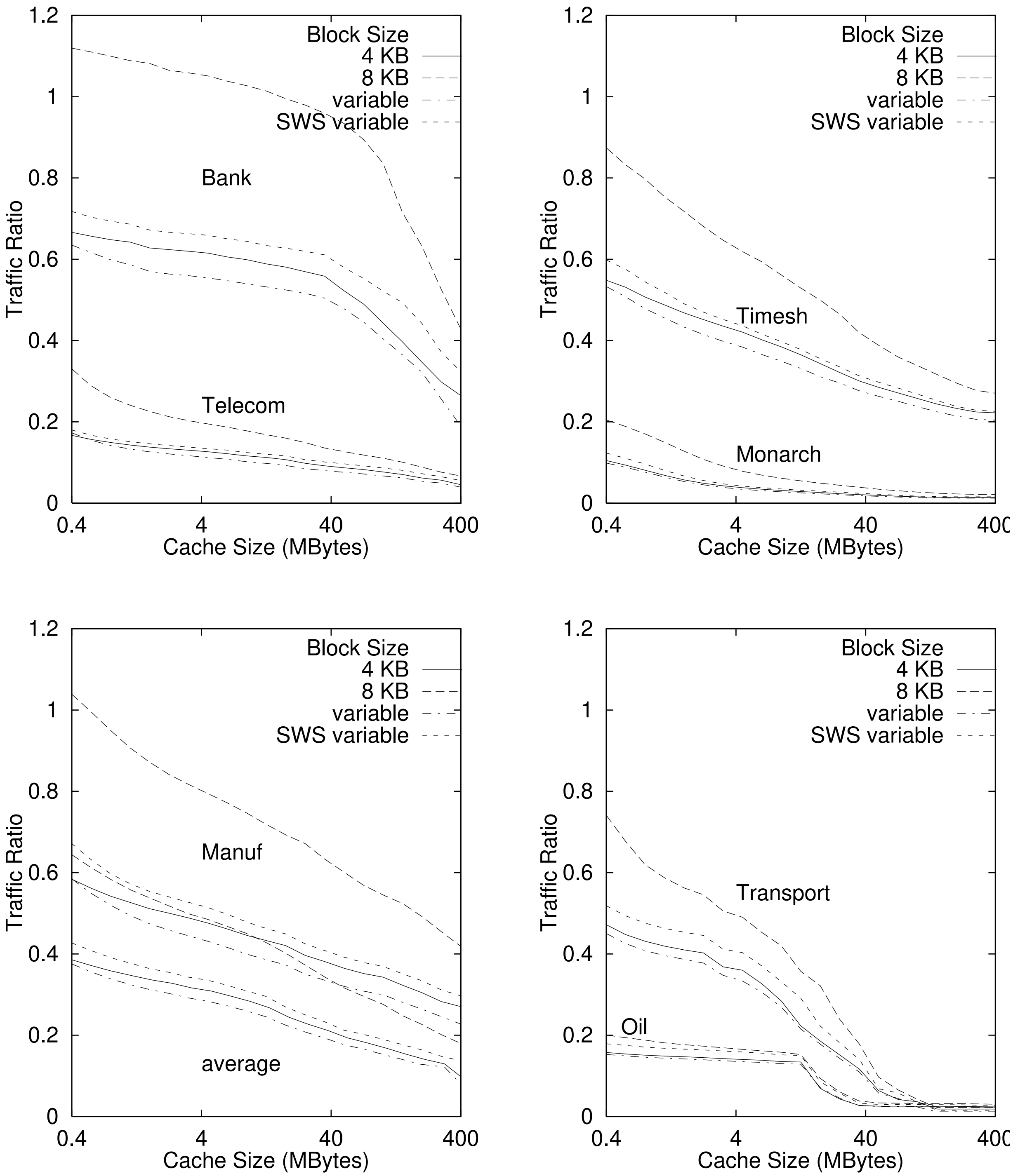


Figure A17: Write-allocate and no-write-allocate write-through write policy read miss ratios.

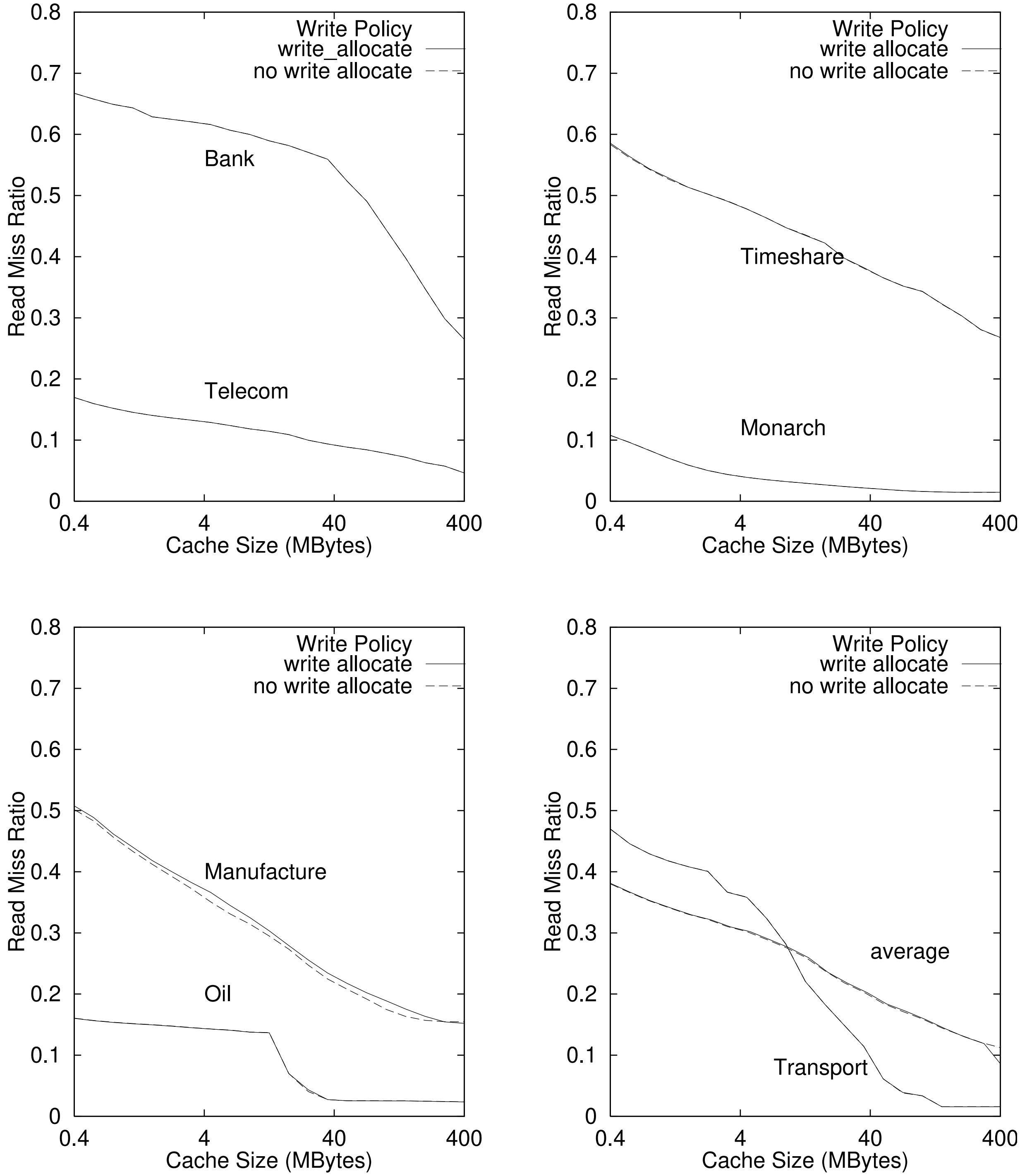


Figure A18: Ratio of write-through writes to write-back writes.

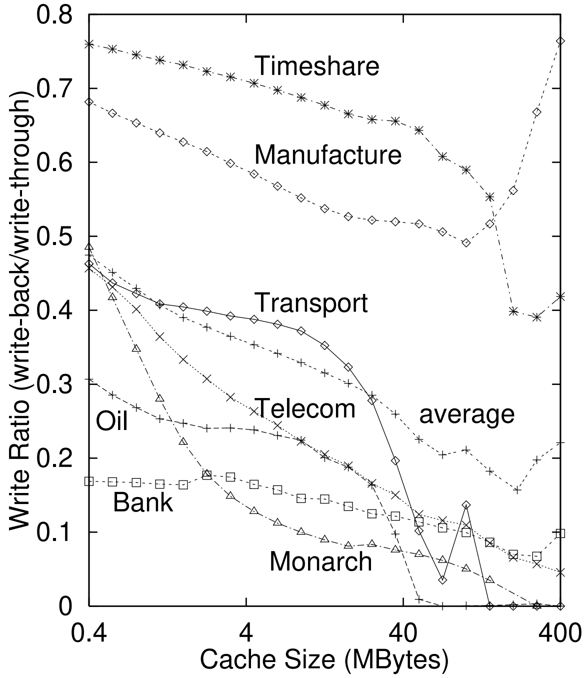


Figure A19: Traffic ratios for write-back and write-through caches.

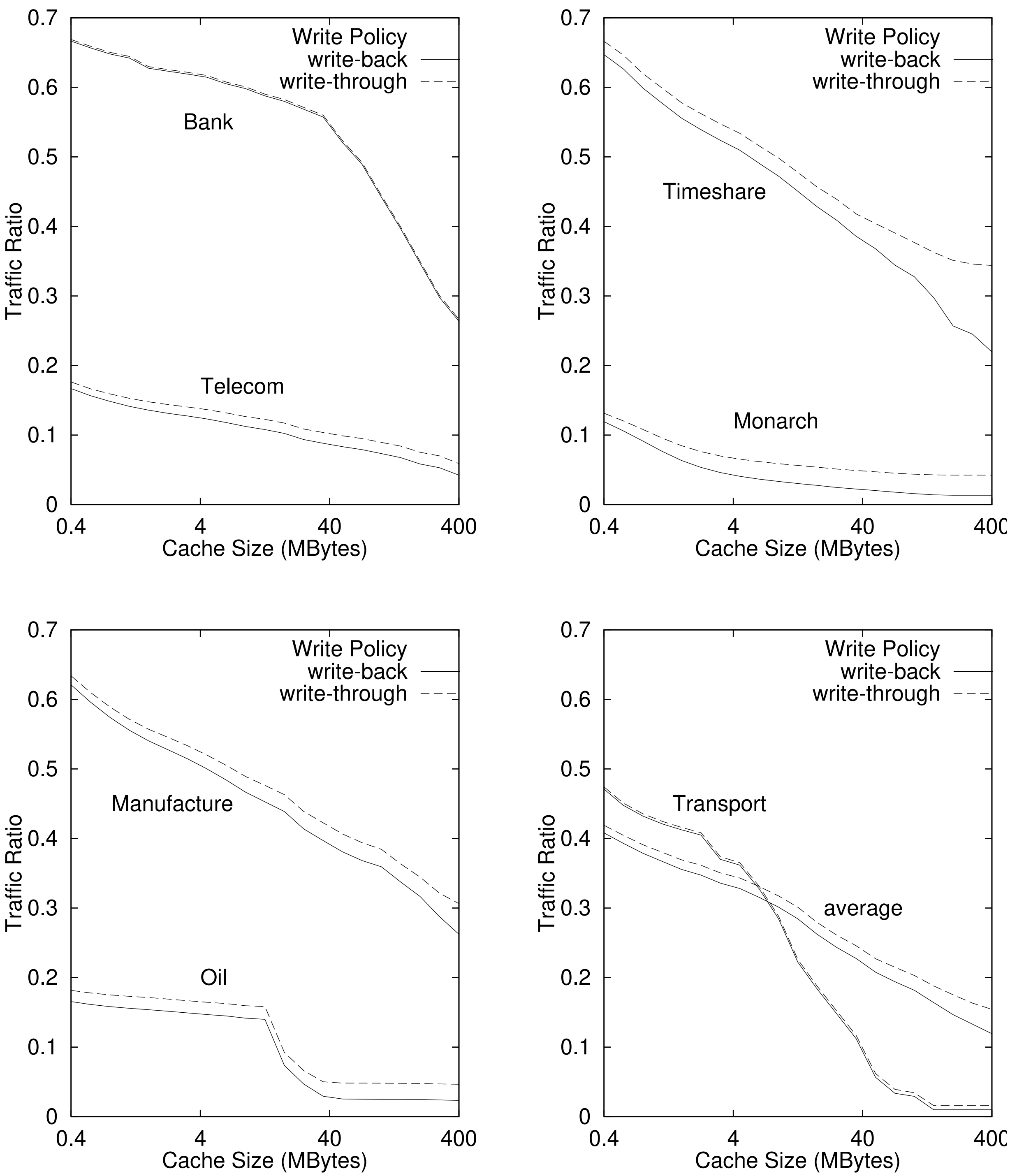


Figure A20: Delayed write policy traffic ratios for various cache sizes.

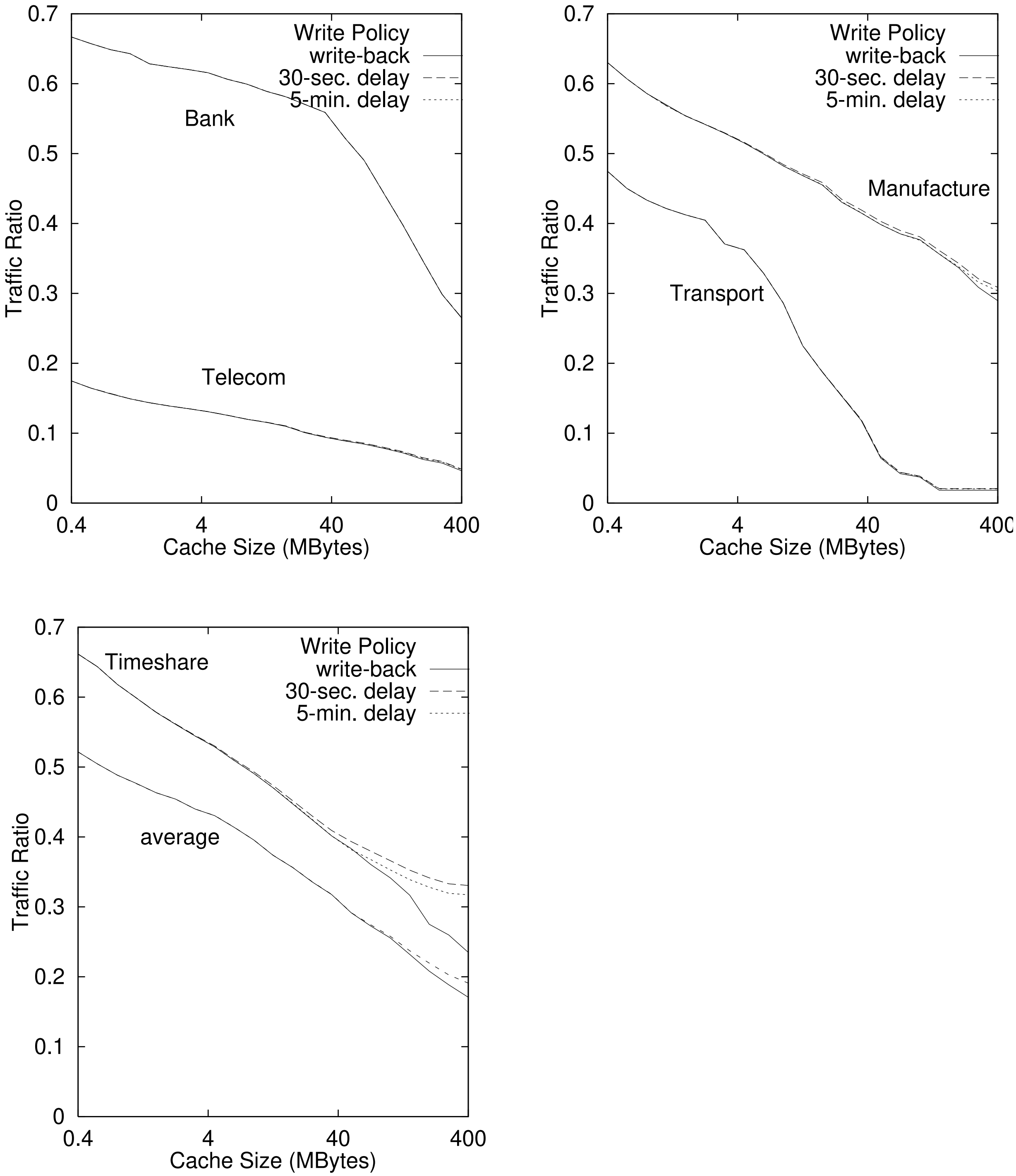


Figure A21: Average lifetime of blocks after they are modified.

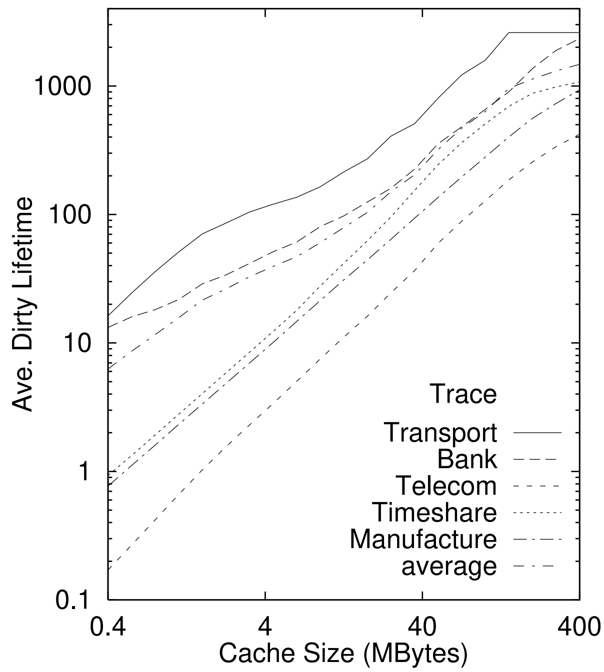


Figure A22: Fraction of dirty blocks which are referenced before being replaced.

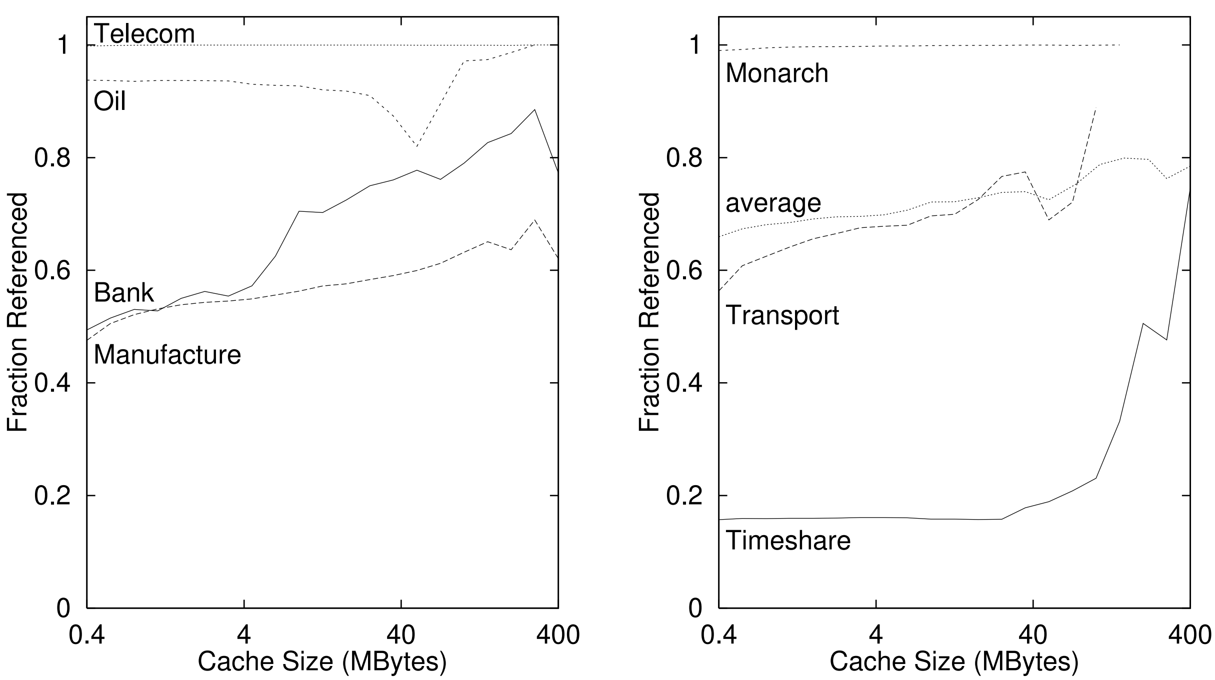


Figure A23: Percentage of cache blocks dirty upon replacement.

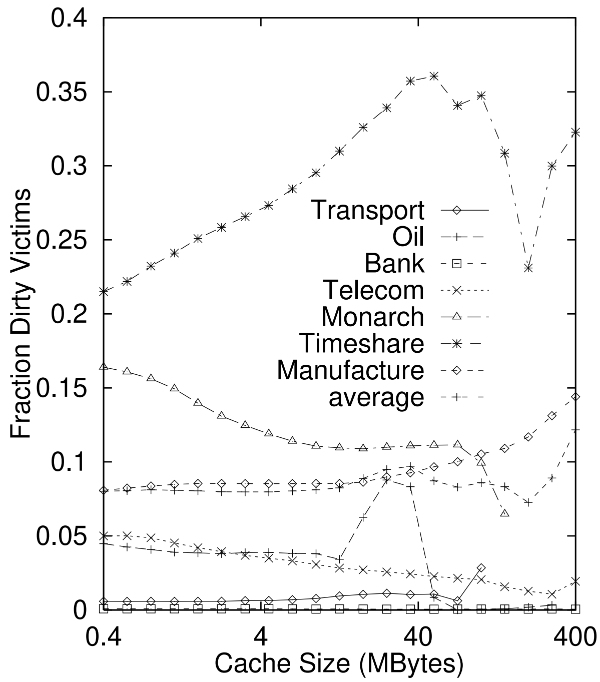


Figure A24: Cumulative fraction of writes. *For a point (x, y) on the graph, y is the percentage of writes to blocks which are written x or fewer times.*

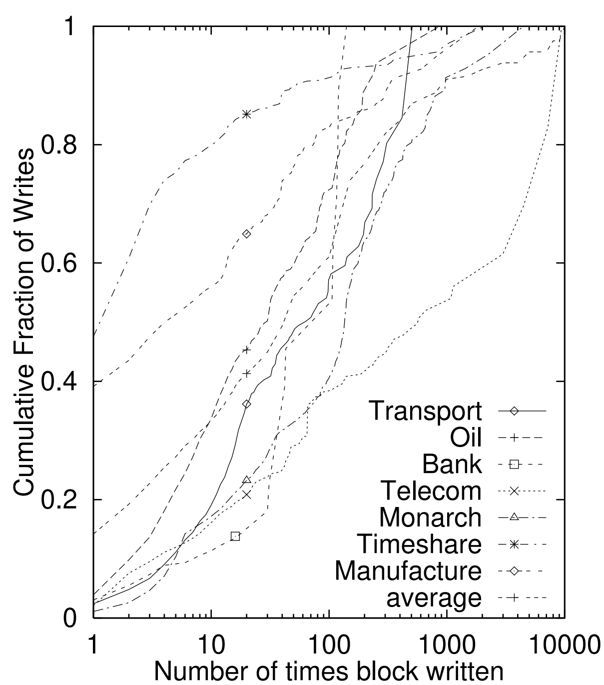


Figure A25: Traffic ratios for mixed, write-back and write-through write policies.

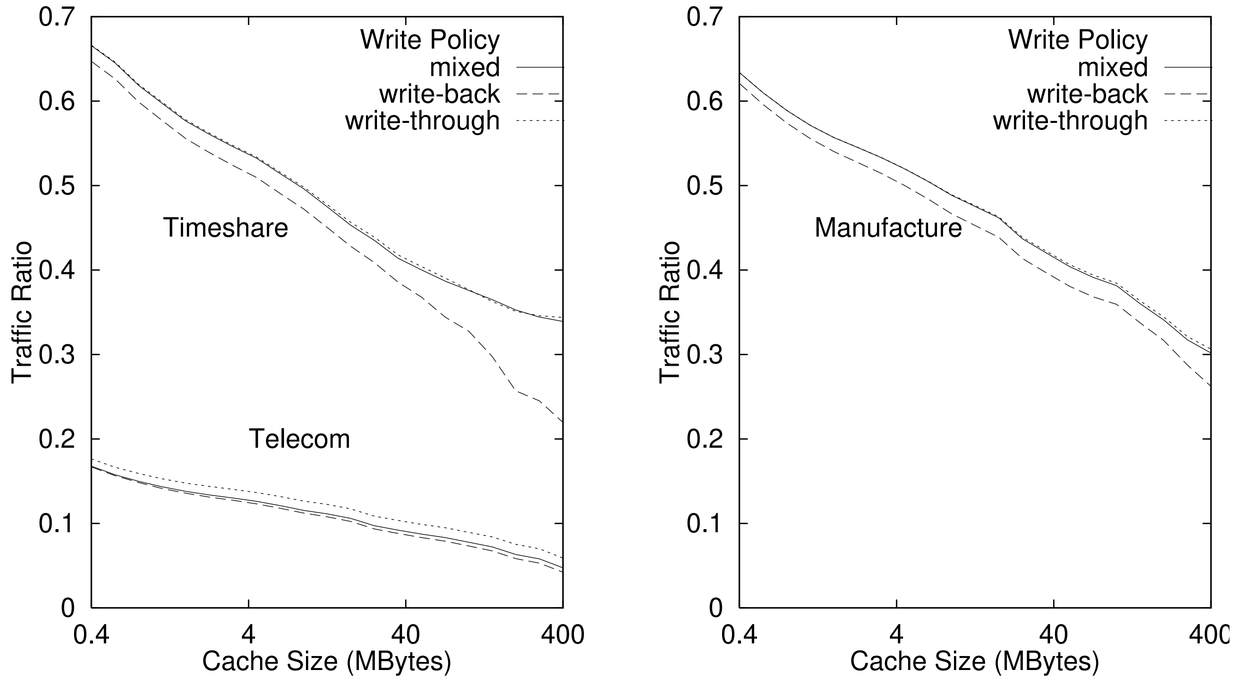


Figure A26: Fixed limiting cache miss ratios for various limits (limits expressed in terms of a percentage of cache capacity.)

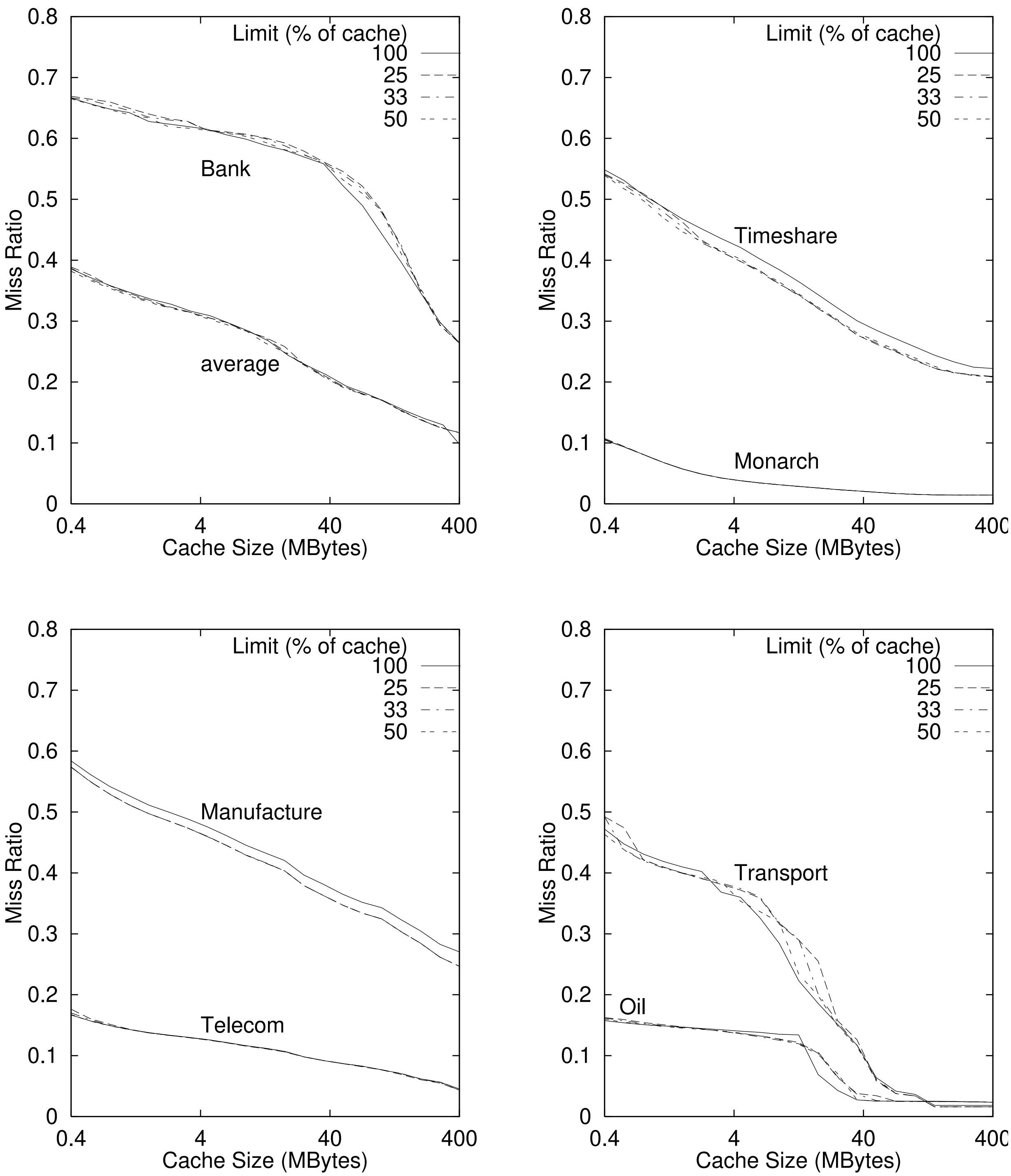
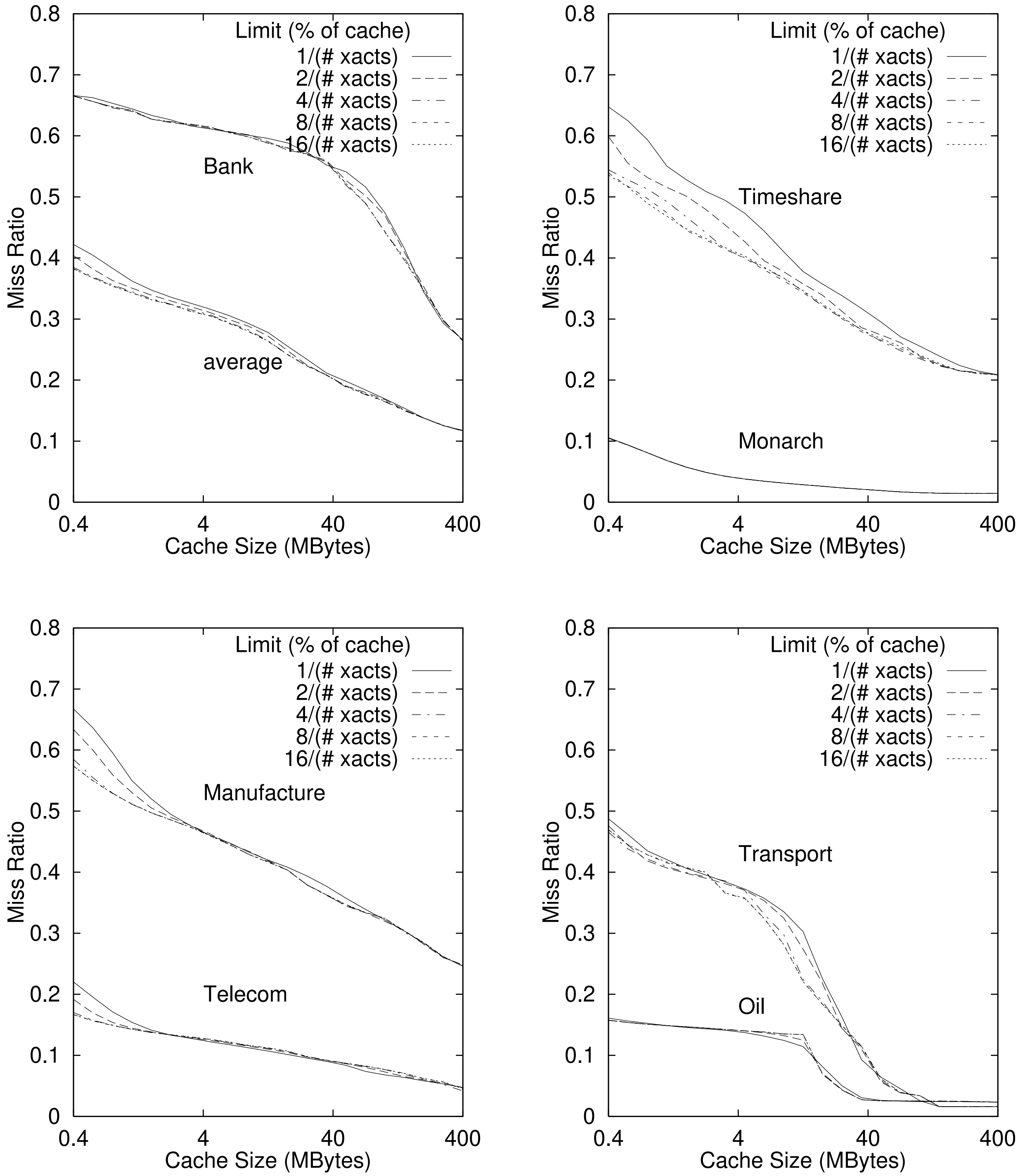


Figure A27: Dynamic limiting cache miss ratios for various limits.



References

[Alon90]: Rafael Alonso, Daniel Barbara' and Hector Garcia-Molina, "Data Caching Issues in an Information Retrieval System," *ACM Trans. on Database Systems 15, 3* , (Sept. 1990,) pp. 359-384.

[Arti95]: Dr. H. Pat Artis, "Record Level Caching: Theory and Practice," *CMG Transactions,* (Winter 1995,) pp. 71-76.

[Arun96]: Meenakshi Arunachalam, Alok Choudhary and Brad Rullman, "Implementation and Evaluation of Prefetching in the Intel Paragon Parallel File System," *Proc. 10th Intl. Parallel Processing Symposium,* (April 1996,) Honolulu, HI, pp. 554-559.

[Bake93]: Mary Baker and John Ousterhout, "Availability in the Sprite Distributed File System," *Operating Sys. Review, 27,5* (Dec. 1993,) pp. 95-98.

[Berg93]: Jeffrey A. Berger, "DFSMS Dynamic Cache Management Enhancement," *CMG Transactions* , (Spring 1993,) pp. 5-11.

[Bhid93]: Anupam K. Bhide, Asit Dan and Daniel M. Dias, "A Simple Analysis of the LRU Buffer Policy and Its Relationship to Buffer Warm-Up Transient," *Proc. 9th Intl. Conf. on Data Engineering* (April 1993,) Vienna, Austria, pp. 125-133.

[Bisw93]: Prabuddha Biswas, K. K. Ramakrishnan, Don Towsley, "Trace Driven Analysis of Write Caching Policies for Disks," *Proc. ACM SIGMETRICS Conf. on Measurement & Modeling of Computer Systems* , (May 1993,) Santa Clara, CA, pp. 13-23.

[Bolo93]: Jean-Chrysostome Bolot and Hossam Afifi, "Evaluating Caching Schemes for the X.500 Directory System," *Proc. 13th Intl. Conf. on Distributed Computing Systems,* (May 1993,) Pittsburgh, PA, pp.112-119.

[Bozi90]: G. Bozikian and J. W. Atwood, "CICS LSR Buffer Simulator (CLBS,)" *CMG Transactions* , (Spring 1990,) pp. 17-25.

[Bran88]: Alexandre Brandwajn, "Modeling DASDs and Disk Caches," *CMG Transactions* , (Fall 1988,) pp. 61-70.

[Brau89]: Andrew Braunstein, Mark Riley, and John Wilkes, "Improving the Efficiency of UNIX File Buffer Caches," Proc. 12th SOSP *, Operating Sys. Review, 23, 5* , (Dec. 1989,) pp. 71-82.

[Busc85]: John R. Busch and Alan J. Kondoff, "Disc Caching in the System Processing Units of the HP3000 Family of Computers," *Hewlett Packard Journal* , (Feb. 1985,) pp. 21-39.

[Buze83]: Jeffrey P. Buzen, "BEST/1 Analysis of the IBM 3880-13 Cached Storage Controller," *Proc. SHARE 61*, (Aug. 1983,) New York, NY, pp. 63-74.

[Cars92]: Scott D. Carson and Sanjeev Setia, "Analysis of the Periodic Update Write Policy For Disk Cache," *IEEE Trans. on Software Eng., 18, 1* , (Jan. 1992,) pp. 44-54.

[Casa89]: R. I. Casas and K. C. Sevcik, "A Buffer Management Model for Use in Predicting Overall Database System Performance," *5th Intern. Conf. on Data Engineering* , (Feb. 1989,) Los Angeles, CA. pp. 463-469.

[Ciga88]: John Francis Cigas, *The Design and Evaluation of a Block-level Disk Cache Using Pseudo-files* , (1988,) Ph.D. dissertation, University of California, San Diego.

[Dan90a]: Asit Dan, Daniel M. Dias and Philip S. Yu, "The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment," *Proc., 16th VLDB Conference* , (1990,) Brisbane, Australia, pp. 419-431.

[Dan90b]: Asit Dan, Daniel M. Dias and Philip S. Yu, "Database Buffer Model for the Data Sharing Environment'" *Proc. 6th Intl. Conf. on Data Engineering* , (Feb. 1990,) Los Angeles, CA, pp. 538-544.

[DeFa88]: Samuel DeFazio, *Predictive Database Buffer Management Strategies: An Empirical Approach* , (1988,) Ph.D. Thesis, University of Pittsburgh.

[Denn68]: P. J. Denning, "The Working Set Model for Program Behavior," *Commun. ACM, 11* , (May 1968,) pp. 323-333.

[Dixo84]: Jerry D. Dixon, Gerald A. Marazas, Andrew B. McNeill and Gerald U. Merckel, *Automatic Adjustment of Quantity of Prefetch Data in a Disk Cache Operation,* U.S. Patent 4,489,378.

[D'Aze95]: E. F. D'Azevedo and C. H. Romine, *EDONIO: Extended Distributed Object Network I/O Library* , Tech. Rept. ORNL/TM-12934, (March, 1995,) Oak Ridge Natl. Lab., Oak Ridge, TN.

[D'Gue92]: Darryl D'Guerra and Sandeep Deshmukh, "RAM Disk and Disk Cache: Observations on Two I/O Optimizing Solutions," *CMG Transactions* , (Winter 1992,) pp. 25-35.

[Effe84]: Wolfgang Effelsberg and Theo Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Management, 9, 4* , (Dec. 1984,) pp. 560-595.

[Falo91]: Christos Faloutsos, Raymond Ng and Timos Sellis, "Predictive Load Control for Flexible Buffer Allocation," *Proc. 17th Intern. Conf. on Very Large Data Bases* (Sept. 1991,) Barcelona, Spain, pp. 265-274.

[Frie83]: Mark B. Friedman, "DASD Access Patterns," *Proc. 1983 CMG Intern. Conf.* (Dec. 1983,) Washington, DC, pp. 51-61.

[Frie95a]: Mark B. Friedman, "The Feasibility of Using a General Purpose Unix Multicomputer to Emulate an IBM 3990 Cached Controller," *CMG Transactions* , (Winter 1995,) pp. 55-65.

[Frie95b]: Mark Friedman, "Evaluation of an Approximation Technique for Disk Cache Sizing," *CMG Proceedings, 21st Intl. Conf. for the Resource Management and Performance Evaluation of Enterprise Computing Systems,* (Dec. 1995,) Nashville, TN, pp. 994-1001.

[Fuld88]: Stephen Fuld, "The Amperif Cache Disk System," *Proc. IEEE Computer Society Intl. Spring Compcon Conf.* , (Feb.-Mar. 1988,) IEEE Computer Society, pp. 156-157.

[Grim93]: Knut Steiner Grimsrud, James K. Archibald and Brent E. Nelson, "Multiple Prefetch Adaptive Disk Caching," *IEEE Trans. on Knowledge and Data Engineering, 5, 1,* (Feb. 1993,) pp 88-103.

[Gros85]: C. P. Grossman, "Cache-DASD storage design for improving system performance," *IMB Sys. Jour.,* *24, 3/4*, (1985,) IBM Corp., Armonk, NY.

[Gros89]: C. P. Grossman, “Evolution of the DASD storage control,” *IMB Sys. Jour., 28, 2*, (1989,) IBM Corp., Armonk, NY.

[Hoff89]: J. C. Hoffecker, Nathan Walsh and Mike Cunard, “DASD Performance Improvements with Actuator Level Buffers,” *Proc. 1989 CMG Intern. Conf.* (Dec. 1989,) Reno, NV, pp. 424-434.

[Hosp92]:Andy Hospodor, “Hit Ratio of Caching Disk Buffers,” *Proc. Spring CompCon 92* (Feb. 1992,) San Francisco, CA, pp. 427-432.

[Hunt81]: D. W. Hunter, “DASD Arm Buffers,” *IBM Tech. Disc. Bull. 24, 4* (Sept. 1981,) p. 2035.

[IBM83a]: IBM, *IBM 3880 Storage Control Model 13 Introduction*, GA32-0062-0, (1983,) IBM Corp., Tuscon, AZ.

[IBM83b]: IBM, *IBM 3880 Storage Control Model 13 Description*, GA32-0067 (1983). IBM Corp., Tucson, AZ.

[IBM85a]: IBM, *IBM 3880 Storage Control Model 23 Introduction*, GA32-0082, (1985,) IBM Corp., Tuscon, AZ.

[IBM85b]: *IBM, IBM 3880 Storage Control Model 23 Description*, GA32-0083, (1985,) IBM Corp., Tuscon, AZ.

[Jali89]: Paul J. Jalics and David R. McIntyre, “Caching and Other Disk Access Avoidance Techniques on Personal Computers,” *Communications of the ACM 32, 2*, (Feb. 1989,) pp. 246-255.

[Jone93]: A. L. (Roy) Jones, “The Implications and Potentials of Advanced Disk-Array Technology,” *CMG Transactions*, (Winter 1993,) pp 59-63.

[Kame92]: Nabil Kamel and Roger King, “Intelligent Database Caching Through the User of Page-Answers and Page-Traces,” *ACM Trans. on Database Systems, 17,4* (December 1992,) pp. 601-646.

[Kare94]: Ramakrishna Karedla, J. Spencer Love, Bradley G. Wherry, “Caching Strategies to Improve Disk System Performance,” *IEEE Computer*, (March 1994,) pp. 38-46.

[Katz89]: Randy H. Katz, Garth A. Gibson and David A. Patterson, “Disk System Architectures for High Performance Computing,” *Proc. of the IEEE, 77, 12*, (Dec. 1989,) pp. 1842-1858.

[Kear89]: J. P. Kearns and S. DeFazio, “Diversity in Database Reference Behavior,” *Perf. Eval. Review, 17*, (May 1989,) pp. 11-19.

[Kimb96]: Tracy Kimbrel, Pei Cao, Edward W. Felten, Ann R. Karlin and Kai Li, “Integrated Parallel Prefetching and Caching,” *Performance Evaluation Review, 24, 1/Proc. 1996 SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems* (May 1996,) Philadelphia, PA, pp. 262-263.

[Koba84]: Makoto Kobayashi, “Dynamic Characteristics of Loops,” *IEEE Trans. on Comp., C33, 2,* (Feb. 1984,) pp 125-132.

[Koel89]: Charles Koelbel, Fady Lamaa, and Bharat Bhargava, “Efficient Implementation of Modularity in RAID,” *Proc. USENIX Workshop on Distributed and Multiprocessor Systems* (Oct. 1989,) Fort Lauderdale, FL, pp. 127-143.

[Kotz90]: David F. Kotz and Carla Schlatter Ellis, “Prefetching in File Systems for MIMD Multiprocessors,” *IEEE Trans. on Parallel and Distributed Sys. 1, 2* (Apr. 1990,) pp. 218-230.

[Kova86]: Roger P. Kovach, “DASD Cache Controllers: Performance Expectations and Measurement,” *Proc. CMG ’86* (Dec. 1986,) Las Vegas, NV, pp. 74-78.

[Kroe96]: Thomas M. Kroeger and Darrell D. E. Long, “Predicting File System Actions from Prior Events,” *Proc. 1996 Winter USENIX Tech. Conf* (Jan. 1996,) San Diego, CA, pp. 319-328.

[Lazo86]: Edward D. Lazowska, John Zahorjan, David R. Cheriton and Willy Zwaenepoel, “File Access Performance of Diskless Workstations,” *ACM TRans. on Computer Sys. 4,3* (Aug. 1986,) pp. 238-368.

[Lee88]: T. Paul Lee and Rebecca Wang, “A Performance Study on UNIX Disk I/O Reference Trace,” *Proc., CMG International Conf. on Management and Performance Evaluation of Computer Systems* (Dec. 1988,) Dallas, TX, pp. 121-127.

[Levy95]: Hanoch Levy and Robert J. T. Morris, “Exact Analysis of Bernoulli Superposition of Streams Into a Least Recently Used Cache,” *IEEE Trans. on Software Eng., 21, 8,* (Aug. 1995,) pp. 682-688.

[Maka90]: Dwight J. Makaroff and Dr. Derek L. Eager, “Disk Cache Performance for Distributed Systems,” *Proc. 10th Intern. Conf. on Distributed Computing Systems* (May 1990,) Paris, France, IEEE, pp. 212-219.

[Mank83]: P. S. Mankekar and C. A. Milligan, “Performance Prediction and Validation of Interacting Multiple Subsystems in Skew-Loaded Cached DASD,” *Proc. CMG XIV* (1983,) Washington, D.C., pp. 383-387.

[Mann93]: Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian and Garret Swart, “A Coherent Distributed File Cache With Directory Write-behind,” *DEC Systems Research Center Research Rep. 103*, (June 1993).

[Mara83]: G. A. Marazas, Alvin M. Blum and Edward A. MacNair, “A Research Queuing Package (RESQ) Model of A Transaction Processing System with DASD Cache,” *IBM Research Report RC 10123*, (Aug. 1983).

[Mars94]: B. Marsh, F. Douglis and P. Krishnan, “Flash Memory File Caching for Mobile Computers,” *Proc. 25th Hawaii Intl. Conf. on System Sciences Vol. I: Architecture*, Wailea, HI, (Jan. 1994,) pp. 451-460.

[McDo88]: Shane McDonald, “Dynamically Restructuring Disk Space for Improved File System Performance,” *Univ. of Saskatchewan Dept. of Computational Science Research Rept. 88-14,* (1988).

[McKu84]: Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry, “A Fast File System for UNIX,” *ACM Transactions on Computer Systems 2, 3* (Aug. 1984,) pp. 181-197.

[McKu90]: Marshall Kirk McKusick and Michael J. Karels, “A New Virtual Memory Implementation for Berkeley UNIX,” *EUUG Conference Proceedings* (Autumn 1986,) Manchester, U.K., pp. 451-458.

[McNu93]: Bruce McNutt, “A Simple Statistical Model of Cache Reference Locality, and its Application to Cache Planning, Measurement and Control,” *CMG Transactions* (Winter 1993,) pp. 13-21.

[Memo78]: Memorex, *3770 Disc Cache Product Description*

Manual, 3770/00-00 (1978,) Memorex Corp, Santa Clara, CA.

[Meno88]: Jai Menon and Mike Hartung, "The IBM 3990 Disk Cache," *Digest, Spring Compcon 88*, San Francisco, CA, (Feb.-Mar. 1988,) IEEE Computer Society, pp. 146-151.

[Mill82]: C. Milligan and P. Mankekar, "Analytic and Simulation Modelling of Cached Storage Controllers," *Tech. Rept. Storage Technology Corp.* (1982,) Louisville, CO.

[Mill95]: Douglas W. Miller and D. T. Harper III, "Performance Analysis of Disk Cache Write Policies," *Microprocessors and Microsystems, 19,3,* (Apr. 1995,) pp. 121-130.

[Minn93]: Ronald G. Minnich, "The AutoCacher: A File Cache Which Operates at the NFS Level," *Proc. 1993 Winter USENIX Tech. Conf.* (Jan. 1993,) San Diego, pp. 77-83.

[Moha91]: C. Mohan and Inderpal Narang, "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment," *Proc. 17th Intern. Conf. on Very Large Data Bases* (Sept. 1991,) Barcelona, Spain, pp. 193-207.

[Moha95]: C. Mohan, "Disk Read-Write Optimization and Data Integrity in Transaction Systems Using Write-Ahead Logging," *Proc. 11th Annual Conf. on Data Engineering* (March 1995,) Taipei, Taiwan, pp. 324-331.

[Mung86]: Anthony Mungal, "Cached I/O Subsystems: Analysis and Performance," *Proc. CMG'86* (Dec. 1986,) Las Vegas, NV, pp. 42-62.

[Munt92]: D. Muntz and P. Honeyman, "Multi-level Caching in Distributed File Systems—or—Your Cache ain't nuthin' but trash," *Proc. 1992 Winter USENIX Tech. Conf.* (Jan. 1992,) San Francisco, CA, pp. 305-313.

[Naga90]: Umpei Nagashima, Fumio Nishimoto, Takashi Shibata, Hiroshi Itoh, and Minoru Gotoh, "An Improvement of I/O Function for Auxiliary Storage: Parallel I/O for a Large Scale Supercomputing," *Proc. 1990 Intern. Conf. on Supercomputing* (June 1990,) Amsterdam, the Netherlands, pp. 48-59.

[Nels88]: Michael N. Nelson, Brent B. Welch and John K. Ousterhout, "Caching in the Sprite Network File System," *ACM Trans. on Computer Sys, 6, 1* (Feb. 1988,) pp.134-154.

[Ng91]: Spencer W. Ng, "Improving Disk Performance Via Latency Reduction," *IEEE Trans. on Computers 40, 1*, (Jan. 1991,) pp. 22-30.

[Ng96]: Raymond T. Ng and Jinhai Yang, "An Analysis of Buffer Sharing and Prefetching Techniques for Multimedia Systems," *ACM Multimedia Systems 4, 2,* (1996,) pp. 55-69.

[Ordi93]: Joann J. Ordille and Barton P. Miller, "Distributed Active Catalogs and Meta-Data Caching in Descriptive Name Services," *Proc. 13th Intl. Conf. on Distributed Computing Systems,* Pittsburgh, PA (May 1993), pp. 120-129.

[Orji92]: Cyril U. Orji and Jon A. Solworth, "Write-Only Disk Cache Experiments on Multiple Surface Disks," *Proc. ICCI'92 4th Intl. Conf. on Computers and Information* (May 1992,) Toronto, Ontario, Canada, pp. 395-388.

[Oust85]: John K. Ousterhout, Herve' Da Costa, David Harrison, John A. Kunze, Mike Kupfer and James G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proc. 10th Symp. on Operating Systems Principles* (Dec. 1985,) Orcas Island, WA, pp. 15-24.

[Palm91]: Mark Palmer and Stanley B. Zdonik, "Fido: A Cache That Learns to Fetch," *Proc. of the 17th Intern. Conf. on Very Large Databases* (Sept. 1991,) Barcelona, Spain, pp. 255-264.

[Patt87]: David A. Patterson, Garth Gibson and Randy Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID,)" *UC Berkeley Tech. Rept. UCB/CSD 87/391*, (Dec. 1987,) Univ. of Calif. at Berkeley, Berkeley, CA.

[Patt93]: R Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan, "A Status Report on Research in Transparent Informed Prefetching," *Operating Systems Review, 27, 2,* (April 1993,) pp. 21-34.

[Patt95]: R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky and Jim Zalenka, *Informed Prefetching and Caching*, Carnegie Mellon University Tech. Rept. CMU-CS-95-134 (May 1195,) Carnegie Mellon University, Pittsburgh, PA.

[Phal95]: Vidyadhar Phalke and B. Gopinath, "Program Modelling via Inter-Reference Gaps and Applications," *Proc. 3rd Intl. Conf. on Modeling, Analysis and Simulation of Computers and Telecommunications Systems*, Durham, NC, (Jan. 1995,) pp. 212-216.

[Post88]: Alan Poston, "A High Performance File System for UNIX," *Proc. Workshop on UNIX and Supercomputers* (Sept. 1988,) Pittsburgh, PA, pp. 215-226.

[Prie76]: B. G. Prieve and R. S. Fabry, "VMIN—An Optimal Variable Space Page Replacement Algorithm," *Commun. ACM, 19*, (May 1976,) pp. 295-297.

[Pura96]: Apratim Purakayastha, Carla Schlatter Ellis and David Kotz, "ENWRICH: A Compute-Processor Write Caching Scheme for Parallel File Systems," *Proc. 4th Annual Workshop on I/O in Parallel and Distributed Systems* (May 1996,) pp. 55-68.

[Rich93]: Kathy J. Richardson and Michael J. Flynn, "Strategies to Improve I/O Cache Performance," *Proc. 26th Annual Hawaii Intern. Conf. on System Sciences* (1993,) Hawaii, pp. 31-39.

[Robi90]: John T. Robinson and Murthy B. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. ACM SIGMETRICS Conf. on Measurement & Modeling of Computer Systems* (May 1990,) Boulder, CO, pp. 134-142.

[Sche73]: Lee J. Scheffler, "Optimal Folding of a Paging Drum in a Three Level Memory System," *Proc. 4th SIGOPS Conf. Yorktown Heights, NY, Operating Systems Review 7 (4,)* (1973,) pp. 58-65.

[Schr85]: Michael D. Schroeder, David K. Gifford and Roger M. Needham, "A Caching File System for a Programmer's Workstation," *Oper. Sys. Review, 19, 5* (Dec. 1985,) pp. 25-34.

[Selt90]: Margo Seltzer, Peter Chen and John Ousterhout, "Disk Scheduling Revisited," *Proc. 1990 Winter USENIX Tech. Conf.* (Jan. 1990,) Washington, D.C., pp. 313-324.

[Shih90]: F. Warren Shih, Tze-Ching Lee, Shauchi Ong, "A File-Based Adaptive Prefetch Caching Design," *Proc. 1990 IEEE Intl. Conf. on Computer Design: VLSI in Computers and Processors* (Sept. 1990,) Cambridge, MA, pp. 463-466.

[Smit76]: A. J. Smith, "A Modified Working Set Paging Algorithm," *IEEE Trans. on Comp., C 25, 9*, (Sept. 1976,) pp. 907-914.

[Smit78]: A. J. Smith, "Sequentiality and Prefetching in Database Systems," *ACM Transactions on Database Systems 3, 3*, (Sept. 1978,) pp. 223-247.

[Smit81]: Alan Jay Smith, "Input/Output Optimization and Disk Architectures: A Survey," 1, 2, (1981,) pp. 104-117.

[Smit85]: A. J. Smith, "Disk Cache—Miss Ratio Analysis and Design Considerations," *ACM Transactions on Computer Systems 3, 3*, (Aug. 1985,) pp. 161-203.

[KSmit90]: Kevin F. Smith, "Caching Techniques in Distributed File Systems: A Survey," *CMG Transactions* (Winter 1990,) pp. 93-98.

[Solo96]: Valery Soloviev, "Prefetching in Segmented Disk Cache for Multi-Disk Systems," *Proc. 4th Ann. Workshop on I/O in Parallel and Distributed Systems* (May 1996,) Philadelphia, PA, pp. 69-82.

[Solw90]: Jon A. Solworth and Cyril U. Orji, "Write-Only Disk Caches," *Proc. Intl. Conf. of ACM SIGMOD* (May 1990,) Atlantic City, NJ, pp. 123-132.

[Sper81]: Sperry Univac, "Cache/Disk System," *Sperry Univac Product Announcement for 5057 Cache Disk Processor and 7053 Storage Unit*, Sperry Univac, (1981).

[STC82]: Storage Technology Corp., *Sybercache 8890 Intelligent Disk Controller*, Storage Technology Corp., Louisville, Colo. (1982).

[Ston92]: H. S. Stone, John Turek and J. L. Wolf, "Optimal partitioning of Cache Memory," *IEEE Trans. on Computers, 41, 9,* (Sept. 1992,) pp. 1054-1068.

[Thek92]: Chandramohan A. Thekkath, John Wilkes and Edward D. Lazowska, *Techniques for File System Simulation*, Hewlett Packard Laboratories Tech. Rept. HPL-92-131 (Sept. 1992,) Hewlett Packard Corp.

[Thie92]: Dominique Thiebaut, Harold S. Stone and Joel L. Wolf, "Improving Disk Cache Hit-Ratios Through Cache Partitioning," *IEEE Trans. on Computers, 41, 6*, (June 1992,) pp. 665-676.

[Thom87]: James Gordon Thompson, *Efficient Analysis of Caching Systems*, UC Berkeley Tech. Rept. UCB/CSD 87/374, (Oct. 1987,) Univ. of Calif. at Berkeley, Berkeley, CA.

[Toku80]: T. Tokunaga, Y. Hirai and S. Yamamoto, "Integrated Disk Cache System with File Adaptive Control," *Proc. IEEE Computer Society Conf. '80* (Sept. 1980,) Washington, D.C., pp. 412-416.

[Trei95]: Kent Treiber and Jai Menon, "Simulation Study of Cached RAID5 Designs," *Proc. 1st IEEE Symp. on High-Performance Computer Architecture* (Jan. 1995,) Raleigh, NC, pp. 186-197.

[Van88]: A. J. Van de Goor and A. Moolenaar, "UNIX I/O in a Multiprocessor System," *Proc. 1988 Winter USENIX Tech. Conf.* (Feb. 1988,) Dallas, TX, pp. 251-257.

[Varm95]: Anujan Varma and Quinn Jacobson, "Destage Algorithms for Disk Arrays with Non-Volatile Caches," *Proc. 22nd Intl. Symp. on Computer Architecture,* (June 1995,) Santa Margherita Ligure, Italy, pp. 83-95.

[Viav89]: Steven Viavant, *Collection, Reduction, and Analysis of DB2 Trace Data*, Master's Report, University of California, Berkeley/Computer Science Department, (July 1989,) Univ. of Calif. at Berkeley, Berkeley, CA.

[Welc91]: Brent B. Welch, "Measured Performance of Caching in the Sprite Network File System," *Computing Systems 3,4*, (Summer 1991,) pp. 315-341.

[Will93]: D. L. Willick, D. L. Eager and R. B. Bunt, "Disk Cache Replacement Policies for Network Fileservers," *Proc. 13th Intl. Conf. on Distributed Computing Sys.,* (May 1993,) Pittsburgh, PA, pp. 2-11.

[Wu94]: Yuguang Wu, "Evaluation of Write-back Caches for Multiple Block-Sizes," *Proc. 2nd Intl. Workshop on Modeling, Analysis and Simulation of Computers and Telecommunications Systems,* (Jan.-Feb. 1994,) Durham, NC, pp. 57-61.

[Yim95]: Emily Yim, "Fortran I/O Optimization," *NERSC Buffer, 19, 12,* (Dec. 1995,) Natl. Energy Research Supercomputer Center, Livermore, CA. pp. 14-23.