I/O CHARACTERIZATION AND ATTRIBUTE CACHES FOR IMPROVED I/O SYSTEM PERFORMANCE

Kathy J. Richardson

Technical Report No. CSL-TR-94-655

Dec 1994

Supported by NASA under NAG2-248 and Digital Western Research Laboratory.

I/O CHARACTERIZATION AND ATTRIBUTE CACHES FOR IMPROVED I/O SYSTEM PERFORMANCE

by

Kathy J. Richardson

Technical Report No. CSL-TR-94-655

Dec 1994

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

Abstract

Workloads generate a variety of disk I/O requests to access file information, execute programs, and perform computation. I/O caches capture most of these requests, reducing execution time, providing high I/O rates, and decreasing the disk bandwidth needed by each workload. A cache has difficulty capturing the full range of I/O behavior, however, when it treats the requests as single stream of uniform tasks. The single stream contains I/O requests for data with vastly different reuse rates and access patterns.

Disk files can be classified as accesses to inodes, directories, datafiles or executables. The combined cache behavior of all four taken together provides few clues for improving performance of the I/O cache. But individually, the cache behavior of each reveals the distinct components that make up aggregate I/O behavior. Inodes and directories turn out to be small, highly reused files. Datafiles and executable files have more diverse characteristics. The smaller ones exhibit moderate reuse and have little sequential access, while the larger files tend to be accessed sequentially and not reused. Properly used, file type and file size information improves cache performance.

The dissertation introduces *attribute caches* to improve I/O cache performance. Attribute caches use file attributes to selectively cache I/O data with a cache scheme tailored to the expected behavior of the file type. Inodes and directories are cached in very small blocks, capitalizing on their high reuse rate, and small space requirements. Large files are cached in large cache blocks capitalizing on their sequential access patterns. Small and medium sized files are cached in average 4 kbyte blocks that minimizes the memory required to service the bulk of requests. The portion of cache dedicated to each group varies with total cache size. This allows the important features of the workload to be captured at the appropriate cache size, and increases the total cache utilization. For a set of 11 measured workloads an attribute cache scheme reduced the miss ratio 25–60% depending on cache size, and required only about 1/8 as much memory as a typical I/O cache implementation achieving the same miss ratio.

Key Words and Phrases: I/O cache, I/O architecture, disk cache

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Input and output, known collectively as I/O, is essential for useful computation. Without input from the outside world, there would be no way to direct computation, or to provide new data for computations. And without output, there would be no way to obtain results of computations.

The majority of I/O is not new data entering or leaving a computer system, but rather existing data being reused or saved for later use or evaluation. Most computer systems use magnetic disk drives to store the bulk of this data. Input occurs when the computer brings data in from a disk, and output occurs when it writes data out to a disk.

Disks store a wide range of information including computation results, application programs, intermediate results, data files and databases, and the operating system. Some of the information stored on disks results from the outside world giving input to the computer system, but computation generates much of the stored information. Regardless, disks store data for future use. This information requires significant storage space. Many of today's workstations or personal computers have 1 to 2 gigabytes of disk storage.

Magnetic disks have several properties that make them particularly attractive for storing all this information. They have a huge capacity, are significantly cheaper than higher performance alternatives and provide permanent storage.

Disks also have a major disadvantage. Access to disk information is slow when compared with the rest of the computer system. Disks are much slower than DRAM memory in both access time and in transfer rate. Transferring data to and from disk is slow in comparison to the peak rate that data can be consumed or generated by current processors. It is also slow in comparison to the peak rate that data can be transferred to and from memory.

Disk I/O is a result of economics and technology. All disk I/O could disappear, if vast amounts of infinitely fast, reliable, and cheap memory were available. Even small amounts of fast memory are costly, and it tends to become slower as the amount increases. In addition, memory in its traditional DRAM form is unreliable because it is volatile; its contents are lost if the power or any other part of the computer system fails. Disks provide massive amounts of data storage. They only require power while reading or writing data. If there is a power failure or a failure in some other part of the system, the data on the disk remains intact.

Magnetic disks have been used extensively for the past 30 years, and they are unlikely to be eliminated in the near future for two reasons: (1) The cost per storage byte on disk is maybe, 1/50th, that of other directly addressable storage, and storage requirements keep growing. (2) The volume of information continues to increase rapidly, both in the number of different pieces of information and the size of the individual files. Large files keep getting bigger; the large data files of ten years ago would all fit into today's memory, but today's large files are ten times larger [BHK+91].

All of this means that disks are likely to remain the technology for storing information for the immediate future. Disk I/O speeds have seen only moderate increases while CPU speeds have increased rapidly. Insufficient I/O system performance can considerably reduce the computation that a processor can perform. Sufficient I/O performance merely allows full utilization of the CPU. Minimizing the negative impact of I/O on total system performance requires aggressive techniques to reduce the amount of actual disk usage and effectively hide disk latency from the user.

Two key parameters determine the impact of I/O on total system performance. One is the amount of time spent waiting for each I/O to complete, and the other is the number of disk I/O operations performed. The amount of disk I/O required for a particular workload is affected by the amount of intermediate storage resources available, and the amount of information available in those resources from previously completed work.

Disk I/O cache systems use intermediate storage to eliminate disk accesses by servicing data requests much more quickly than a disk, and by capturing the locality inherent in the workload. An I/O cache stores the most recently referenced disk data in memory. When that data is re-referenced, the I/O cache supplies it in a fraction of the time that the disk would require. The I/O cache eliminates disk accesses by servicing requests for disk data. I/O caches provide higher I/O rates than disks because they consist of memory that has shorter access times and greater data transfer rates than disks. This work focuses on understanding the behavior of file requests and using that knowledge to improve the overall effectiveness of I/O caches.

I/O requests exhibit temporal locality and spatial locality. *Temporal locality* happens when a workload re-references the same data within a short time. *Spatial locality* occurs when the data request follows a request for adjacent data. Locality allows I/O caches to significantly reduce disk accesses. Servicing many requests from the faster I/O cache reduces the average I/O request access time. Improving the I/O cache function is thus crucial to improving overall system performance.

The work presented here shows how to reduce the read requests not captured by the I/O cache which must be serviced by the disk. Capturing more read requests with the I/O cache reduces the average read service time seen by the applications. The average read service time is determined by the disk service time and the I/O cache service time. Since the disk service time is much greater than the I/O cache service time, reducing the number of disk accesses reduces the average read service time.

1.1 I/O System Architecture

Figure 1.1 shows a conceptual block diagram of the CPU, memory system and the I/O system. An I/O cache logically belongs to the I/O system, but might be implemented elsewhere. A workstation usually implements the I/O cache as part of the main memory.

Read and write requests made to the I/O cache depend on the workload of the machine. Figure 1.1 represents these requests as R1 and W1. Requests the I/O cache cannot service result in disk requests, shown as R2 and W2. The relationship between R1 and R2 and between W1 and W2 depends on the size of the I/O cache and of the I/O cache management policy and the write policy. The interaction between the I/O cache and the disk system determines the final I/O system performance for a given workload.

1.1.1 Disks

A disk is a stack of platters that rotate on a spindle. The platters are coated with a magnetic material onto which data is recorded. A magnetic head reads or writes data on each platter. The platters have a large surface area for storing data, divided into discrete blocks called *sectors*. To read or write a particular sector, the arm must move the read/write head to the appropriate *track* and then wait until the sector rotates under the head before accessing the data. Figure 1.2 shows the top view of a single platter.

The disk organization and construction provide a large volume of relatively inexpensive storage. The diameter of each platter is between 2" and 14", and there may be up to 20 platters on a spindle. The head records data at about 10^9 bits per square inch, now 1995. Increases in technology can rapidly improve the recording density [HP90]. Having a single head to access a whole platter provides an inexpensive access mechanism for a large volume of data.

The disk organization and construction also determine the data access properties and system utilization. Accessing data is slow. To read or write a particular sector the arm must *seek* to the appropriate track. The seek time includes time to move the head and for the mechanical arm to settle. The seek

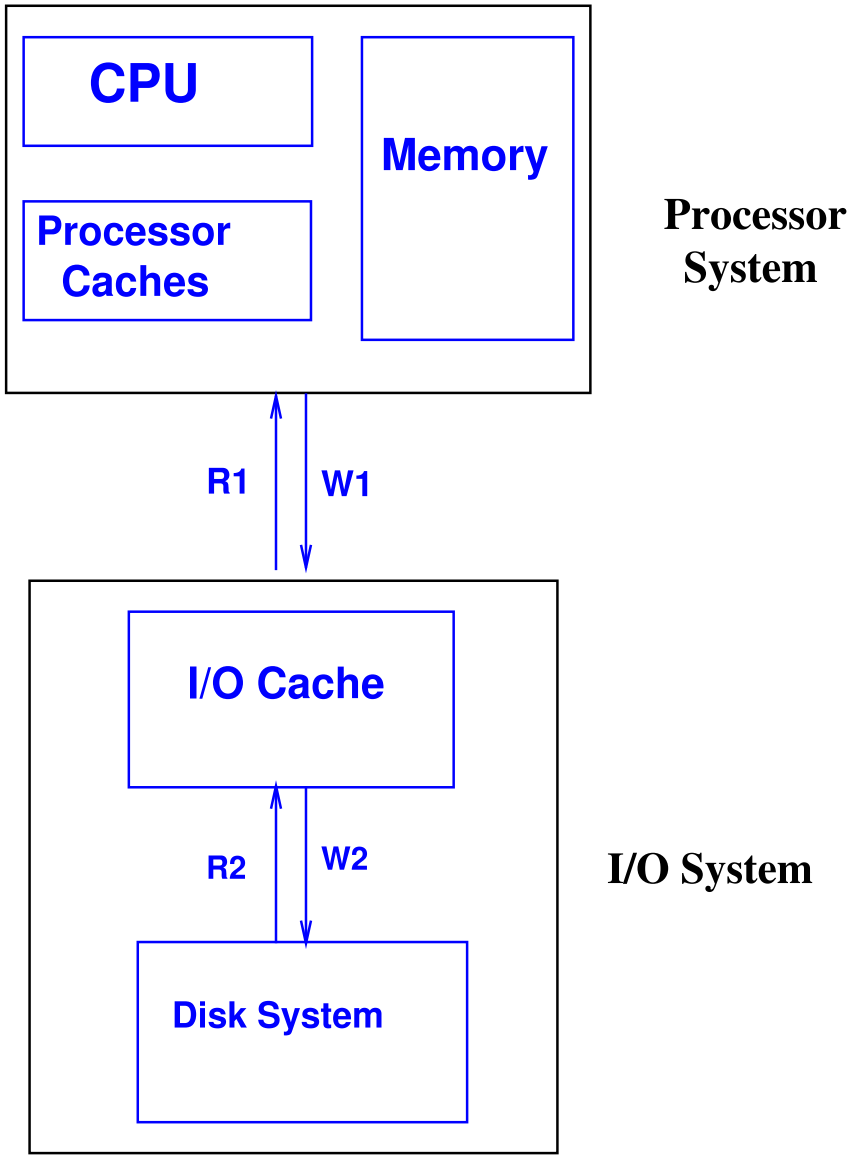


Figure 1.1: Computer system diagram showing the relation between the CPU, memory and the I/O cache.

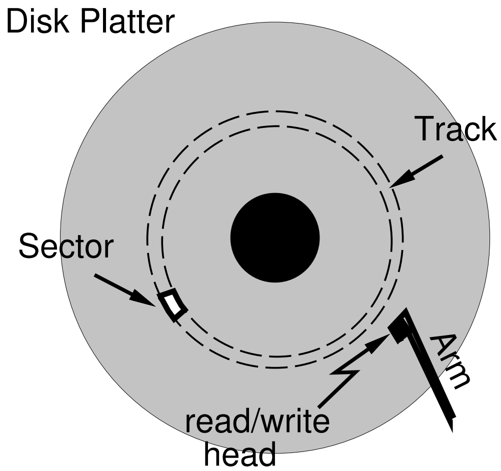


Figure 1.2: Overhead View of a Disk Platter and Key Access Features.

time relates to the distance the arm moves. The seek time is expressed as

$$\text{Seek Time} = a + b\sqrt{\Delta} \tag{1.1}$$

where a and b are mechanical constants: a is the mechanical settling time, and b is related to the track width [STH83]. Δ is the traversal displacement in measured in tracks. A typical seek requires between 4 ms and 25 ms depending on the distance traversed. An average seek takes about 12 ms, but only about a third of all disk accesses require seeks [Kim86, Lyn72]. Once the seek completes, the head waits for the sector to rotate under the head; this is the *rotational latency*. The latency, on average requires half a disk rotation. Rotational speeds range from 3600 to 7400 RPM, requiring 8.1 to 16.7 ms per rotation. Half a rotation requires 4.1–8.3 ms. Once the head is correctly positioned, it can read or write the data. The time for this depends on the amount of data and the disk transfer rate.

The total time to access disk data also includes waiting for prior requests to complete, and time to transfer the request to the disk. Figure 1.3 depicts the time components that contribute to the total disk access time. In a reasonably tuned system, the wait queue time should always be less than one request, but it must still be considered [Kim86, KT91]. Transferring the request and data can involve additional arbitration that adds further to the total time [RW94b]. The waiting time, request transfer time, seek time and rotational latency occur on a disk access regardless of the transfer size. The data transfer time is the only part specifically related to the size of the request.

The average access time for requests can be approximated as

$$\text{Disk Access Time} = T_{overhead} + R_{xfer} * n \tag{1.2}$$

where $T_{overhead}$ is the average overhead for a disk request, R_{xfer} is the disk transfer rate and n is the number of bytes in the request. This equation can be used to estimate the total access time of

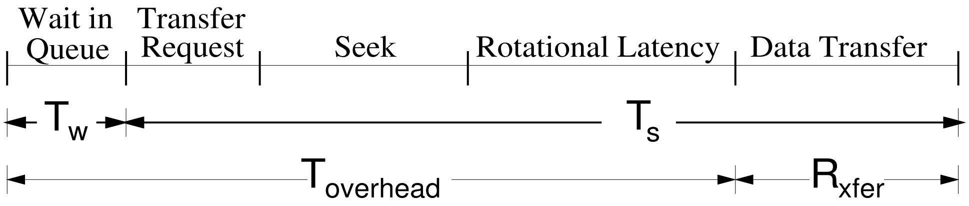


Figure 1.3: Latency components of a disk request.

many requests, but is not accurate for an individual request. The $T_{overhead}$ is calculated from disk parameters and measured behavior.

Disks provide non-volatile storage. Once data is stored on disk the power can be removed without destroying any data. However, the data is only retrievable if information about the data is also stored on disk. Operating systems build file systems on the disk, including data placement information that organizes the data and makes the disk self-contained. The information about data placement, called meta-data, describes which sectors are associated with which files and allows users and applications to organize data without any knowledge of the disk layout.

Many techniques are being applied to reduce the service time of disks (T_S) which then also reduces the wait time (T_W) [Fly94]. Traditional technology improvements decrease the service time. Increases in recording density increase the data transfer rate, which decreases the data transfer time. Increased densities can shrink the disk size or increase capacity. If the disk size shrinks, the track span distance shrinks, reducing the average seek time. Smaller disks also enable higher rotational speeds, which are limited by centrifugal forces.

Operating disks in parallel can reduce rotational latency, seek time, data transfer time and/or the queue wait time, providing more disk bandwidth. Multiple arms on a disk reduce the rotational latency and seek time. The closet arm processes a data request. With two arms the average rotational latency becomes one-fourth of a revolution instead of a half revolution [Ng91, MK89].

Alternatively, data can be synchronously interleaved across several disks, reducing the data transfer time [Kim86, GMS88]. All disks operate in unison; a single seek accesses data on all disks, effectively increasing the transfer bandwidth by the number of disks. Data can also be interleaved asynchronously by sectors or blocks [RB89a, KT91, KT87]. This reduces the transfer time for multi-block requests without requiring all the disks to operate in unison. Both striping techniques reduce access time and balance the load by spreading heavily used data across several disks [RB89a, GWHP93].

Striping data across multiple disks reduces the overall reliability, since a single failure effects all data files. Redundancy combined with striping yields high performance and very reliable disk systems, frequently referred to as *Redundant Arrays of Inexpensive Disks* (RAID) [PGK88, CGK$^+$88, KOP$^+$89] . Currently, much of the disk system research is involved with understanding the performance and reliability issues of RAID systems [GHK$^+$89, Ols89, GWHP94, Che89, Ng88, MSG89, RB89b, RB89a].

Although these techniques improve disk access time, disks are and will continue to be considerably slower than the rest of the computer system. Increasing I/O parallelism and the speed of individual disk components does not change the nature of disk accesses. Accessing data requires significant overhead to position the read/write head over the data, and the transfer time corresponds to the volume of data being accessed.

1.1.2 I/O Caches

An I/O cache, also known as a disk cache, file buffer or buffer cache, is a memory buffer that stores recently used portions of the disk address space [Red92b]. The operating system checks the cache for each disk request to see if the disk block currently resides in the cache. If it finds the requested data, the cache satisfies the request and no physical I/O operation is performed. The cache thus reduces the average I/O service time. It also speeds up the remaining I/O operations by reducing contention at the disk. The first I/O cache was reported in 1968 [SV68]. For a complete bibliography on I/O caches see Smith's bibliography [Smi81].

The I/O cache attempts to support the capacity of disk data at the speed of the memory data. An I/O cache impacts the system performance in two ways [Smi85]. First, it reduces the average I/O access time by servicing hits quickly, and second, it reduces the amount of time a workload spends using the disk. An I/O cache lookup requires at most a few thousand instructions (less than 0.05 ms) [BRW89], compared with a disk access requiring 10–20 ms. Reducing the disk usage allows the I/O system to support faster processors, more processors, or larger workloads.

An I/O cache manages data blocks. The block size is typically larger than a disk sector, often the same size as a track for caches closely associated with the disk, and often equal to the memory system page size for main memory caches. Figure 1.4 depicts a typical I/O cache configuration. Most I/O caches are fully associative and use a *least recently used* (LRU) replacement policy. The cache looks to see if the current request resides in the cache. On a match, the cache *hit* data becomes the *most recently used* (MRU) entry. All the intervening entries move closer to the LRU entry. When no match occurs (a *miss*), the least recently used entry leaves removed or *evicted* from the cache and all the entries move down; the new data enters the cache as the MRU entry. The cache size and the block size determine the number of entries. The cache lookup, implemented in software, uses hash tables and multiply linked lists to reduce lookup time. The data transfer size and the block size are usually identical.

I/O caches logically reside between the disk and memory (see Figure 1.1). They can be physically located at the disk (commonly called disk caches), at intermediate I/O controllers, or in main

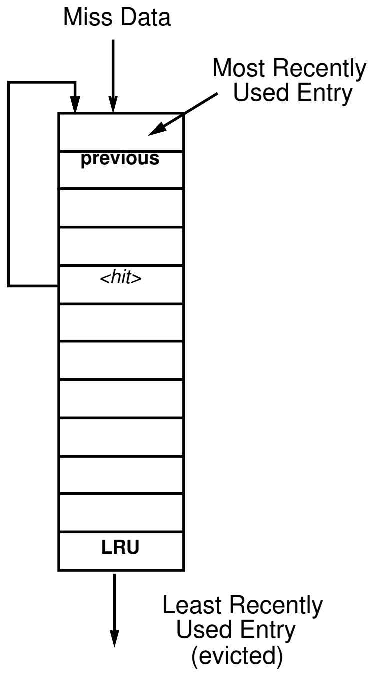


Figure 1.4: I/O cache organization and LRU replacement policy.

memory (called a buffer cache in Unix). They can consist of either volatile or non-volatile memory. The type of memory influences the choice of write policy.

True disk caches are implemented in hardware and are accessed as if they were a disk. In the pure sense, a disk cache acts like a disk. It has the same level of reliability, and the same access methods. IBM introduced their first disk cache as part of the IBM 3880 Storage Control Unit in 1979 [Gro89]. Today's external caches support a multitude of features, including mechanisms for sophisticated prefetch and selective removal of data [Men87]. Many disk drives available today have a small internal disk cache. These caches speed up the reported average access time for a disk request, but depending on the other caches in the system they may have little effect on the actual service time in a real environment. Large main memory I/O caches capture the bulk of the I/O requests so there may not be sufficient locality in the resulting request stream for the small internal caches to provide much benefit.

All modern operating systems utilize I/O caches to reduce disk accesses. For example, Unix has several types of I/O caches: one is called and inode buffer, which maintains information about file, and the other is called a buffer cache, which actually caches disk blocks [Bac86, MJLF84]. T Typical caches store information in main memory. Main memory is a working space for programs to hold and manipulate data. The data resides in memory only while being used, so it need not be protected from long term failures. DRAM is not designed to store data across systems failure; after a system failure all data stored in main memory is reloaded or regenerated. Using main memory as an I/O cache reduces the security of newly written data. Non-volatile memory can increase data security and eliminate writing data through to the disk or flushing periodically.

I/O caches are particularly important in distributed systems, where disk data often resides on another computer's disk. In this case, a disk lookup involves setting up the disk request, sending it to another machine, and interrupting that machine. The remote machine then sets up a physical disk request on its local disk, and sends the data back to the requesting machine. Most networked machines now have I/O caches. One major difference between network operating systems is the mechanism for sharing file write information. Some periodically flush information and data back to the remote server. A few operating systems maintain full knowledge of writes by having all file lookups performed on the file servers [NWO87].

1.2 Improving I/O Cache Performance

The performance of an I/O cache depends on its ability to capture workload locality. An effective cache closely matches workload needs with cache size, configuration and data management policies. Although the basic techniques for selecting the proper I/O cache size and configuration are known, designers rarely use them to select or tune I/O caches. The task of tuning an I/O cache is daunting, especially given that the cache must function under a wide range of workload environments and system memory configurations. I/O cache performance evaluation requires collecting I/O traces or installing instrumentation. Designers need traces to simulate and directly compare various cache alternatives and understand workload locality. Instrumentation can help measure and tune existing cache algorithms. Tracing or instrumenting the I/O stream activity requires monitoring I/O from within the operating system or using hardware to monitor physical I/O requests. This task is not simple, and the type of tracing or instrumentation determines which I/O characteristics are visible [Smi85]. Some I/O cache studies have used disk requests to study cache performance [Smi85, Red92a, Red92b], while others have used operating system traces of file system activity [ODCH$^+$85, NWO87, BHK$^+$91, RF93, LZCZ86].

A study by A.J. Smith evaluates much of the basic I/O cache behavior [Smi85]. The study characterizes disk cache behavior using disk traces that include information about the request type (system, paging, or other), and information about the job type (system, batch, interactive). This work evaluates cache location, block size, and the locality of various disk partitions. The results show that caches are most effective when located in main memory; they provide the best hit ratio for the least cost, and incur less overhead because there is no penalty for actually setting up the disk request. The effective block size depends on the cache size. In small caches, small blocks are more effective, while large caches perform better with large blocks. The study proposes prefetching as a solution for handling the desire for larger blocks in large caches. The various workload components have different cache properties and working set sizes. Although some classes of disk requests clearly showed poor cache behavior, none of the selective cache schemes could be consistently applied across workloads.

Several related studies characterize the I/O workload and show that I/O caches reduce I/O traffic in a distributed file system and provides reasonable system performance [ODCH+85, BAD+92, NWO87]. The studies record the file operations *open*, *close* and *reposition* in UNIX 4.2 BSD, and later in the SPRITE operating system, to produce a trace of the file regions accessed. The studies did not capture the sizes of individual requests. The results showed that the majority of file accesses are to small files of less that 10 Kbytes, that files tend to be accessed sequentially, and that the majority of bytes transferred reside in large files. Over time the size of large files has increased, and the length of sequential runs has increased. Files and sequential runs of larger than one megabyte are common. This leads to the suggestion of allocating only small files in the cache and devising some alternative mechanism for large sequentially-read files. One key feature of the SPRITE operating system is its file I/O cache [NWO87, Nel88, Nel90]. The I/O cache size varies with system activity, competing with the virtual memory system for physical memory resources. When program physical memory needs are small, the I/O cache size grows until it occupies the majority of the main memory. This variable I/O cache size has since been incorporated in several commercial operating systems.

I/O benchmarks typically stress various aspects of the I/O system to determine maximum throughput rate on live systems. Transaction processing benchmarks such as TCP-A and TCP-B and the Wisconsin Benchmark are used to rate the performance of both database systems and the underlying hardware [Gra91]. IOBENCH lets experimenters model both the CPU and I/O load for a database [WO89]. Insufficient characterization of general I/O environments limits the use of benchmarks for I/O cache evaluation. One adaptive technique uses self-scaling benchmarks to stress the I/O system, evaluating cache size, cache bandwidth, and disk bandwidth [Che92, CP92]. Such scaling benchmarks measure and test the limits and optimal operating points for existing systems. Without realistic reuse models or file access distributions they cannot guide I/O system design.

Main memory caches capture only a limited amount of write data locality. If the cache stores the only data copy, the data is vulnerable to loss. Writing all new data directly to disk protects the data but results in high write disk traffic. Capturing write data locality reduces the write traffic considerably. Data writes exhibit more sequential locality than do reads, and newly written data is often deleted or overwritten quickly [Red92a, RF93]. Caching writes for even a short period of time captures the sequential locality and eliminates writes for quickly-deleted or overwritten data.

Three common write policies include: (1) *Copy-back,* where writes may be copied back to disk only when they are evicted from the cache; (2) *write-through,* where writes may be written through to disk immediately, in which case they may not even be allocated in the cache; or (3) *periodic write,* where writes may be periodically written to disk. Periodic writes limit the time during which the disk and cache have different copies of data. Misses on data writes vary in importance depending on the allocation and management policy and size of the cache.

Non-volatile caches provide reliable data storage, which allows newly written data to reside in the cache for extended periods of time. Non-volatile caches allow write locality to be captured. Reddy [Red92a] characterizes the behavior of write data in non-volatile caches. Writes have lower miss ratios than reads, signifying better spatial locality. Using non-volatile caches, the read to write ratio of disk requests remains about constant across cache size, whereas with volatile caches, the write component becomes increasingly dominant with larger cache sizes. Two non-volatile cache organizations were evaluated by Baker et al. showing the feasibility of adding non-volatile RAM beside a volatile cache [BAD$^+$92]. One megabyte of non-volatile cache reduces the number of bytes written to disk by 40–50%. Non-volatile caches are becoming more common, and as the price of non-volatile RAM drops they will become more commonplace.

The cache configuration can help capture references to sequentially accessed files. Separating the fetch block size from the cache block size allows the cache to retrieve large blocks on a miss, while internally managing much smaller blocks [Red92b]. The small cache block size increases the number of independent small files that can reside in the cache. The larger fetch size captures the sequential locality in the I/O request stream. The additional data brought into the cache by the larger fetch soon gets evicted when it remains unreferenced.

The cache organization can reduce harmful effects of infrequently used data on the cache performance. Infrequently reused data occupies cache space and reduces the space available for high reuse data. The *segmented LRU* cache is designed to protect actively referenced data from being evicted by infrequently referenced data [KLW94]. This scheme segments the LRU list into two parts. The head of the LRU list (most recently used section) is reserved for actively used data. Only data that has been reused at least once may reside in the head segment. The tail segment of the LRU list holds less recently referenced data and new misses. On a miss, data is allocated to the tail section, and a re-reference moved it to the most recently used slot in the head segment. Highly reused data typically gets re-referenced quickly, making it unlikely to be evicted. The segment reserved for actively reused data is 70–80% of the cache. This protects the bulk of the cache from being flushed by data that will not likely be reused. For many workloads, the Segmented LRU cache miss ratios equal those of LRU caches twice the size.

1.3 Dissertation Outline

- Chapter 1 motivates the need for improved I/O caches. It describes I/O caches and disk systems, and the interaction between the two.
- Chapter 2 describes the methodology used throughout the research including the tracing, simulation system and file system model. It defines many of the metrics used to understand locality capture and to evaluate I/O cache performance.
- Chapter 3 evaluates the I/O characteristics of the workloads. The characteristics of the workload determine the usefulness of I/O caches, and form a basis for understanding the

cache mechanisms required to reduce disk requests effectively.

- Chapter 4 describes the cache characteristics of I/O workloads, both as unified workloads and then broken down by type. It shows that each type has a different cache performance characteristic, and that certain characteristics can be isolated.
- Chapter 5 describes attribute caches as a mechanism to exploit the workload characteristics and improve I/O cache performance.
- Chapter 6 presents the conclusions and discusses future work.

The raw data and individual data plots for the 11 workloads are available in a companion technical report - CSL-TR-94-656. This report is self contained; only those wishing to perform their own data analysis should consider the companion report since it contains numerous detailed data plots and little discussion.

The companion technical report, CSL-TR-94-656 includes:

- Appendix A has a quantitative analysis of each workload.
- Appendix B contains characterization data to support Chapter 3.
- Appendix C contains cache characterization data to support Chapter 4.
- Appendix D contains attribute cache results to support Chapter 5.

1.4 Contributions of this Dissertation

This dissertation extends I/O tracing by including both file system information and I/O system information, where previous traces only included one or the other. The additional information allows I/O characterization at the system level, and greatly increases the body of knowledge about the make-up and type of disk I/O requested. The new information shows that the I/O request stream contains statistically diverse components that can be separated.

The *attribute cache* is introduced as a mechanism to capture the statistically distinct behavior of the workload. The attribute cache has various partitions, each efficiently tailored to cache files with certain properties or attributes. Unlike other caches it does not rely entirely on the statistical nature of the I/O workload; information about an I/O request becomes an attribute that determines how best to cache a request. Using attributes, cache resources are allocated to capture specific types of I/O data locality.

Finally, the dissertation develops an I/O cache scheme using attribute caches to improve total I/O cache performance. The attribute cache scheme relies on workload characteristics to determine the appropriate cache configuration for a given cache size. For a set of eleven measured workloads, an attribute cache scheme reduced the miss ratio 25-60% depending on cache size, and required only about 1/8 as much memory as a typical I/O cache implementation achieving the same miss ratio.

Chapter 2

Methodology and Models

The tracing environment, simulation systems, and simulation model determine the kind of I/O behavior that can be measured and simulated. The tracing environment monitors and records only part of the existing system behavior. The recorded behavior constrains the kind of measurements that can be made and thus constrains the resulting conclusions. The simulation system measures facets of the original system and predicts performance for other systems. The simulation model interprets the recorded behavior and uses it to simulate other configurations. This chapter describes the trace environment, simulation system and models used to characterize the components and the cache behavior of I/O requests.

2.1 Tracing

To directly compare the effectiveness of various I/O caches requires delivering an identical I/O stream to each cache. Using an I/O trace guarantees repeatable use of identical I/O streams. The I/O activity can be traced as disk requests, file requests, or application I/O requests. Understanding the nature of application I/O requests can help improve I/O cache performance. This requires tracing application I/O requests and including information about each request. The consumer of the trace uses the extra information to determine the source of I/O requests, or the type of I/O requests and their expected locality and cache behavior properties.

A DECstation 5000 running ULTRIX generated the traces, using a new version of the WRL tracing facilities. The original system was designed primarily to study processor memory issues in a multi-programming environment [BKLW90, CBJ92]. Figure 2.1 shows the system configuration. A modified version of the ULTRIX 4.2 operating system stores trace information in a large buffer. When the buffer becomes sufficiently full, the kernel schedules a special process called the *analysis*

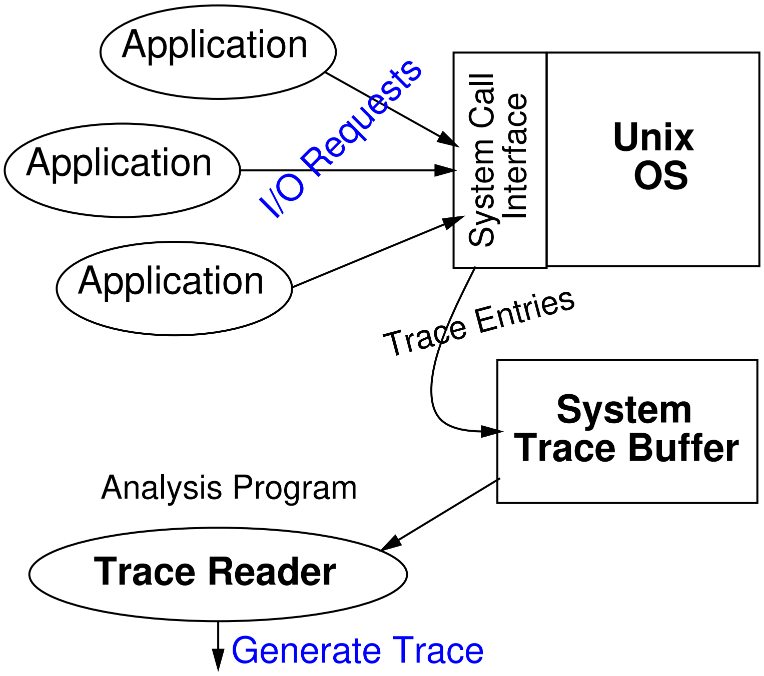


Figure 2.1: Logical diagram of trace, and simulation configuration.

program to read and process the buffer contents. The modified system logs system call information, rather than individual instructions, in the trace buffer. On an I/O system call, the call type, process ID, and call parameters are entered in the buffer. A separate entry in the buffer holds the return value, error status and information matching the appropriate call. The kernel traces all processes.

When the trace buffer is sufficiently full, the kernel invokes the analysis program. The analysis program runs until it has processed the entire contents of the system trace buffer. For these experiments, the analysis program matches call and return values, produces a file system event trace, compresses the trace and writes it to a file. The tracing has little effect on execution because the trace code executes fewer than one percent of the instructions. The analysis program can possibly affect process scheduling because it changes the timing of I/O relative to the other processes being scheduled. This, however, occurs infrequently. For each of the traces used here, the system invoked the analysis program fewer than ten times. Lastly, the kernel disables tracing while executing the analysis program, so as not to record the analysis activity.

The trace has the order in which system calls are issued, the order in which they complete, and the ID of the process that performs each system call. The trace contains timing information, but it represents only the system being monitored. The traces came from workstations in a distributed computing environment, and the timing includes load effects from many remote machines. The consumer of the trace can derive the actions of each system call from its call parameters and return values, and the type of the call. The kernel traces all system calls relating to I/O.

2.2 Workloads

This dissertation uses traces from actual user activity. The system collected traces in a manner entirely transparent to the user; the system ran normally. Using the modified operating system, the user saw no performance degradation from the tracing mechanism.

Most of the traces monitor several days of user activity. Such long traces are necessary to capture significant I/O activity and to show the interaction of the many large and small files that comprise a workload. For most workloads, I/O calls occur relatively infrequently; long trace periods are required to capture adequate I/O call events. For user driven workloads, the CPU is idle much of the time waiting for user commands which lengthens the time required for an I/O trace.

The traces cover a wide range of user applications and types of work. All traces were collected within a research laboratory with a broad range of activities. Individual traces cover activities of researchers involved in software and hardware development, as well as writing papers. Although not necessarily typical of heavy commercial use, such as large database systems, the traces should represent many engineering development and office environments. This thesis uses traces from eleven different workloads.

Kernel Build trace is a configuration and compile of the ULTRIX kernel for a diskless DECstation 5000 [Cop]. Fifty-eight modules are recompiled to make this kernel. The build takes about 2 hours.

Ingres Transactions performs 10,000 banking debit and credit transactions on an Ingres database [Sto86]. The transactions are entirely random, and there are no complicated searches. Indexes exist for all of the lookups. The database environment consists of a server and client. Both run on the same machine. The trace includes starting the server and running the 10,000 transactions.

Application Data Analysis manipulates a series of traces, simulates caches and displays cache results. A filter makes minor modifications to the traces using system utilities such as *fgrep* and *awk*. The cache simulations then use the traces, formatting the results into postscript plots.

Software Development 1 (8 to 5) develops and tests the ATOM simulation system [SE94]. ATOM is an instrumentation and simulation platform for understanding and debugging performance problems. It selectively instruments applications with an additional compiler pass, and incorporates libraries to count events or simulate performance of the program's execution. Although ATOM uses a standard set of libraries, the platform encourages architects to write custom applications and simulation libraries.

The machine performing this activity only logged traces during the day while doing actual work. The primary system user turned tracing on when they arrived, and off when they left for the day.

Software Development 2 (24 hr.) is the same basic environment as **Software Development (8 to 5)**. The trace includes idle time at night and all the maintenance activity that goes on at night. The trace records 4 days of activity.

Document Preparation records six hours of intense work on a technical report. It includes text processing, editing, simulation, drawing, and data manipulation.

Mecca Development (24 hr.) records the development and testing of a centralized e-mail system [Bor92]. Mecca uses an Ingres database with information about individuals to direct incoming mail to the proper recipient. This workload includes development and testing of the interface modules.

Mecca Development with Server (8 to 5) is the same basic environment as **Mecca Development (24 hr.)**, except that (1) the Ingres server resides on the traced machine, and (2) the machine recorded traces only during the day.

CAD: Chip Build traces the construction of a CPU layout from a high level description [DM92]. The high level description is in C, and much of the build involves standard compilation of the various chip pieces into a single program. The program then constructs the layout of the chip. This is not a complete layout because the chip is only partially finished. The workload ran design tests on the compiled chip description.

Network Update (24 hr.) mostly consists of a large network gather-scatter operation. The operation gathered information from the whole DEC NET and then updated the whole net with new information. Additional work included the typical UNIX applications, and some special network configuration tools that run on top of the Ingres database.

Compute Server (24 hr.) consists of batch simulations that ran on a machine with an idle console. The simulations evaluated potential architectural features from a compiler technology standpoint.

2.3 File System Model

I/O system call traces cannot be directly used for I/O cache simulations. The operating system maintains state from I/O calls for use by subsequent calls. Through the *open* system call, a process asks the operating system to open files. The process then gets an index number for future file references. Figure 2.2 shows a request for opening a file and subsequently reading the first 8 Kbytes of the file. When it opens a file, the operating system uses pathname information to locate the inode for the file and check access permissions. After verifying that the file exists and that the process may access the file as requested, the operating system returns a descriptor that the process then uses for further I/O requests on the file.

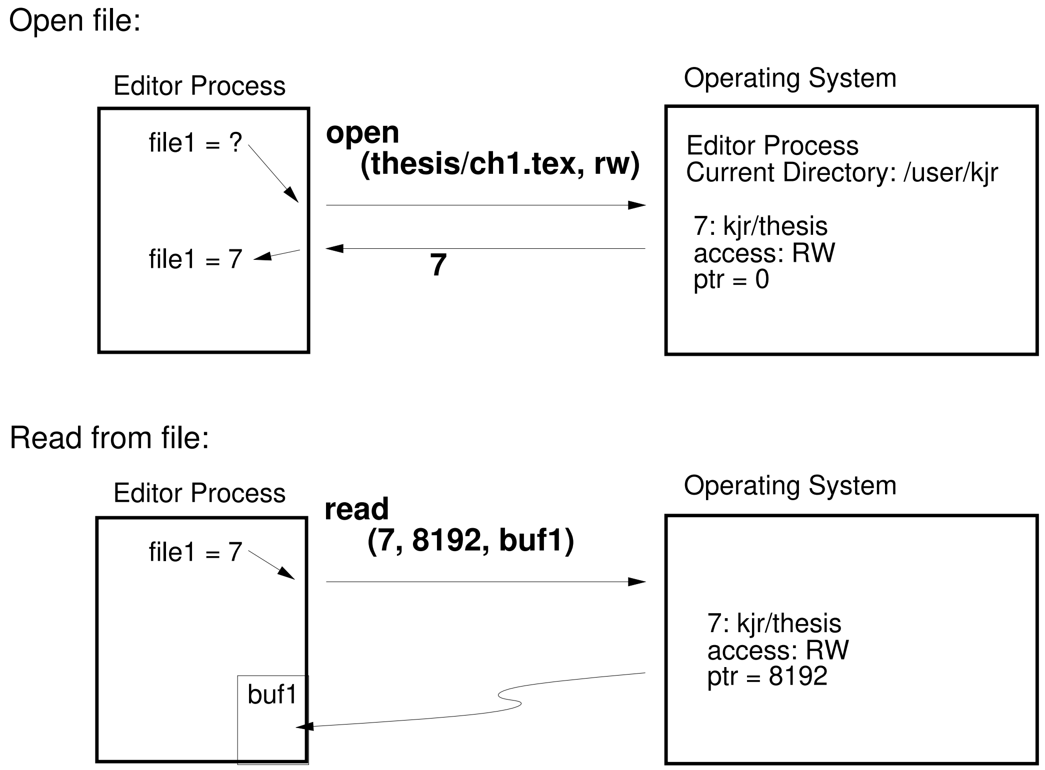


Figure 2.2: State changes resulting from process and operating system interaction on I/O system calls.

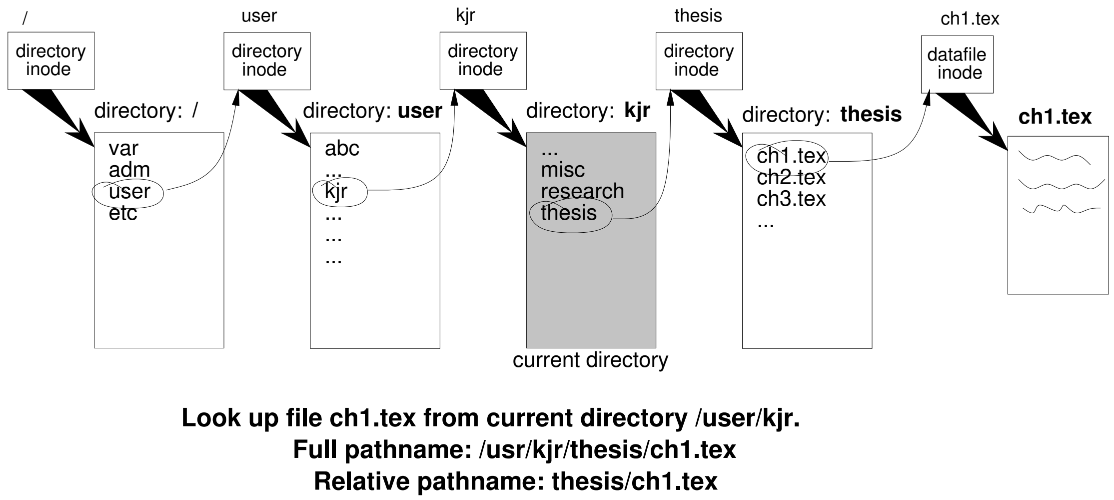


Figure 2.3: The inode and directory access associated with looking up a file.

In Figure 2.2, the editor process issues an *open* request for the file thesis/ch1.tex. The file name is relative to the current working directory of the process, so the operating system uses the current directory (`/user/kjr`) to begin the file lookup, as seen in Figure 2.3. After locating the file, the operating system establishes an entry in the file table with information on the file, the permitted process access modes, and a pointer to the inode containing disk locations for the file blocks. The operating system associates a file pointer with each file, initially set to the beginning of the file.

The I/O system call *read(7, 8192, buf1)* asks the operating system to read the next 8 Kbytes from the file associated with descriptor 7, and place the resulting data in process buffer *buf1*. The operating system uses the pointer to access the first 8 Kbytes of the file, and then adjusts the pointer. Subsequent reads deliver sequential pieces of the file. The process can reposition the file pointer with a special I/O system call.

2.3.1 Simulating File System Activity

Determining which file blocks are read and written requires simulating the operating system file table information and file descriptors [RT74, Bac86, LMKQ90]. Most workloads have many active processes, some of which might share files and file descriptors. Maintaining the simulated state requires much space, and evaluating the state for each I/O system call requires much time. It proved too costly to simulate the state for each I/O cache simulation.

A state-free trace can be generated from the original I/O system call trace by simulating the operating system file table and assigning unique identification numbers to each file. Figure 2.4 shows the simulation system, including the file system simulation. The stateless trace includes the filename of each I/O request in the form of a unique ID. It also includes the file type, which is implied by the system call and the explicit range of data bytes within the file. The stateless trace drives all I/O cache simulations.

In addition to file read and write requests generated by processes, the stateless trace also includes simulated inode, directory and executable file references. Inode and directory references occur the operating system opens a file, or evaluates access permissions. Figure 2.3 illustrates the inodes and directories read in opening a file. Inodes contain file access permissions and disk locations for the file blocks. Directories are special files that maintain file names and their associated inodes. From the user's point of view, directories generate the hierarchical tree of the file system. Inodes are an implementation detail not visible to the user.

The operating system writes information to inodes whenever the file state changes. Figure 2.5 shows the information it stores in a disk inode. Maintaining the access time means that an inode write must occur on each file access, regardless of whether the file itself is modified by the access. Inode writes from file modifications are modeled as a single write when the file is closed.

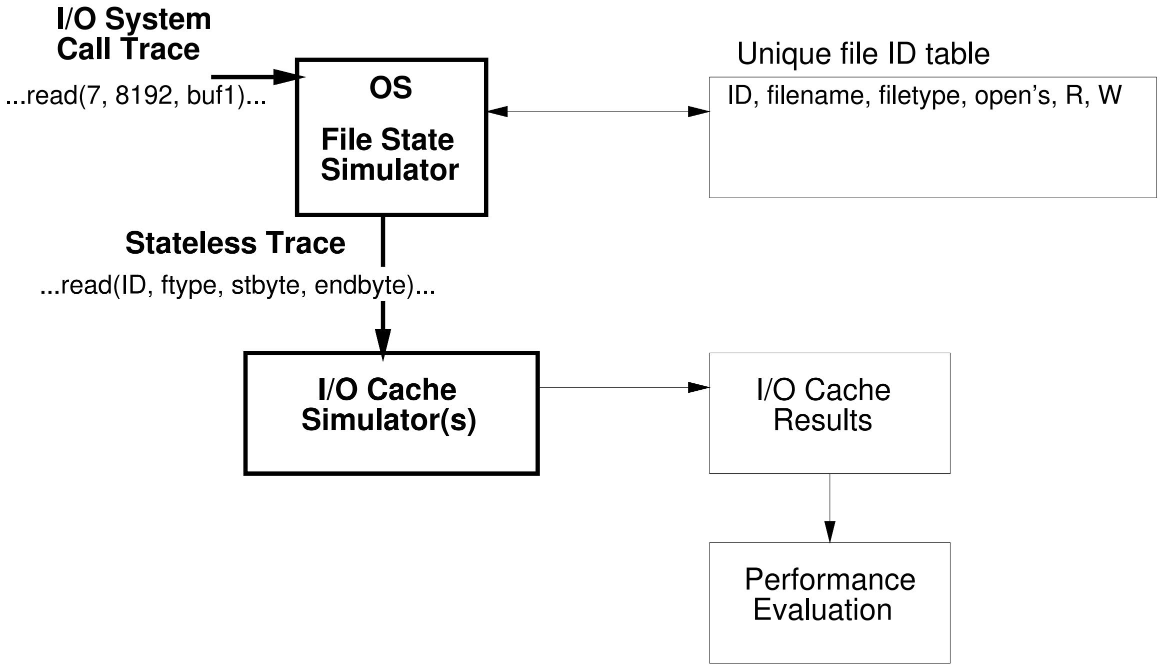


Figure 2.4: Overall simulation system.

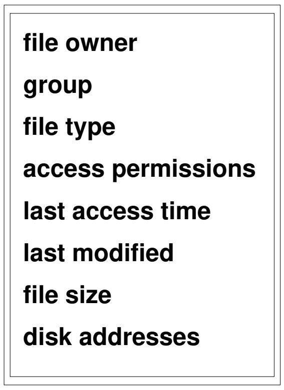


Figure 2.5: Inode information.

For a process to run a program, the operating system must look up the executable file and check its access permission. The simulator models program executions as a file lookup followed by a single read of the entire executable file.

2.3.2 File Types

In a UNIX system, I/O requests resulting directly from program execution can be grouped into four categories: datafiles, executables, inodes, and directories. Datafiles are explicitly read or written by active processes. Executables are run by processes, and initiated via one of the *exec* systems calls. Inodes and directories contain meta-data. Inodes contain information used by the operating system to locate the actual data on the disk. Directories facilitate the user organization of data and point to inodes. The access behavior determines the file type:

Datafile Reads are usually buffered by the I/O libraries. The reads are large and most of them request a fixed amount of data: 2, 4, or 8 Kbytes. The size of the reads depends on the I/O libraries used, or the application. In the traced systems, the FORTRAN I/O libraries use 4 Kbyte reads, while the C libraries use 8 Kbyte reads. Ingres, a database application, has its own libraries that use variable size reads.

I/O cache misses on data reads are costly. The process usually blocks and a context switch occurs if possible. The cost of a read miss includes (1) the overhead of the context switch; (2) the penalty of lost local state, including additional CPU memory cache misses and page faults when restarted [MB91]; and (3) any idle cycles resulting from the disk access.

Datafile Write requests are often smaller than a block, and of arbitrary size. Writes often occur sequentially, making spatial locality important. Most writes can happen asynchronously with other computation, including other writes. Three common write policies include: (1) *Copy-back,* where writes may be copied back to disk only when they are evicted from the cache; (2) *write-through,* where writes may be written through to disk immediately, in which case they may not even be allocated in the cache; or (3) *periodic write,* where writes may be periodically written to disk. Periodic writes limit the time during which the disk and cache have different copies of data. Misses on data writes vary in importance depending on the allocation and management policy and size of the cache.

Executable Files are read only. Executable files are generally large and could potentially be read from a single cylinder if properly placed on disk. The actual access pattern depends on the executable management in the operating system. The model used here reads the entire executable from the disk into the I/O cache with one request. Subsequent requests by the operating system to bring the executable into memory will then hit in the cache. Because the executables are large, however, bringing them into the cache all at once may flush other useful data.

Inode Reads and Writes. An inode is the Unix term for the data structure that maintains information about files, including the location of file data blocks on disk. Figure 2.5 shows the information contained in an inode. To reference a file initially, the operating system first must find the associated inode. The filename and directory pathname in the system call (*open, create, link,* or *execute*) determine which inodes and directories to reference. Inodes are a fixed size—128 bytes in ULTRIX version 4.2. All inodes reside in common areas on disk [Bac86, MJLF84].

Inode read misses are costly in the same way as datafile read misses. Before it can access a datafile or an executable, the operating system must access the associated inodes. The inode structure requires that inodes be serially accessed. A single file access can cause multiple inode misses.

There are two types of inode writes. (1) The operating system writes inodes whenever files change. These writes reflect changes in the file system structure. (2) The operating system writes inodes to update file access information. These writes do not correspond to file system changes or even actual data modifications.

Directory Reads and Writes. Directories provide a hierarchal organization for files. Directories link file names to inodes. To initially reference a file, the directories in the pathname need to be read. Figure 2.3 shows the steps required to find a file with the path and name `/user/kjr/thesis/ch1.tex.` Each process has a *current directory*. System calls can specify file lookup requests with full pathnames starting from the root directory "/" or they can be specified relative to the current directory. In Figure 2.3, accessing the file `ch1.tex` relative to the current directory `/user/kjr` requires one directory access and two inode accesses.

Directory read misses are costly in the same way as datafile read misses. On file lookup, the operating system accesses directories and inodes serially. Directory writes occur only when the operating system creates or deletes a file, or when it changes the directory permissions.

2.4 I/O Cache Simulation Model

A single cache simulator, applied and configured in many different ways, generated the I/O cache results in this dissertation. The I/O cache simulator models fully associative I/O caches using an LRU replacement policy. Fully associative caches efficiently simulate multiple caches simultaneously [TS87, Tho87].

To simulate the I/O request stream behavior, a simulator must track I/O requests along with cache block behavior. An individual I/O request may encompass several cache blocks, each of which may hit or miss in the cache. Figure 2.6 shows how the request manager generates I/O cache block references and then uses the LRU stack hit depth or cache miss information to determine the appropriate number of request misses for each given cache size. The request manager also records

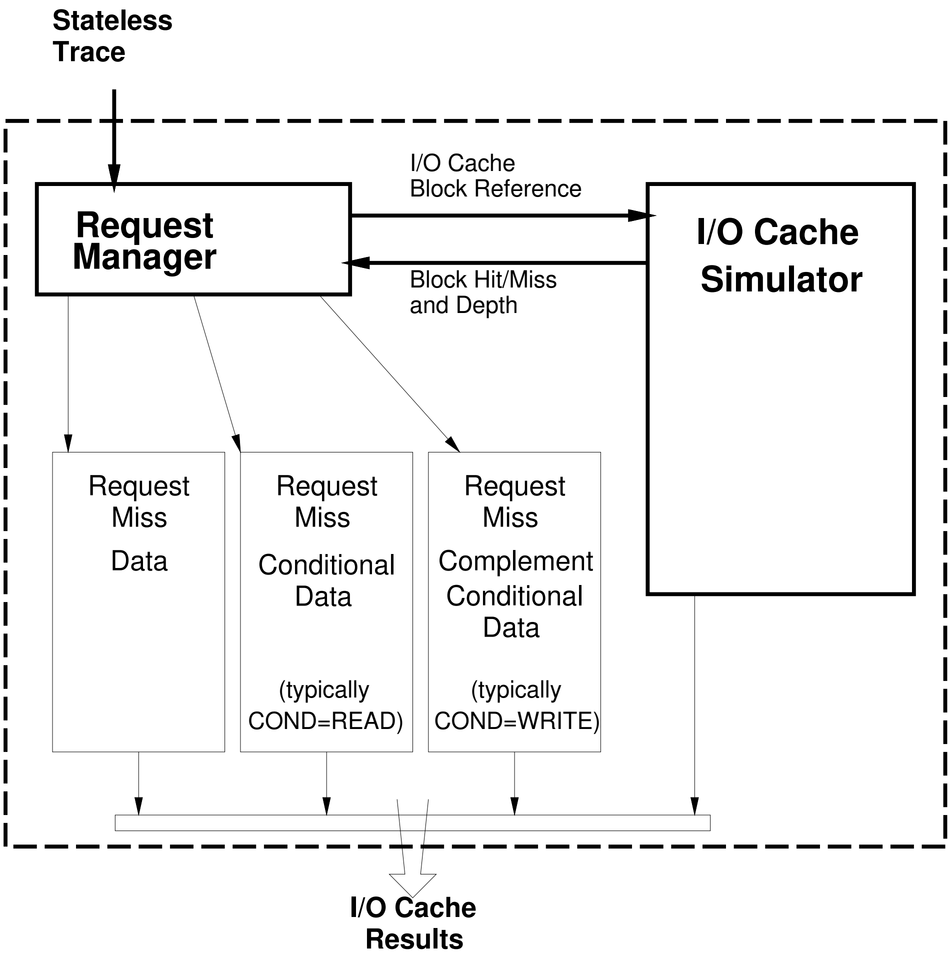


Figure 2.6: Cache simulator layout.

the request misses based on a supplied condition (COND) and its complement. The simulator typically uses the condition to break down read and write requests separately. A single simulation produces a full range of I/O cache behavior, request miss behavior, and request miss component behavior. The TIME toolkit, tools for I/O and memory evaluation contains early versions of the cache simulators [RF92].

2.4.1 Request Model

The request model describes I/O requests from the application viewpoint and how the requests are broken down into cache block requests. An application, requests file I/O. The size of the request is independent of the underlying I/O cache configuration and cannot be used to directly access a cache block. Each I/O request turns into one or more cache block requests. The number of cache block requests generated by an I/O request depends on the original request size, the block size and the offset of the request from the beginning of the file. The number of blocks multiplied by the block size is always greater than or equal to the request size.

Each file has a unique file ID. This ID, along with an offset into the file, form the address for a data request. The file offset is converted to a cache block offset that is then used to access the I/O cache. The cache matches two tag fields: one for file ID and one for block offset. Full associativity eliminates the problem of generating a single uniform index. A cache block does not hold data from more than one file. Caches that store actual disk blocks, rather than file blocks could have pieces of several files in a single block. This requires knowledge of the disk layout; the traces did not contain actual disk block addresses, so this was not modeled.

If the request size is larger than the cache size, the requests is modeled as multiple requests. This is necessary because all requests go through the I/O cache before being delivered to the application. A request size greater than the cache size incurs multiple request misses. A small cache might therefore have more request misses than actual requests.

The cache simulates block requests. Each block can produce a cache miss. An I/O request that generates multiple block requests can produce multiple cache misses. All the block misses for a request are coalesced into a single request miss. A single disk assess is then issued for each request miss. Disk blocks for a single file are likely to be located together on disk. Requesting all the blocks for a request reduces the operating system overhead for initiating the request and transferring the data.

Request misses reflect the disk activity caused by I/O cache misses. A request misses in the cache if any of the cache blocks it accesses miss. The size of the request miss depends on how many blocks miss in the cache. Read request misses reflect the number and size of read disk accesses for a workload. The number of write accesses depends on the write policy and the cache activity. In a non-volatile cache, disk writes occur when previously written data becomes the least recently used entry in the cache and is then evicted. These are termed write expulsions. Each disk write is a cache block in size. More advanced write expulsion techniques exist [Red92b], but they were not modeled. Their impact should be similar to other reported results.

2.4.2 Cache Model

The cache model describes the I/O cache parameters and the cache configurations supported in the cache simulator. This dissertation uses several different I/O cache configurations. All are non-volatile caches, fully associative with a least recently used replacement policy, uniform block size, and a fetch size equal to the cache block size. Section 1.1.2 discusses this general I/O cache configuration. A unified I/O cache has only three parameters, but caches can be split into subcaches, each with its own set of values for the three parameters. The three parameters are:

block size The block size the management unit used in the cache; it is the granularity of storage, lookup and data transfer. The block size determines the amount of temporal and sequential locality a particular size cache can capture. A cache with a smaller block size contains more blocks and can support more independently referenced entries. A cache with a larger block size captures spatially adjacent references within a file. Small block sizes tend to capture more temporal locality, while larger blocks capture more sequential locality.

cache size The cache size, along with the block size, determines the number of entries in the cache.

attribute An attribute is a property associated with each file. Each cache selectively caches all or part of the workload. A designer uses the *attribute* to evaluate the cache behavior of parts of the workload and determine feasible parameters for split caches. The cache accepts files with matching attributes and rejects the rest. Chapter 4 evaluates cache behavior of each file type independently, using file type as the attribute. In Chapter 5, subcaches with different attributes are grouped together to form a cache for the entire workload.

The split cache shown in Figure 2.7 is an example of an attribute cache. This cache uses file type and file size as attributes. The first subcache, labeled “ID cache,” holds only inode and directory file types. The other two subcaches hold executable and datafiles, but the *temporal cache* holds small- and medium-sized files, while the *sequential cache* keeps large files. The relative size of each individual subcache depends on the total cache size. Each subcache has a vastly different block size, to efficiently capture the locality of each component. Chapter 5 shows the design requirements for such a cache.

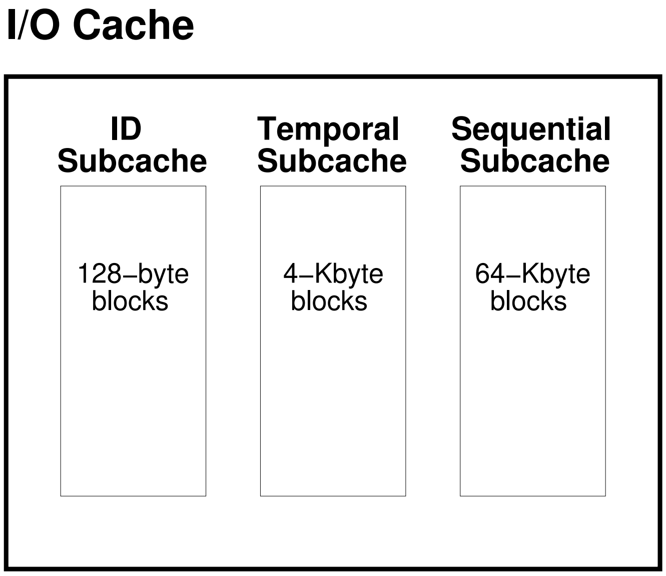


Figure 2.7: Sample split cache configuration.

2.4.3 Write Policy

A non-volatile cache reliably stores newly written data in the cache. The cache writes data to disk only when the data gets evicted from the cache. Each evicted block produces a disk request.

The I/O cache simulator implements a pure LRU replacement policy; it evicts the least recently used entry when a cache miss occurs. Evicting a dirty entry causes a write. All data writes are thus caused by misses, and the write of the evicted data must be completed before servicing the miss. In practice, the cache evicts the least recently used clean entry and asynchronously writes the dirty entries to disk after servicing the miss. This eliminates the need to service the write evictions before servicing the miss [NWO87, BAD$^+$92].

Asynchronously evicting dirty data from the cache produces the same number of disk writes as a pure LRU scheme. It also has a similar total write service time, because it perturbs ordering of only the least recently used cache entries. A scheme using a write buffer could produce similar results. The I/O cache results assume an asynchronous write scheme.

When used in a certain way, an asynchronous eviction policy can even reduce the number of disk writes. The cache can scan the last portion of the LRU stack for adjacent dirty blocks, coalescing them into single writes. Coalescing multiple writes into a single write reduces the total write service time by reducing the write overhead. Log structured file systems use this technique to reduce the overhead of periodic write flushing [OD88, RO91].

2.5 Disk Access Model

A simple disk access model approximates the disk access time. Individual disk accesses incur overhead to locate the proper disk location and then a transfer time determined from the transfer rate R_{xfer} and the number of bytes transferred n.

$$\text{Disk Access Time} = T_{overhead} + R_{xfer} * n \tag{2.1}$$

$T_{overhead}$ includes any queue waiting time, request transfer time, disk seek time, and disk rotational latency time. Only exact I/O timing and disk block locations can generate the individual overhead terms. Disk system studies often simulate the individual components as distributions. The distributions provide a reasonable model of disk access behavior [Kim86, KT87, Ng91].

I/O system studies primarily measure the total disk time or the number of disk requests generated by the workload. The workload disk requests may be spread across many different disks, and many other workloads may be accessing those same disks. The total disk access time for a workload can be estimated using an average value for the overhead. The workload access time becomes

$$\text{Disk Access Time} = T_{overhead} * N_{requests} + R_{xfer} * N_{bytes} \tag{2.2}$$

where

$$\begin{aligned} N_{requests} &= \text{number of requests} \\ N_{bytes} &= \text{total bytes transferred} \end{aligned}$$

$T_{overhead}$ is typically 20 ms, and R_{xfer} is typically 0.5 ms/Kbyte [RW94b].

2.6 Performance Measures

Disk I/O impacts total system performance. The overall I/O system performance determines the amount of time spent performing disk I/O. Reducing the number of disk I/O requests and the amount of time spent on disk I/O reduces the potential impact on total system performance. Fewer disk accesses mean either more efficient I/O or less I/O; both require less time. Fewer disk accesses for a process results in fewer arbitrary context switch points. This reduces I/O related context switch overhead and improves system efficiency [MB91].

An I/O cache is judged by its ability to capture I/O requests and eliminate disk accesses. Several different measures can evaluate the cache performance. They include the cache hit and miss ratio, the request miss ratio, the byte transfer ratio and resulting disk service time.

2.6.1 Cache Hit and Miss Ratios

The cache hit ratio measures the ratio of requested cache blocks found in the cache to the total cache block requests. The cache miss ratio expresses the same information in a different way:

$$\text{cache miss ratio} = 1 - \text{cache hit ratio}$$

The cache hit ratio is simple to measure, and helps understand a workload's cache properties.

2.6.2 Request Miss Ratio

While the cache miss ratio measures cache block requests generated by the cache, the request miss ratio measures I/O requests generated by the operating system. The request miss ratio is related to the number of requests that miss in the cache relative to the total requests. A request misses in the cache if any block of the request misses in the cache. Requests smaller than the I/O cache size generate at most a single request miss. Requests larger than the cache can cause multiple request misses, since multiple sequential replacements must be performed.

The request miss ratio includes both reads and writes. The read request miss ratio measures only read performance. Read request misses cause a process to stall, waiting for the I/O to complete. While multiple cache read misses might be coalesced into a single disk access, each read request miss generates a single disk read request. Thus a lower read request miss ratio always means fewer I/O waits.

2.6.3 Write Expulsions

Write expulsions are the cache blocks that must be written to disk when the cache expels previously written, or *dirty,* blocks. Because of the high cost involved in tracking write expulsions, the I/O cache simulator only produces this result for the largest cache simulated. Each write expulsion generates a disk write request.

2.6.4 Estimated Bytes Transferred

The number of bytes transferred to and from disk determines the data transfer time for disk accesses. The simulator calculates the bytes transferred by multiplying the cache block misses times the cache block size. How well does this number reflect the actual bytes transferred?

1. The granularity for disk reads and writes is a disk sector. Thus, byte transfer ratio accurately measures the bytes transferred if the cache block size equals the disk sector size. Otherwise, the byte transfer ratio is high.
2. For larger block sizes, parts of the block may be empty, resulting in an overly high byte transfer ratio. For example, in smaller caches with large block sizes, such as a 64K cache with 16K blocks, small requests to files of fewer than 2 Kbytes generate most of the misses. The bytes transferred assumes that the disk transfers a whole 16 Kbyte block on each miss. In reality, the operating system stops transferring bytes when it reaches the end of a file.

Thus the estimated bytes transferred always over-estimates the number of bytes transferred. The over-estimation is greater for cache configurations with large blocks.

The estimated byte transfer ratio uses 128-byte inode and directory cache blocks, and 4-Kbyte datafile and executable cache blocks to determine the reference byte transfers. The byte transfer ratio includes both cache read miss and cache write miss bytes. The read byte transfer ratio includes only read cache miss bytes.

2.6.5 Service time

The disk service time reflects the time spent performing disk I/O.

$$\text{disk service time} = (\text{read service time} + \text{write service time})$$

The read service time is based on the number of read request misses and the number of bytes transferred to service the misses. The simulator measures read request misses directly, but the number of bytes transferred is estimated from the missed cache blocks, as discussed in subsection 2.6.4 above. Thus,

$$\text{read service time} = (\text{read request misses} \times \text{disk access overhead}) + (\text{read bytes transferred} \times \text{disk transfer rate})$$

The write service time is based on the number of write expulsions and the number of bytes transferred to disk. The simulator measures the number of expulsions directly, but the number of bytes transferred is estimated from the cache block sizes.

$$\text{write service time} = \text{write expulsions} \times \left[\text{disk access overhead} + (\text{cache block size} \times \text{disk transfer rate})\right]$$

The accuracy of these measures depends on the cache size, the cache block size and the relative time for the disk access overhead and disk transfer. The service time measure is most accurate for large caches, small blocks, and disk overheads that are large in comparison to the disk transfer time. Inaccuracies in the service time come from inaccuracies in the byte transfer count, and thus always overestimate the actual time.

Chapter 3

I/O Characteristics

This chapter describes the I/O requests made to the operating system by a set of workloads. Section 2.2 described the workloads. The resulting body of data should help determine the best way to reduce the impact of I/O on the overall computer system performance.

Knowing how data is used and reused can determine how effective different algorithms will be in reducing the time spent waiting for I/O to complete, and reducing the required I/O. Data buffers and caches can cause I/O workload requests to differ significantly from the I/O requests made to the disk hardware. The operating system buffers file data and file system structures at many different levels. Some buffering occurs within a process, and some occurs across processes.

Knowledge of the statistical distribution of I/O requests can help design systems that achieve the best price/performance ratio. Some techniques for improving performance rely heavily on the statistical nature of requests. Caches work because, statistically, programs and work environments exhibit locality. A better understanding of the statistical properties of the workloads allows cache designers to exploit the most prominent statistical properties of the workloads. The characterization shows the type of requests that are the most important to performance, and focuses attention on the most important part of the problem to be solved.

Although a cache or disk system rarely needs any information about the nature of the data or why it is requested, this information can provide insight into the root cause of I/O requests. The source of I/O requests can guide cache policies or system management policies. Understanding I/O behavior to improve locality capture across processes is the main focus of this work.

This chapter describes the characteristics of I/O with no cache effects.

1. Characteristics by type of data transferred.

2. Characteristics by transfer type (read or write).
3. Characteristics by file size.

3.1 Static and Dynamic I/O Requirements by Type

Static measures count unique objects and ignore the frequency with which the objects are used. The static I/O requirements of a workload measure the unique I/O data and meta-data that is either read or written. Assuming no data buffering between workloads, this is the minimum amount of I/O that the I/O must perform for the workload. This chapter uses two static measures:

Static objects are the number of unique objects touched by the workload. An object is a datafile, an inode, a directory or an executable file.

Static size is the total amount of space required to store all the objects touched by the workload.

Dynamic measures count the total references made by a workload. The dynamic I/O requirements represent the total I/O performed to support the workload applications. Assuming no data buffering other than that explicitly performed within the applications, this is the amount of I/O performed by the workload. This chapter uses two dynamic measures:

Dynamic references are the number of requests made for objects, either whole or partial.

Dynamic bytes transferred is the total number of bytes requested by the workload. These are the bytes transferred through the I/O cache, not the actual I/O to or from disk.

3.1.1 Breakdown of Static Objects by Type

Figure 3.1 breaks down, by type, the unique objects referenced by each workload. An object is a unique file: either a datafile, an executable, an inode or a directory. Most of the unique objects are either inodes or directories (greater than 75%). Inodes make up more than 50 percent of the unique objects accessed in all workloads, ranging from 51.3% for *Kernel Build* to 86.7% for *MECCA Development*. The file system associates an inode with each directory, datafile, and executable file, and the inode must be accessed to access the other objects. Thus, at least 50% of the static objects must be inodes. Datafiles make up the majority of the remaining objects, ranging from 6.9% for *MECCA Development* to 41.7% for *Kernel Build*. The combination of inodes and datafiles accounts for at least 85% of the unique objects in each workload, with an average of 91% over all.

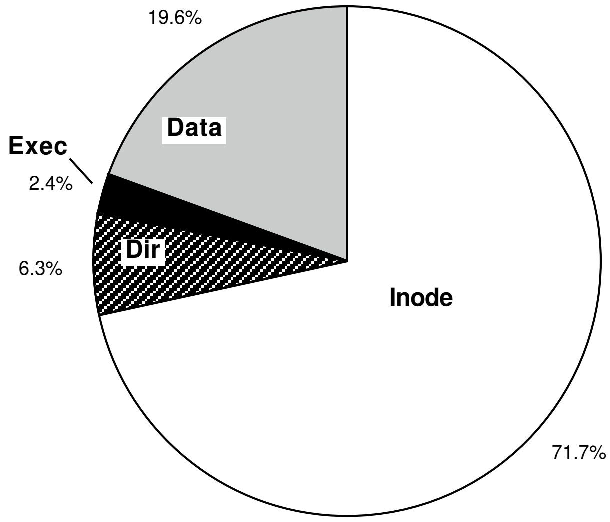


Figure 3.1: Breakdown of unique objects by type, averaged over all workloads.

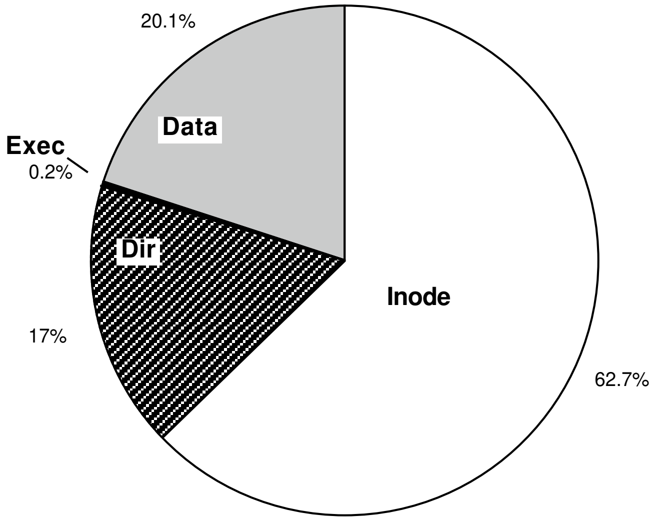


Figure 3.2: Breakdown of I/O references by type, averaged over all workloads.

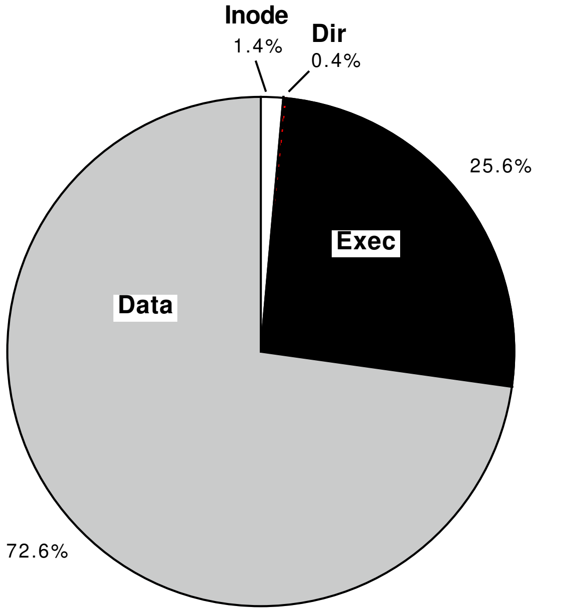


Figure 3.3: Relative physical space requirements broken down by type (static size), averaged over all workloads.

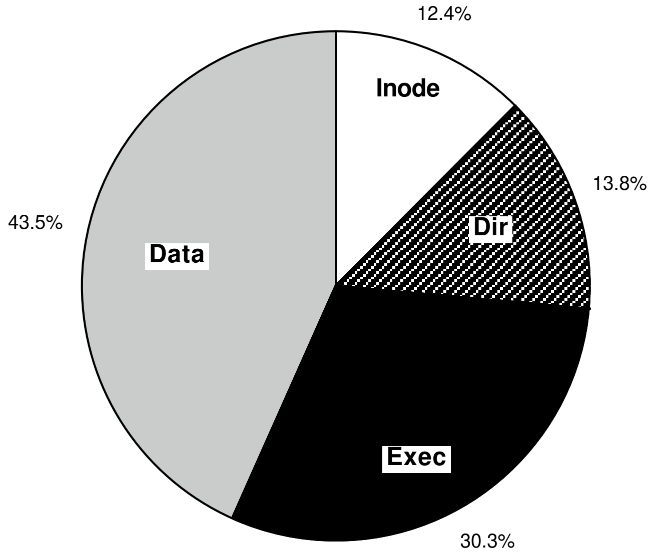


Figure 3.4: Breakdown of bytes transferred dynamically by type (dynamic size), averaged over all workloads.

3.1.2 Breakdown of Dynamic References

Figure 3.2 shows the average breakdown of references by type. A reference is a request for all or a piece of an object. Inodes are only referenced as whole objects, and the model used here does the same for executables and directories.

The breakdown of reference types shows the relative frequency with which a workload requests each type. For datafiles, references are the application's actual I/O calls. They incorporate the data needs of the application and the buffering properties of any I/O libraries used.

In the dynamic stream, executable references make an insignificant contribution to the total requests (less that half a percent). If executables were referenced in pieces, their contribution to the whole would be greater. Directory references contribute 10% to 20% of the total requests, a much higher percentage than their static object count. Inodes are still the most frequently referenced type in all the workloads except *Ingres*, contributing between 41% and 73% of the total references. On average, datafiles make up 20% of the total references, about the same as their static object contribution. For individual workloads, the datafile reference contribution does not correspond to its static contribution. The relationship varies considerably with workload.

Workloads that run *Ingres* as all or part of the workload have a larger datafile reference contribution than the static contribution. This is because *Ingres* randomly accesses a few very large files. The large files have a relatively low object count, but the reference count is high because they are repeatedly referenced.

In summary, executables contribute much less to the dynamic references than to the static object count. Directories do just the opposite; they contribute much more to the dynamic requests than to the static object count. The bulk of the requests are still inodes and datafiles. Their relative contribution to the dynamic reference stream is workload related and not related to their static object contributions.

3.1.3 Static Size

Figure 3.3 shows the relative physical space requirements for each of the data types. Each type has a different average size, and therefore requires a different amount of storage space. The total static size of all objects forms an upper bound on the amount of storage space required to capture all re-references to the objects. Since the working set changes over time and some parts of the datafile never get referenced, the actual space requirements for full capture may be less.

For all the workloads, the inodes and directories occupy only a small portion of the total space requirements. For half of the workloads they are less than one percent of the total; in all workloads

Object Type	Average Reuse
Inodes	223.3
Directories	707.4
Executables	32.1
Datafiles	241.6

Table 3.1: Average reuse ratios for each type

they are less than 5%. Executables contribute much more. They always occupy more space than the inodes plus directories. Their relative contribution varies from 1.7% to 41.6% of the space requirements. Datafiles require the majority of the total space requirements, averaging 73%. Their relative contribution varies from 57.3% to 97.9%.

3.1.4 Dynamic Bytes Transferred

Figure 3.4 shows the average breakdown of bytes transferred dynamically by type. The number of bytes transferred dynamically, also known as the dynamic size, is the amount of data that the operating system must deliver to or from the applications by the operating system. With no I/O cache, this is the number of bytes transferred to/from disk.

Although inodes and directories comprise only a small portion of the static size requirements, they make up a significant portion of the dynamic size. Together, they account for 5% to 49% of the traffic bytes. The byte traffic component from inodes and directories is almost equal. The directory entries are about four times the size of the inodes, but have about a fourth the number of dynamic references, resulting in comparable dynamic byte requirements.

The executable component of bytes transferred is larger than would be expected from the other measures. The average executable is larger than the other types, and each request retrieves the entire file. A small number of requests transfer a large amount of data.

The majority of bytes transferred are executables or datafiles. They comprise at least 50%, averaging 74%. Approximately 25% of the bytes requested by the workload are for meta-data rather than real data. If executables are included as overhead, then only about 45 % of the bytes requested from the operating system are used as actual computation data.

3.1.5 Ratio of Dynamic References to Static Objects

The *reuse ratio* is the number of dynamic references per object. Table 3.1 shows the average reuse ratios for the various types.

Object Type	Average Reuse
Inodes	223.3
Directories	707.4
Executables	25.5
Datafiles	10.1

Table 3.2: Average byte reuse for each type.

For inodes, directories, and executables, a dynamic reference accesses the entire object. Thus, the ratio of dynamic references to static objects gives the average usage for these object types. Datafile requests often reference a subset of the entire file, so the ratio measures both reuse and uncaptured prefetching.

On average workloads use executables many fewer times than either inodes or directories. The average workload makes 32 references to each executable, as compared to 223 references per inode and 707 per directory. Directories have the highest reuse ratio because many of the files for an application or even the entire workload reside in a small number of directories. Directories contain information about several files and, afterwards, the directory contents need to be checked whenever a file is opened. Since many files are associated with a single directory, the directory is often highly reused.

Inverting the reuse ratio yields the *infinite cache miss ratio* for inodes, directories and executables. The infinite cache miss ratio for whole executables is 3.13%, for inodes 0.45%, and for directories 0.14%.

Datafile requests often access only a portion of the file. In this case, the application generates multiple references to access the entire file. In Table3.1, the reuse ratio for datafiles includes all multiple references. Most multiple requests sequentially access data. As applications consume or generate data sequentially they make I/O requests for subsequent datafile pieces as execution continues.

A cache can potentially capture the sequential locality of the multiple requests. The reuse ratio for datafiles includes both true reuse, which indicates temporal locality, and adjacent use, indicating spatial locality. Including both of these results in a high reuse ratio for datafiles, averaging 242 for the workloads.

3.1.6 Ratio of Dynamic Bytes Transferred to Static Size

The *byte reuse ratio* is the number of bytes transferred to and from an object relative to the static size of the object. Table 3.2 shows the average byte reuse for the various types. To the extent that an object's static size relates to its size in the cache, the byte reuse ratio is an effective cache area utilization measure. Since the model assumes a single constant size for all inodes and directories, the average byte reuse ratio is identical to the average reuse ratio.

The difference between the average byte reuse ratio and the reuse ratio indicates the size distribution of the data requested.

For executables the workload always requests a whole file, so any differences in the two ratios comes entirely from the uneven size distribution of the executables executed. The average byte reuse ratio for executables is 20% lower than the average reuse ratio. This means the workloads make proportionally more re-references to the smaller executables than to the larger executables.

The byte reuse ratio for datafiles is much lower than the reuse ratio. The byte reuse ratio measures the number of bytes requested relative to the total size of the datafiles. Multiple requests to different portions of a datafile do not increase the byte reuse ratio as they did for the reuse ratio. For some datafiles, only a portion of the file is ever accessed by the workload, which tends to result in further lowering the byte reuse ratio. The accessed portions are not used frequently enough to compensate for the unreferenced portions. In addition, the workloads make many of the references to the smallest datafiles.

The reuse ratio of 242 references per datafile might indicate considerable locality, but the byte reuse ratio of only ten bytes transferred per static byte indicates that most of the locality is either spatial or occurs from small files. A reuse ratio of 242 from temporal locality would mean a high reuse rate for small sections within files, while much of the files remain untouched. The fact that few files have such a random access pattern (the vast majority of files get accessed sequentially), leads to the conclusion that the locality is not temporal.

3.1.7 Conclusions from Static and Dynamic Type Characteristics

Different I/O types differ in importance. Inodes and directories compose a significant percentage of the objects, but they are small and require little space. They also have the greatest average reuse ratio, so they are the most important objects to keep in the cache.

As Figure 3.3 showed, executables require considerably more space than do directories and inodes, and datafile objects require still more. The model represents executables, directories and inodes as indivisible objects, so if the cache does not contain the entire object, the request misses. Figure 3.5 shows the relative differences in usage and static sizes for the different types.

To get the most benefit, a cache should capture the most references with the smallest amount of space. I/O accesses have two components: an overhead to access the data, and a per-byte time to transfer the data. Thus, a good cache policy should minimize the number of requests as well as the actual amount of data retrieved. Reducing the number of misses decreases overhead, even if the

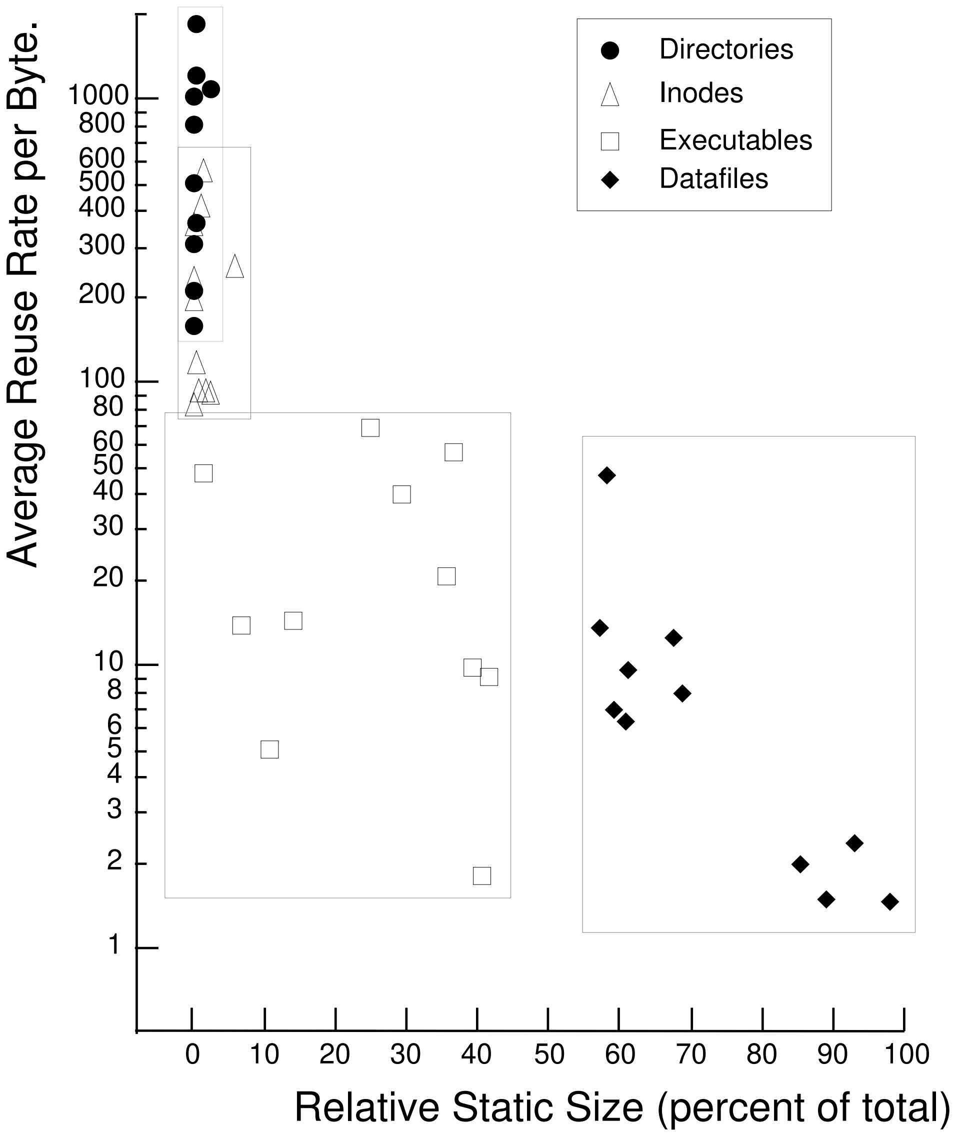


Figure 3.5: Comparison of the relative size and byte ratio, by type, for all the workloads.

File type	Percent of requests	Breakdown for type (percent)	Percent Read requests	Percent Write requests
Average				
Inode	62.7	51.1 Read	32.1	
		48.9 writes		30.6
Directory	17.0	99.5 Read	16.9	
		0.5 Write		0.1
Executable	0.2	100.0 Read	0.2	
Datafile	20.1	54.2 Read	11.3	
		45.8 Write		8.8
Total RW Breakdown			60.5	39.5

Table 3.3: Average breakdown of read and write requests.

amount of data transferred is not decreased. Because of the large differences in the object size and reuse between the various types, a cache scheme that relies on the statistical properties of the whole group will not match the properties of any one group.

3.2 Read and Write I/O Components

3.2.1 Read and Write References

Reads and writes have different requirements for performance and functional implementation. Reads generally force computation to wait until the data returns. Writes can often be performed asynchronously, but this involves addressing issues of consistency and reduced data integrity. Table 3.3 shows the average breakdown of read and write references for the workloads.

Almost half of all inode references are writes, while very few directory references are writes. Writes to inodes and directories potentially affect the file system structure and file access permissions. Inodes and directories maintain the structure of the Unix file system. Inodes store the location of file blocks, the size of the file, and modification and access data. This access data includes the last time a file was read as well as when it was last changed. This means that the operating system *updates* an inode each time it reads the file or directory maintained by the inode. These inode *updates* do not alter the file system structure or file protections, so they have the same importance as normal file writes. Inode reads account for 50% to 52.3% of inode references, inode update writes for 47.4–49.8%. Inode writes, which change the file system or file permissions, account for only 0.1–0.6% of the inode references. Directories contain no access information, and any modifications to directories alter the file system structure. These file system modification writes account for less than 2% of the directory Less than one-half percent of all references (0.1–0.4%) are writes to directories or inodes that are required to maintain the consistency of the file system.

File type	Percent of bytes	Breakdown for type (percent)	Percent Read bytes	Percent Write bytes
Average				
Inode	12.4	51.1 Read	6.4	
		48.9 writes		6.0
Directory	13.8	99.5 Read	13.8	
		0.5 Write		0.1
Executable	30.3	100.0 Read	30.3	
Datafile	43.5	64.7 Read	28.8	
		35.3 Write		14.6
Total RW Breakdown			79.3	20.7

Table 3.4: Average read and write byte transfer measures.

Executable references are entirely read requests. A file that is written as a datafile for later execution is classified as a datafile. This occurs when the user compiles a program. The compiler reads source files as datafiles and writes the executable output as a datafile.

Writes to datafiles constitute the bulk of writes that affect the user data. The breakdown of read requests and write requests varies significantly with workload. The fraction of reads reaches as high as 88% and as low as 20%.

The total read requests exceed the total write requests for all the workloads. For a couple of workloads, the read and write requests are almost equal. The ratio of reads to writes goes as high as 2.7 for workload Network Update, but the overall read/write ratio for requests averages only 1.5.

3.2.2 Read and Write Byte Comparison

Table 3.4 breaks down the average bytes transferred by reads and writes. The ratio of bytes read to bytes written greatly exceeds the request ratio. There is a much greater volume of read data than write data. On average, a workload reads four bytes for every byte written. Many large files are read-only, including executables. Executables account for a very small fraction of requests, but form a considerable fraction of the bytes read. Read requests to data files tend to be larger than write requests, which is reflected in the larger read component for bytes transferred than for requests.

Ingres is the only workload that writes more bytes than it reads. It is also the only synthetic workload. Although *Ingres* appears in several of the other workloads, it doesn't seem to skew the read/write ratio. In general, the volume of read data greatly exceeds write data, making data transfer optimizations for reads more important than for writes.

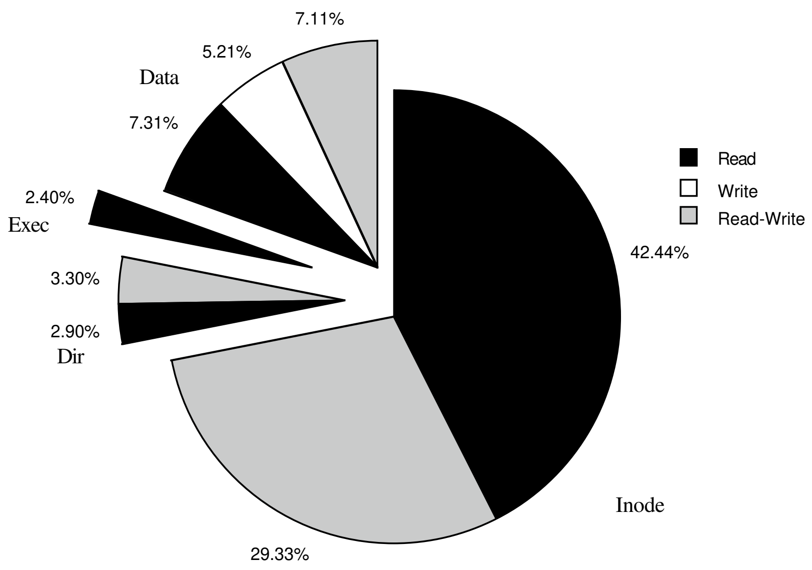


Figure 3.6: Relative quantity of read, write, and read-write files.

3.2.3 Read Files, Write Files, and Read and Write Files

Individual objects fall into three classes. Those that are only read during a workload, those that are only written and those that are both read and written. Separating these three classes helps evaluate the significance of a particular write policy decision on overall performance. It is also necessary to understand the potential interaction between read and writes.

Figure 3.6 shows the breakdown of read, write, and read-write files averaged for the workloads. This figure breaks down the unique files accessed by the workloads and is a static measure. The contribution of each type is the same as shown in Figure 3.1. Figure 3.6 shows how that contribution breaks down into read, write, and read-write files.

Inodes and directories are not written without being also read. Workloads write to inodes and directories as either read-modify-write or modify-and-use. Many inodes and directories merely get read. Executable files are only read, and datafiles have all three classes of file access. For the majority of workloads the write-only files comprise the smallest class of datafile in terms of size and number. Read-only makes up the largest class. The one exception, CAD Chip Build, has write-only as the largest group, followed by read-only and lastly read-and-write. This results from the nature of the problem being solved in the CAD workload. The application uses a relatively small description of the chip to create the layout and routing for the chip. It then writes the much larger created chip

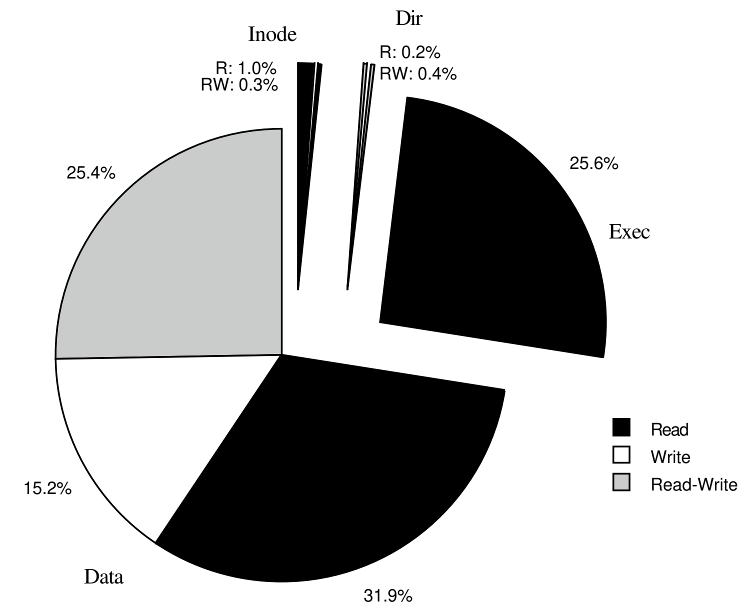


Figure 3.7: Relative static size of read, write, and read-write files.

description for use at a later time by another workload.

The prevalence of files that are both read and written is important for a cache policy. Creating a cache policy, however, requires more information than just knowing that many files are both read and written. These read-write files have either a read-modify-write pattern or a write and later read. If the data is written and later reread, then the cache should try to save the write data for subsequent reading. If data is read and then written but not reread, writing all the way through and flushing the cache of written data would be appropriate. Both types of patterns exist. Section 4.1.3 shows the impact of write data on cache performance. For a large percentage of the files, the write cache policy directly impacts the read policy, or at least the read performance of the cache.

Figure 3.7 shows the average static size of read, write, and read-write files for each workload. The relative contributions of read, write, and read-write components approximate those based on unique files, suggesting no file sensitivity in the read-write distribution. The predominance of read-only files, both by count and size, indicates the significance of the cache to performance. Read policy has a greater impact than write policy.

Maximum Size Category	1k	4k	8k	16k	32k	64k	256k	512k	1M	4M	GT 4M
Files	6.0	3.2	2.7	17.5	19.6	9.4	45.6	10.2	4.5	3.4	0.0
Uses	157.9	89.8	405.9	226.4	516.7	1040.7	1283.7	436.9	158.8	8.5	0.0
Reuse	26.3	28.1	150.3	12.9	26.4	110.7	28.2	42.8	35.3	2.5	0.0

Table 3.5: Executable File Size and Usage by Size Distribution (Average)

Maximum Size Category	1k	4k	8k	16k	32k	64k	256k	512k	1M	4M	GT 4M
Files	924.5	265.5	87.9	73.7	51.4	41.7	108.3	7.0	5.2	7.5	1.5
Uses	118952.4	5891.1	851.2	791.1	174.1	197.5	3736.6	38.5	2509.5	178.8	5.2
Reuse	128.7	22.2	9.7	10.7	3.4	4.7	34.5	5.5	482.6	23.8	3.5

Table 3.6: Datafile File Size and Usage by Size Distribution (Average)

3.3 I/O Size Distributions

3.3.1 File Size and Use Distributions

Tables 3.5 and 3.6 show the distribution of executables and datafiles by file size and usage, averaged for the eleven workloads. Each column label denotes the maximum sized file in the category. The files in each category are larger than the previous column. For example, the "32k" column includes all files greater than 16 Kbytes and less than or equal to 32 Kbytes. The distribution of file sizes is a static measure based on the unique files touched. For example, Table 3.5 shows that the average workload touched 6.0 executable files of size 1 Kbyte or smaller.

File usage measures the number of times a workload opens a file, reflecting uses of the files rather than actual reads or writes. Applications may open a file without reading or writing any data, may access part or all of the file, or may use the file multiple times before closing it. Table 3.5 shows that, in an average workload, the 6.0 executable files of size 1 Kbyte or smaller were opened a total of 157.9 times, indicating that the files were reused an average of 26.3 times each. In this context *reuse* is the number of opens/file. These results give an indication of the amount file sharing that occurs between processes and the most likely size of shared files.

As expected, executable files are reasonably large, clustering in the 16–256 Kbyte range. The majority of the executable uses for each workload, 54–85%, come from executables in the 16–32K, 32–64K, and 64–256K categories. The reuse of executables has no correlation to either the static size or use distributions. A few actively used executables dominate the high reuse ratios for executables. These are very workload dependent. Because there are so few executables in each category, no single category dominates the reuse.

A large percent of the datafiles are smaller than 1 Kbyte and, except for one workload, more

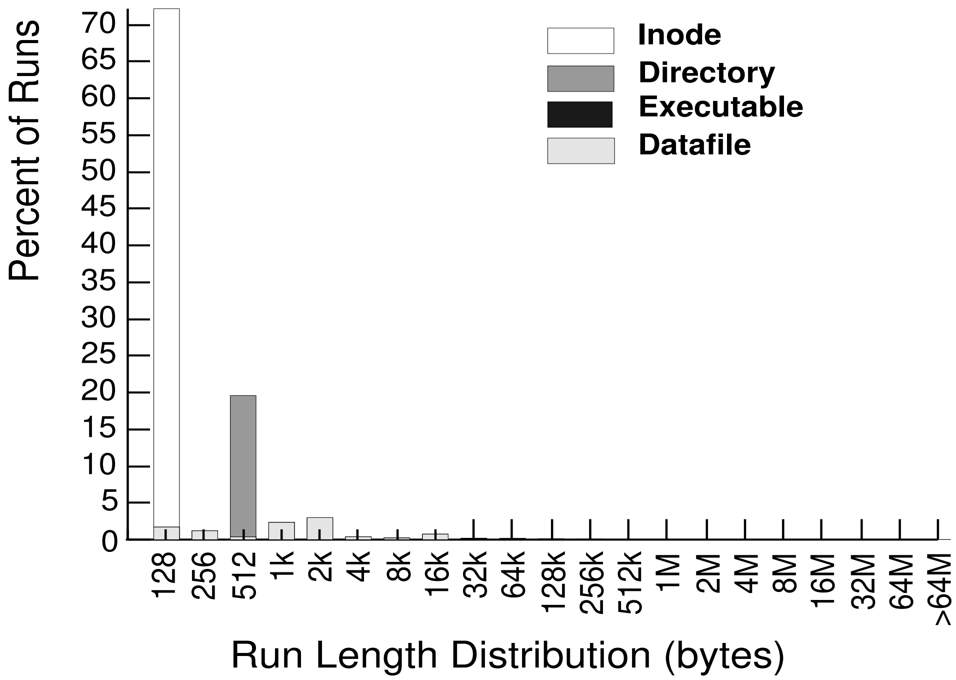


Figure 3.8: Distribution of run lengths.

than 50% of the datafiles fall in the 1K–4K category. Consistently high reuse ratios occur for datafiles less than 1 Kbytes. The 1k and 4k categories account for an average of 87% of the datafile uses. Surprisingly high reuse ratios occur for files in the 512K-1M and 1M-4M categories. These categories have few unique datafiles, typically fewer than ten, but they are frequently opened.

3.3.2 Dynamic Run Length and Request Length

A run is a sequential access of a file or a portion of a file. The run length is the number of sequentially accessed bytes. The run length for read accesses must be no larger than the file size, but a write may extend the file size.

Figure 3.8 shows the distribution of run lengths broken down by type, and Figure 3.9 shows the distribution of bytes transferred for these runs. The x-axis shows the maximum run size for each category. The distribution of run lengths tails off quickly. Fewer than 1% of the runs exceed 16 Kbytes. As Figure 3.9 shows, the number of bytes transferred by large runs does not tail off. In fact, half of the bytes transferred occur in sequential runs of greater than 64 Kbytes and a quarter of all bytes transferred are in runs of more than 256 Kbytes. Thus, even though large sequential runs do not make up a significant fraction of the total file access in terms of actual runs, most of the bytes transferred by the workload occur in these large sequential runs. This type of behavior has

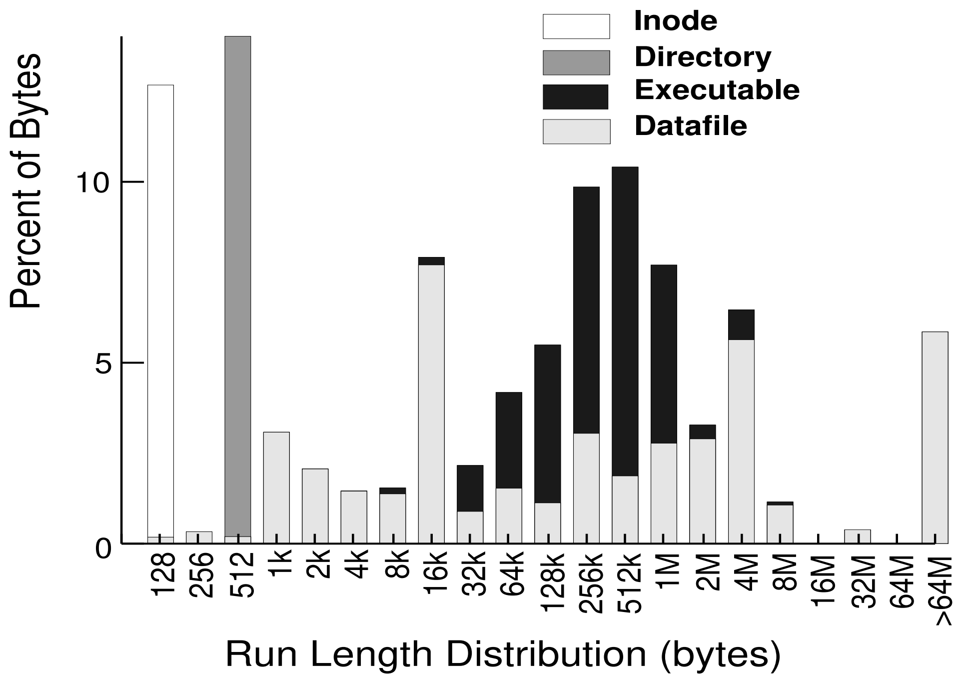


Figure 3.9: Distribution of bytes transferred by run length.

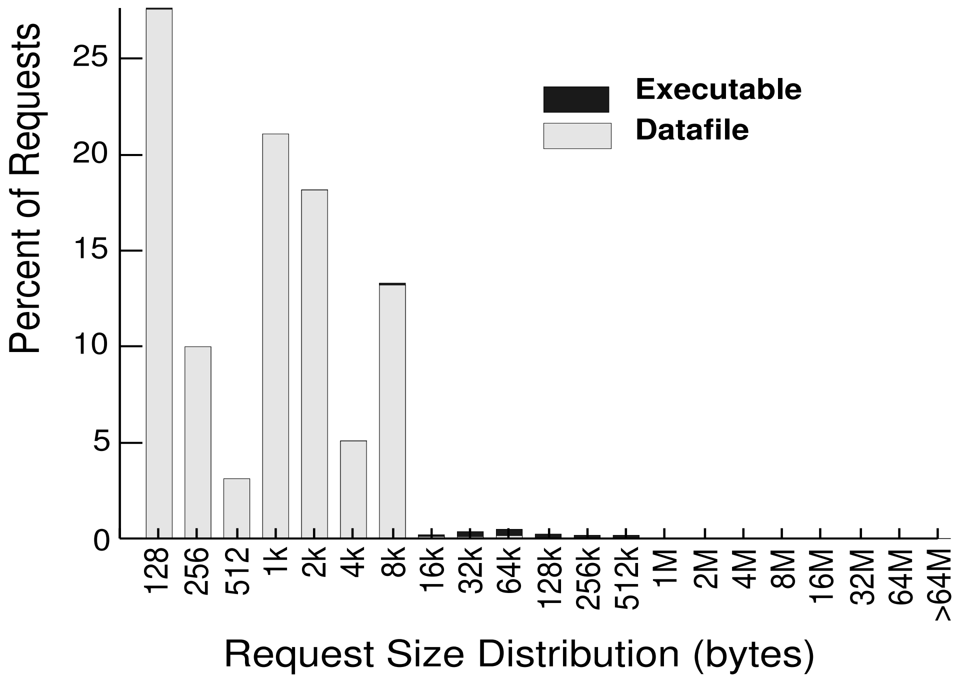


Figure 3.10: Distribution of request sizes for datafiles and executables.

been measured previously [ODCH+85, BHK+91].

Figure 3.10 shows the distribution of requests for datafiles and executables. Since executables are modeled as a single request for the entire executable, they generate large requests. Most of the datafile requests are for at most 8 Kbyte blocks regardless of the file size or run length. Large sequential runs thus generate many requests.

Since many smaller requests produce large runs, requests to these runs have a considerable sequential locality that can be exploited with a cache policy. Transferring the runs in a few larger blocks can reduce I/O cache misses, disk overhead and the time the disk spends servicing requests.

However, applications use large sequential runs infrequently, so trying to keep them around can pollute the cache with data unlikely to be re-referenced, and can evict many smaller objects that will be re-referenced. Medium to large caches that hold much, if not all, of the working set show the greatest effects of both cache pollution and sequential locality.

Chapter 4

I/O Characteristics in the Presence of a Cache

An I/O cache can capture locality more efficiently if its designer understands the sources of locality and the cache properties that best capture that locality. This chapter examines the cache behavior properties of various components of the I/O workload. The behavior differs for each component. The locality might be sensitive to cache block size or total cache size, but is often sensitive to both. Different components require different block and cache sizes to effectively capture locality.

4.1 Cache Behavior I/O by File Type

If all I/O requests are cached without regard to file type, few options exist for reducing the cache misses and improving cache performance. Figure 4.1 shows the cache miss behavior of a typical workload in a unified cache that uses no information about request types. Due to the overwhelming number of small requests from inodes and directories, smaller block size choices always win. Improving the I/O cache performance requires more information about the statistical properties of the workload and the type of locality that can be captured.

Individually measuring the cache behavior of each of the four different file types—inode, directory, executable, and datafile—helps evaluate the usefulness of type information for an I/O cache. Differences can potentially be exploited to improve the cache behavior of the entire workload. The cache behavior of each type, along with the system utilization of each type, determine system I/O performance.

The hit ratio is a simple means to compare and understand the way individual file types use the

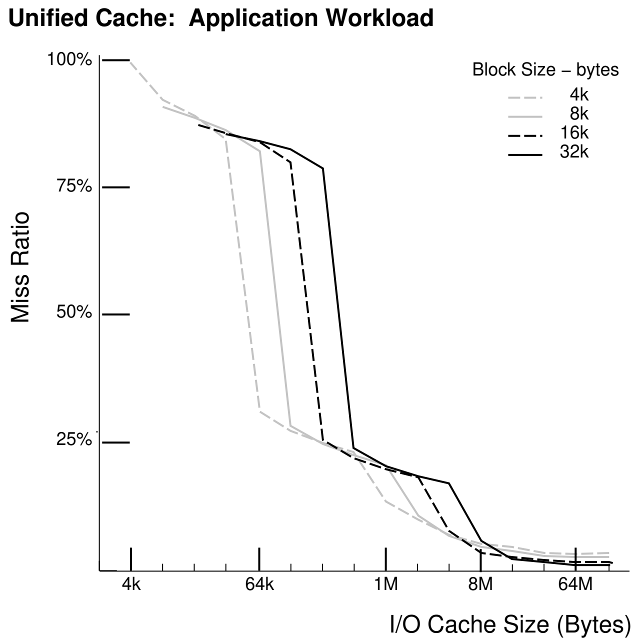


Figure 4.1: Typical workload behavior in a unified cache.

cache. The hit ratio for a given cache is defined as the number of references that hit in the cache divided by the total number of references. The hit ratio can adequately compare schemes when the total number of references remains constant for each scheme. Each workload has some set of highly reused files. Once the LRU cache captures the data requests to these files, the request misses drop dramatically. This is referred to *working set capture*. Low miss ratios, higher hit ratios, and smaller working sets all mean better cache performance.

The rest of this section uses hit ratios to describe the cache behavior of each file type across a range of cache sizes. The cache size for the inodes is measured in *number of entries*, while the others are measured in bytes. All the caches presented use a fully associative LRU replacement policy, and use a copy-back write policy.

4.1.1 Inode and Directory Cache Behavior

Inodes

Figure 4.2 shows that, among the eleven subject workloads, inodes have a well defined working set size of about 128 entries, and that the cache behavior for inodes is virtually independent of workload.

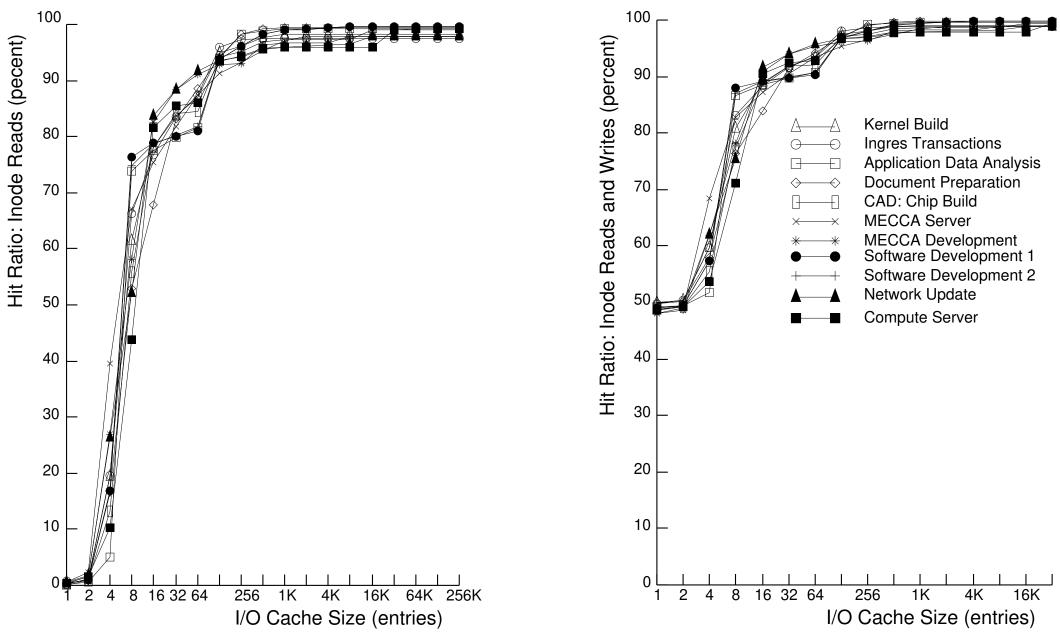


Figure 4.2: Inode references in a fully associative I/O cache.

Sixteen inode cache entries suffice to capture more than 75% of the inode read references, and 128 entries captures the inode working set for all the benchmarks.

One to two entries capture no locality, indicating that the workloads use inodes in groups. This behavior is a direct result of the way the operating system uses inodes to open a file or to check the file status. Typically, it must access several inodes for a single open or status check. For example, each of the four elements in the path name `/user/kjr/thesis/ch1.tex` has an associated inode that must be read to locate the next directory or file in the path name. Often, applications or workloads use many files with similar path names, such as `/user/kjr/thesis/fig1.ps` and `/user/kjr/slides/talk.ps`. The inode working set thus includes the inodes for several paths and many files. The three example path names in this paragraph have a total of 12 elements but include only seven unique inodes.

The cache performance of inodes is particularly important for several reasons. Inode read references make up a large percentage of the total requests. Also, inode reads often form part of an indirect data access sequence. Each inode contains the disk block information required to access the subsequent directory or file in a path name.

Including writes in an inode subcache halves the initial miss ratio, as seen in Figure 4.2. Most inode writes merely *update* file access times and do not modify the file system structure. Each of these updates essentially forms a read-modify-write pattern, such that the write part always hits in the cache. The size of the cache determines whether or not these writes get reused before being written

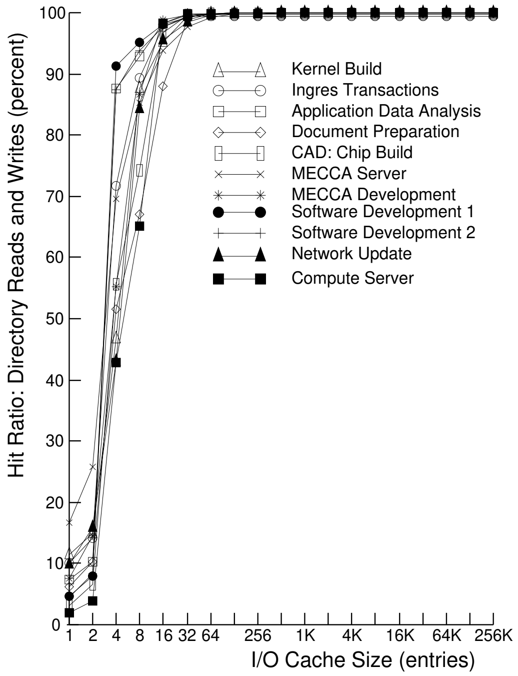


Figure 4.3: Directory references in a fully associative I/O cache.

back to the disk.

Directories

Directories behave almost exactly like inodes. Figure 4.3 shows the cache behavior of both read and write directory requests. Read behavior dominates, however; writes make up less than 1.0% of all directory requests. At one to two directory entries, the hit ratio of 5-10% indicates some individual reuse of directories. Often, accessing a file with a relative filename requires only one directory lookup. Subsequent accesses to other files in the same directory then reuse the same cache entry. Thirty-two entries suffice to capture the working set of all the workloads, with an average individual hit ratio of 99%.

Inodes and Directories

Inodes and directories have similar cache behavior. Including both in the same subcache produces uniform behavior across all workloads. Figure 4.4 shows the read hit ratio and the total hit ratio for inodes and directories together in a cache with 128-byte blocks. In this cache, a single 512-byte

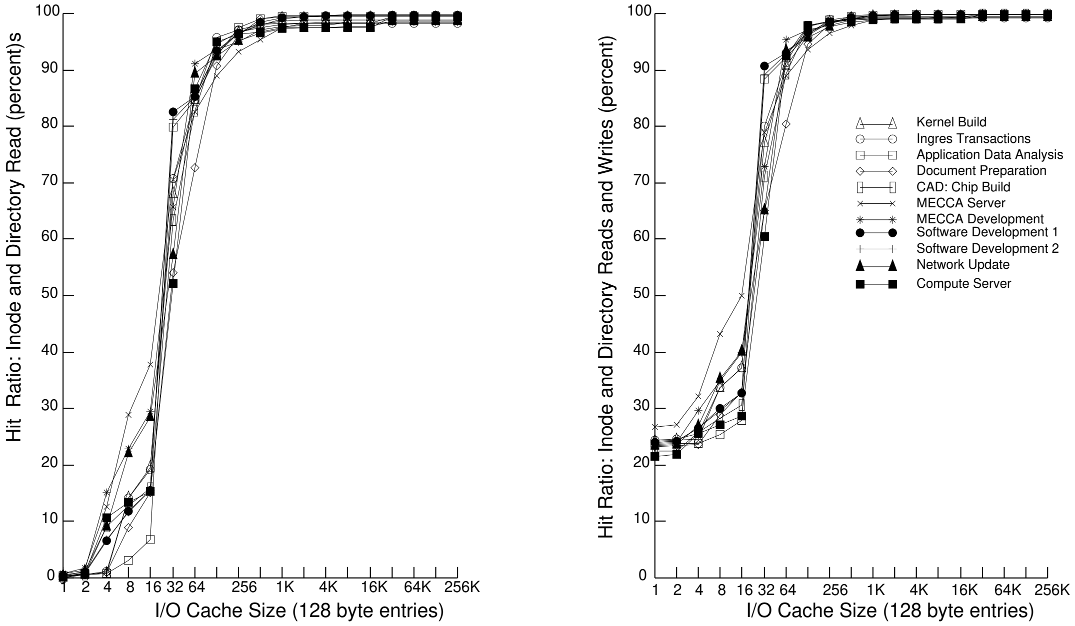


Figure 4.4: Inode and Directory references in a fully associative I/O cache.

directory entry occupies four blocks. Ideally, the inode and directory working sets would not conflict in the cache, and the combination would yield the same hit ratio with less space than if each had a separate cache. For cache sizes greater than the directory working set size of sixteen directory entries, or sixty-four 128-byte blocks, the combined subcache gets a higher hit ratio than two individual subcaches. Because of the relationship between inodes and directories, workloads often require many inodes and directories at the same time. For caches smaller than the minimal working set capture size of about 32 inode entries or sixteen directory entries, the two conflict, competing for cache space. The resulting read hit ratio is lower for the combined subcache than it would be for two separate half-sized subcaches. 256 entries suffice to capture the inode and directory working set completely and eliminate competition. This requires only 32 Kbytes of cache.

4.1.2 Executable Cache Behavior

Figure 4.5 shows the cache behavior of the executable files. The various workloads exhibit surprisingly similar behavior. The executable subcache needs 4–8 Mbytes to capture a significant fraction of the executable references. Most of the workloads have a well defined working set capture size. This comes from the fact that many of the workloads use a fixed set of executables in cyclic fashion. This is a prime characteristic of most current compilers, and many of the workloads are compilation intensive. The compiler consists of several passes, each one a different executable. The C compiler used to build the ULTRIX kernel in the *Kernel Build* workload executes six passes

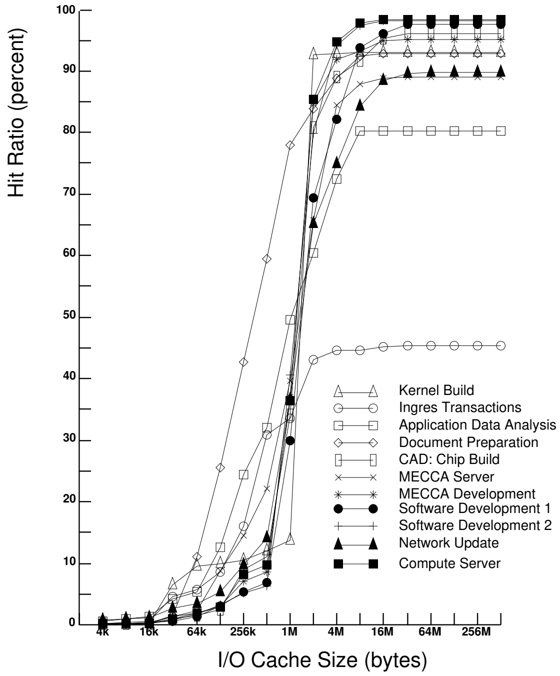


Figure 4.5: Executable references in a fully associative I/O subcache with 4-Kbyte blocks.

to compile each module. *Kernel Build* reuses the executables from the six passes sequentially in a loop. Until the whole group fits in the cache, each executable gets expelled from the cache before it can be reused. Thus, the *Kernel Build* curve has a very sharp knee. The other compilation intensive workloads are *CAD, MECCA Development, Software Development 1* and *2*, and *CPU Server*.

The *Document Preparation* and *Application Data Analysis* workloads reuse a core set of executables in cyclic fashion. This core constrains itself to neither a small set nor a fixed order, so the cache can capture some of the working set even before the whole set fits.

Only the *Ingres* workload fails to reuse its executables frequently. Two-thirds of the executable files are used only once or twice. This behavior probably typifies workloads that perform pure transaction processing. The workload initially executes many processes to start up the database. Once the system starts running, however, the single database executable performs all processing, and no new processes need executables. The *Network Update* and *MECCA Server* workloads have a large data base component, but the results of the database queries lead to subsequent execution.

Improving Executable Cache Performance: Demand Paging

The actual performance impact of the executable part of I/O depends heavily on how the system fetches and manages executables. The model used for the simulations in Figure 4.5 assumes that the system requested the entire executable at the time of an *exec* system call. Often, however, the virtual memory system pages an executable in on demand. Since executables tend to be large, the performance implications of pure demand paging can be quite significant.

In general, the disk time trade-off for demand paging depends on the overhead of a disk access and the amount of data transferred. Equation 4.2 shows what fraction of the executable can be used before demand paging becomes more expensive than simply fetching the entire executable all at once.

O	=	average overhead per disk access (seek, rotational latency and SCSI bus time) in ms
B	=	bytes to transfer in executable
T_x	=	disk transfer rate (bytes/ms)
X	=	fraction of the workload transferred
P_{sz}	=	page size in bytes
P	=	pages per executable in the workload

$$O + \frac{B}{T_x} = X * \left[O + \frac{P_{sz}}{T_x}\right] * P \tag{4.1}$$

$$X = \frac{O + \frac{B}{T_x}}{\left[O + \frac{P_{sz}}{T_x}\right] * P} \tag{4.2}$$

Typical values taken from Rue [RW94b]:

O	=	18.8 ms
T_x	=	2 Kbytes/ms
P_{sz}	=	4 Kbytes

Average executable numbers for the workloads

$$B = 135 \text{ Kbytes} \quad P = 34 \text{ pages} \quad \Rightarrow X = 12\%$$

Transferring the whole executable to the I/O cache on an *exec* call and then demand paging it into the virtual memory space efficiently transfers the executable from disk and prevents the unused executable pages from filling up the virtual memory system space. It is a waste of space for executable pages to reside in both the cache and the virtual memory system. To avoid this duplication executable pages can be transferred between the virtual memory system and the I/O cache [NWO87]. When a process actively uses an executable, the executable's most-used pages naturally wind up in the virtual memory system. Once all the processes using an executable have terminated, the executable pages can be either transferred to the cache or left in the virtual memory system until another process needs to reuse them. Measurements of the SPRITE operating system indicate that the amount of time a page remains in the virtual memory system is comparable to the amount of time it remains in the cache [NWO87, BHK+91]. The virtual memory system caches the executable pages used by the last execution of the program. The I/O cache only needs to fetch the whole executable when the executable no longer resides in the virtual memory system.

The boundary between the I/O cache and the virtual memory system is becoming less clear. Many systems have dynamically moving boundaries. If the operating system caches executables in virtual memory, this can affect the general space requirements and thus impact the entire system memory needs. Those using virtual memory as a cache must acknowledge its I/O cache space requirements. Moving the executables out of the I/O cache does not invalidate the need to understand their cache properties.

4.1.3 Datafile Cache Behavior

The read behavior of datafiles differs considerably from the write behavior. Read system calls tend to request blocks of data. These blocks tend to have a fixed size, often 8 Kbytes, which is the standard request size for C libraries used with ULTRIX system compilers. Each write system call tends to transfer only a small number of bytes. Writes tend to have more sequential locality than reads [Red92a, RF93]. Evaluating the cache behavior of reads and writes separately shows differences that can help improve the read hit ratio or the performance of a write policy.

The important read behavior is not the number of blocks that hit or miss in the cache, but rather the number of requests that miss in the cache. If the cache contains all the data for a given read request, the request *hits,* and does not access the disk. If the cache is missing any part of the requested data, the request *misses,* initiating a single disk access to retrieve the remainder of the data.

The number of requests depends only on the workload, not on the cache configuration and cache block size. If the cache can hold the entire request, a miss will result in a single disk access. Since the cache model requires all data requests to pass through the cache, a miss will require multiple disk accesses if the request size exceeds the cache size. A cache smaller than the average request size results in more misses than requests. This accurately reflects disk behavior, and provides better insight than either the request hits or the cache block hit/miss ratios.

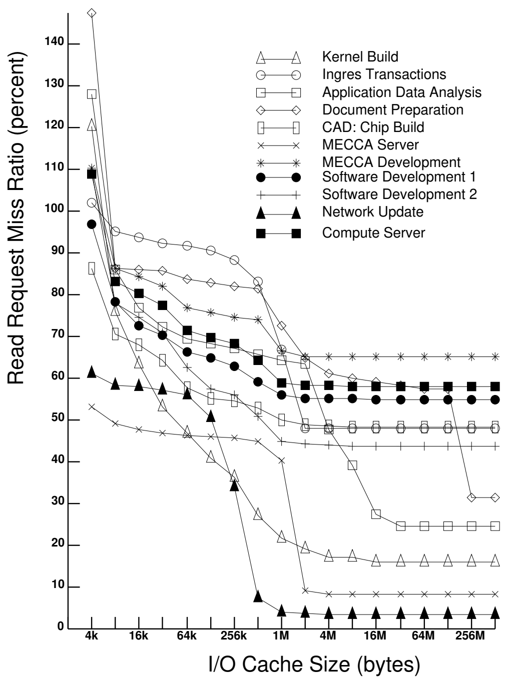


Figure 4.6: Read request miss ratio for a write-invalidate datafile cache with 4-Kbyte blocks.

$$\text{Request Hit Ratio} = \frac{\text{hit requests}}{\text{total requests}}$$

$$\text{Request Miss Ratio} = \frac{\text{disk requests generated}}{\text{total requests}}$$

The *request miss ratio (RMR)* measures the number of disk accesses independently of the cache block size or cache configuration. A lower RMR results in fewer disk accesses. The RMR thus allows direct comparison of cache schemes. The *read request miss ratio (RRMR)* and the *write request miss ratio (WRMR)* measure the ratio of read or write requests that miss in the cache. Very small caches have a high RMR because many requests do not fit in the cache and generate multiple misses. This is especially true for the 4-Kbyte cache, where most of the workloads end up with a RMR greater than 100%.

The No-Write-Allocate Cache

Figure 4.6 shows the request miss ratio for reads in a datafile I/O subcache that invalidates on write. Invalidate on write removes from the cache any cache blocks that match the write request. The invalid blocks move to the end of the LRU list, making room for new read data.

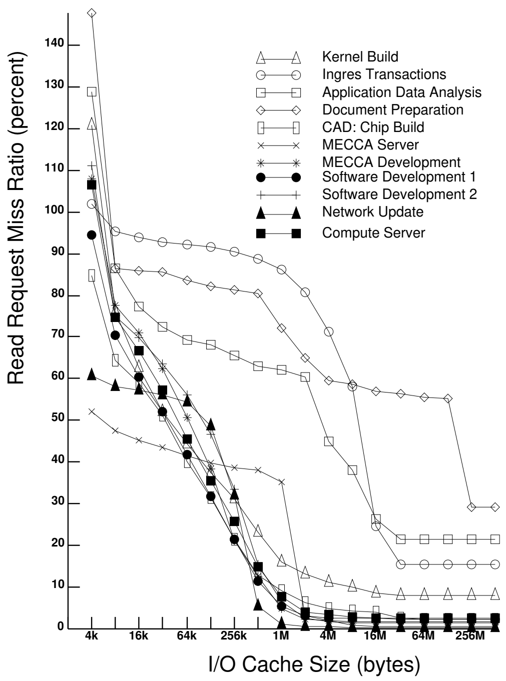


Figure 4.7: Read request miss ratio for a write-allocate datafile cache with 4-Kbyte blocks.

Below the working set capture size, doubling the cache size typically reduces the read request miss ratio (RRMR) by only a couple of percent. For the compilation intensive workloads *Kernel Build, CAD, MECCA Development, Software Development 1* and *2*, and *CPU Server*, capturing the working set reduces the RRMR, but their final RRMR is still high. The *Kernel Build* workload has a lower RRMR because of the large percentage of include files (`*.h`) involved in the compilation. The infinite-cache RRMR for the compilation intensive workloads is about fifty percent. The other workloads generally have a much lower infinite-cache RRMR.

The Write-Allocate Cache

Allocating write data to the cache reduces the amount of cache space available for read data, but subsequent reads of the allocated write data reduce the read misses. If data is written merely for storage, or the gap between a write and a subsequent read exceeds the cache size, allocating data will not reduce the read misses. If allocated writes push out useful read-only data, the read misses will increase. Figure 4.7 shows that allocating writes to the cache dramatically improves the read request miss ratio. Every workload has a lower final RRMR.

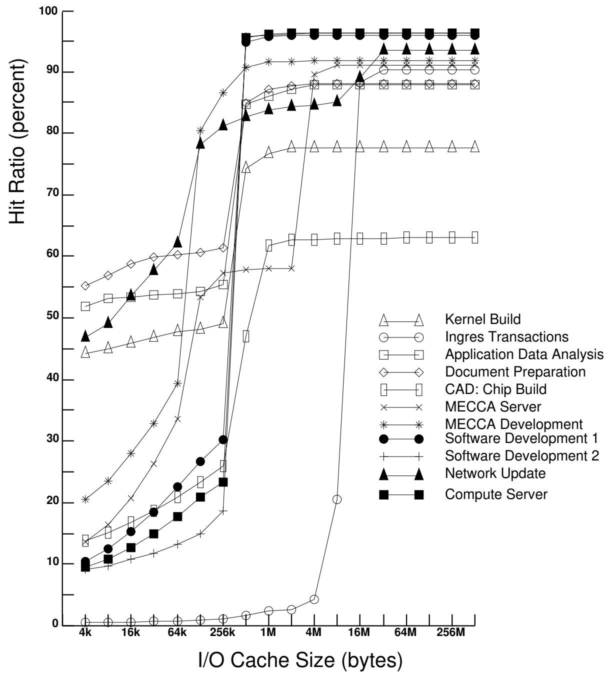


Figure 4.8: Cache hit ratio for writes in a *data-overwrite* cache with 4-Kbyte blocks.

Write-Allocate versus No-Write-Allocate

Allocating write data to the cache improves the read performance substantially. Larger caches capture subsequent reads of newly written data, which is especially common in the compilation intensive workloads. For the compilation intensive workloads, each increase in the cache size continues to reduce the RRMR, up until the point where the cache captures the entire working set. Except for *Ingres*, the non-compilation intensive workloads' cache behavior changes little under write allocation.

The Data-Overwrite Cache

Figure 4.8 shows the cache behavior of datafile write requests in a data-overwrite cache. Here, a write hit overwrites the existing data. If the overwritten data resides only in the cache, then the overwrite eliminates the need to write it back to disk. Only write hits that delete previously written data can reduce the number of bytes written to disk. Coalescing multiple write requests into a single request can reduce the disk overhead.

Writes exhibit distinct working set capture behavior. The compilation intensive workloads have a single capture knee. Some of the other workloads have two knees, because an intermediate cache size captures a subset of the total working set. A smaller cache suffices to capture the write working set, typically about one-fourth the size needed for reads. Having captured the working set, however, the hit ratio for writes tends to be lower than that for reads.

Write hits fall into one of two categories. The first, the *sequential block hit,* accumulates data into an already-allocated cache block. Subsequent partial writes hit in the same block. Here, the cache acts like a buffer, accumulating small sequential writes into large blocks. This results in fewer disk accesses to write the same amount of data to the disk. Sequential block hits form the majority of cache hits in very small caches. In Figure 4.8, the *Ingres* workload has almost no sequential block hits, but the *Kernel Build, Application Data Analysis, Document Preparation* and *Network Update* workloads have almost a 50% sequential block hit ratio. Increasing the block size should increase the sequential block hits.

The second type of write hit overwrites existing data. A file or block may be written, then read, and then written again. The original write data need never go to disk, provided the cache gives enough data security. Increasing the cache size increases the amount of rewriting that can occur before the data leaves the cache. Baker et al. observed similar behavior [BHK+91]. For most of the workloads, a one-megabyte cache captured the rewrites. The database workloads had a much larger rewrite working set, because they rewrite large log files. The high hit ratio for writes means that workloads typically rewrite data many times.

4.2 Impact of Block Size on Cache Behavior

The block size determines the amount of sequential locality a cache can capture. Large cache blocks capture more sequential locality, eliminating extra disk requests for long runs of adjacent data, and reducing the transfer overhead per byte. This improves the cache performance if the workload contains a high proportion of large sequentially-accessed files. If the workload accesses many small files, or small portions of files, large blocks will reduce the cache performance. Blocks much larger than the file size use cache space inefficiently; much of the block remains empty or holds excess data that never gets used. This effectively reduces the amount of data the cache can hold. Transferring larger blocks does not reduce the overhead per byte if the workload never uses the extra data. In fact, transferring unnecessary data increases the transfer time.

Ideally, the cache would have a variable block size, but true variable block size caches have implementation and performance problems. The cache can approximate variable size blocks, however, by fetching multiple small blocks. The small blocks cannot be arbitrary small. Each request for an object much larger than the block size must locate multiple blocks in the cache, slowing down the cache lookup. Also, the cache would contain many more blocks. Searching through them would further slow down the lookup. Lastly, since each block contains tag information about the data, making the block too small causes the tag information to become a significant fraction of the total cache space.

Each of the four file types has a different average object size and size distribution, suggesting the need for different block sizes. Inodes are a fixed size: 128 bytes for current versions of ULTRIX. The typical directory is small and has a fixed size—512 bytes for ULTRIX. Larger directories are an integral multiple of the small directory size. Executables are much larger than inodes or directories, averaging 135 Kbytes in size for the eleven workloads studied. Datafiles have an average size somewhat larger than executables, 160 Kbytes, but have a much broader distribution of file sizes. Most of the datafiles are less than 10 Kbytes, but the majority of the bytes used or generated reside in large sequentially accessed files. The 160 Kbyte average mainly represents a weighted midpoint between small and large files. These different file size characteristics mean that no single block size can be the best choice for all of the file types.

4.2.1 Block Size Effect on Inode Locality Capture

The best block size for inodes takes into account the temporal locality of inode references, the spatial locality between inodes on disk, and the disk sector size. Temporal locality is the most crucial aspect of inodes, because of their high reuse rate. Figure 4.2 showed the temporal locality capture of an inode cache. Capturing 90% of the inode references requires at least 128 entries. For each of the workloads, 512 entries captures at least 95% of the inode references. The cache captures this temporal locality most efficiently when the block size equals the inode entry size.

A cache can potentially reduce the number of compulsory misses by taking advantage of spatial locality between inodes, although the high reuse rate of inodes makes spatial locality much less of a factor than the temporal locality. A cache with 512 inode entries reuses the average inode about fifty times, producing a 98% hit ratio. Ideal spatial locality would increase the hit ratio by

$$\Delta\text{Hit Ratio} \quad = \quad \text{Miss Ratio} \times \left(1 - \frac{1}{\text{\# entries per block}}\right).$$

Non-ideal spatial locality, where all the cache entries do not always belong to the LRU set, may reduce the temporal locality capture. However, increasing the block size should increase the overall inode hit ratio, particularly while the number of blocks exceeds the 1024 entries required to capture the temporal locality working set completely.

Figure 4.9 compares the inode hit ratio for 128-byte blocks to that of 4-Kbyte blocks with varying degrees of locality. To evaluate locality, each inode is assigned a specific block, because the workload traces do not contain disk placement information. Figure 4.9 assigns inodes to 4-Kbyte blocks based on the initial dynamic reference pattern. For a locality of four entries/ block, the first four unique inodes referenced by the workload are placed in the same block, the next four unique references are placed in a different block, and so on. The remainder of each block supposedly holds

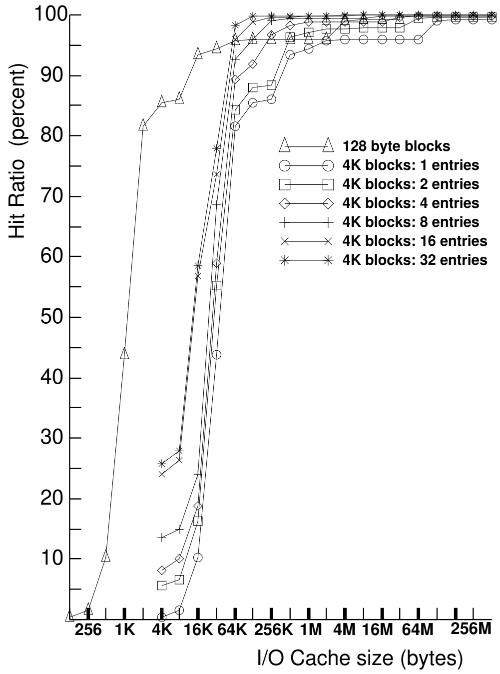


Figure 4.9: Hit ratio for inode reads in the *CPU Server* workload.

inodes not used by the workload. This simulated assignment shows the general impact of inode locality on the inode cache performance[1]. For small caches with large blocks, the density of inodes within the cache greatly affects the hit ratio. But the spatial locality of 4-Kbyte blocks is obviously much less important than the temporal locality of 128-byte blocks [MJLF84].

Figure 4.9 highlights the importance of cache utilization. Even the scheme that packs each 4-Kbyte block with 32 consecutive inodes performs poorly until more than 90 percent of the inode data fits in the cache. The poor performance of the assignment also shows that the workloads use a variety of different access patterns for inodes, and that the set of inodes actively used changes over time. The assignment scheme uses the temporal locality of the first accesses to spatially group the inodes. This would result in good performance if the workload always used inodes in identical groups, or runs. Instead, it appears that inodes are not reused in the same groupings, making it unlikely that any given placement might improve locality. Attempting to capture spatial locality of inode references increases the cache size required to capture inode reads.

[1]Actual inode placement favors files within a directory rather than search path locality.

4.2.2 Block Size Effect on Directory Locality Capture

Block size affects the hit ratio for directory references in the same way as inode references. Directory entries are highly reused. Directories in ULTRIX are four times larger than inodes, and the workloads touch fewer unique directories than unique inodes, meaning much less potential spatial locality benefit from directories than from inodes.

4.2.3 Block Size Effect on Datafile and Executable Locality Capture

Block size has essentially no impact on the individual cache performance of executables. Each use of an executable produces a single I/O read request for the entire executable. If the cache size is smaller than the executable, the operating system breaks the request into smaller requests that fit in the cache. This model thus has no provision for spatial locality. The block size only affects the storage management and access granularity. The block size determines how much cache space is unused when the executable does not fit into a discrete number of blocks. For typical I/O block sizes, unused cache space averages about one-half block per executable. If the block size exceeds the executable size, then the average unused space can be much more than one-half block, because executable sizes have a somewhat bimodal distribution, and the small executables would not, on average, occupy one-half block. There is generally little reason to use large blocks for executables.

Datafiles have a broad distribution of file sizes, reuse rates and access patterns. The average request patterns of a workload determine the best block size choice. The request pattern varies considerably among workloads. In general, most requests access data from small, highly-reused datafiles, while most of the data actually transferred comes from large, sequentially-accessed datafiles. The cache must capture both the temporal locality of small datafiles and the sequential locality of large datafiles.

Figure 4.10 shows the block size effect on cache behavior for datafiles and executable files. The four workloads shown in Figure 4.10 represent the different kinds of behavior found in the workloads. Since block size does not affect the executable cache behavior, the variation comes only from references to datafiles. Caching executables with datafiles eliminates consistency problems, since many executables start out as datafiles generated by compilers, loaders or editors. Executables and datafiles also have similar size distributions, even though there are fewer small executable files than small datafiles. Many implementations transfer the executable blocks to the virtual memory system and remove them from the I/O cache, freeing up more cache space for the datafiles.

The locality in the individual workloads varies, ranging from the primarily temporal locality found in the *CPU Server* workload, to the primarily spatial locality found in *MECCA Server*, to both spatial and temporal in *Application* and *CAD*. In a workload having primarily temporal locality, increasing the block size provides little reduction in the number of request misses and it increases the size of the cache required to capture the working set. However, in a workload having primarily

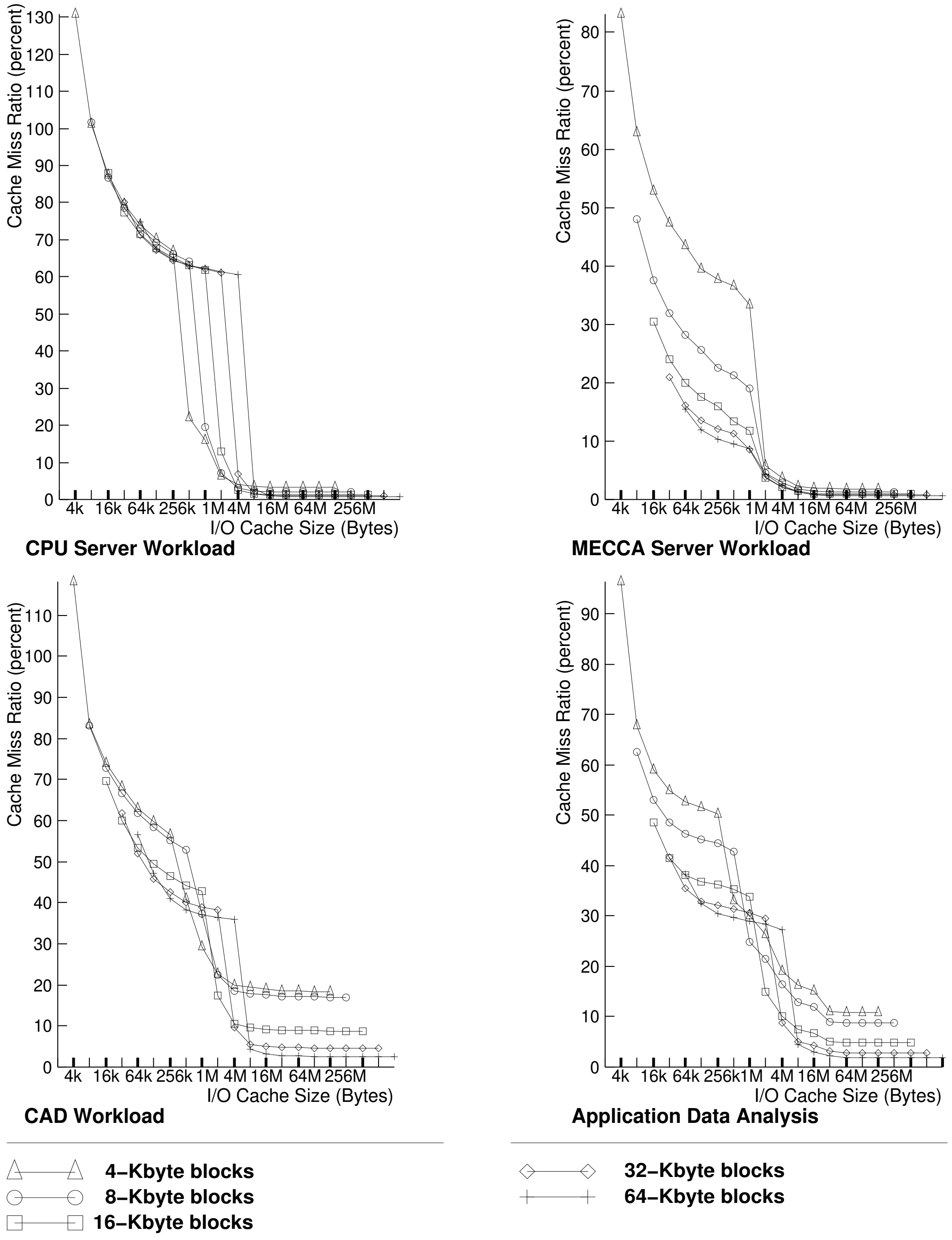


Figure 4.10: Datafile and Executables: miss request ratios showing temporal locality (top left), spatial locality (top right), and mixed locality (bottom row).

spatial locality, increasing the block size significantly reduces the number of request misses, and does not increase the cache required to capture the working set. In a workload exhibiting both spatial and temporal locality, increasing the block size reduces the request misses, but increases the cache size required to capture the working set. For cache sizes between the small and large block working set capture points, the large block size produces substantially worse performance. The best block size choice depends on the I/O cache size, the workload working set size, and the type of workload locality.

For caches smaller than the working set, larger blocks capture the workload's sequential locality. Caches 256 Kbytes or smaller could not capture the working set for any of the measured workloads. These caches are too small to capture the bulk of the temporal locality regardless of block size. Aside from sequential locality, a small cache can capture only the very-local temporal behavior of an individual application, local enough to require only 4 to 8 entries. This very-local temporal behavior produces the sharp decrease in the miss ratio for small caches, which then flattens off as the cache size increases, until the whole working set is captured.

In a cache considerably larger than the working set, large blocks capture spatial as well as temporal locality. The temporal locality capture is dominated by the number of independent small objects that fit in the cache. Increasing the number of blocks beyond that needed to store all the independent objects in the working set does little to improve the temporal locality capture, and nothing to improve spatial locality capture. Beyond this point, increasing the block size increases the spatial locality capture. For the measured workloads, an 8-Mbyte cache using 4-Kbyte blocks always captured the working set. Only a large cache can safely capture the working set. Typically, today's workstation caches are neither small nor large by the above definition. This means that the cache size typically falls within the region just above or below working set capture.

4.2.4 Block Size Impact on Traffic

The block size can impact the disk traffic considerably. Figure 4.11 shows the number of bytes transferred from disk for the *Application Data Analysis* workload. The model assumes that each request retrieves whole blocks from disk regardless of the file size. For small caches, the number of bytes transferred grows directly with the block size. However, the cache need not always retrieve whole cache blocks. Many files are small, and when the block extends beyond the end of the file, the extra data does not need to be retrieved from disk. For small caches, this probably limits most of the traffic explosion that would otherwise be caused by fetching larger blocks. Figure 4.11 shows the worst cache traffic generated by this workload.

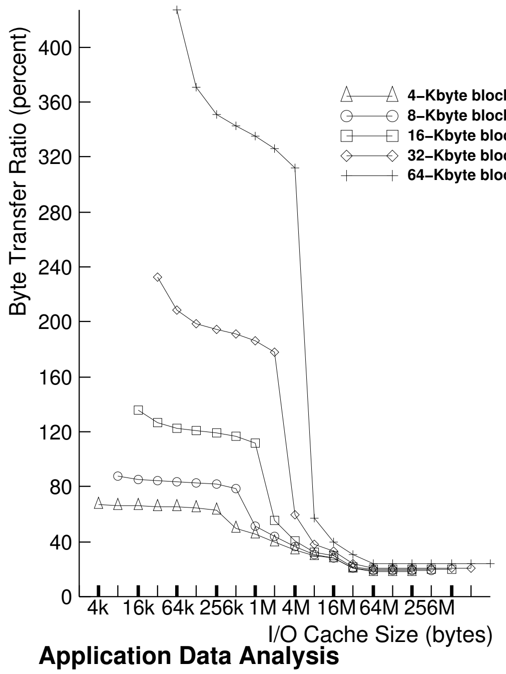


Figure 4.11: Increase in disk traffic from larger block sizes.

4.3 Separating Spatial and Temporal Locality

The statistical breakdown of I/O accesses, such as that described earlier in Section 3.3, provides information about workload locality. Most of the I/O bytes are transferred sequentially to or from large files of more than a half megabyte. Most of the I/O requests, however, access the many small files in a workload. Thus, most of the spatial locality comes from sequentially accessing large files. The small files provide the temporal locality. Assigning different block sizes to the appropriate file sizes could capture both temporal and sequential locality, and improve cache performance.

A Sequential Cache tries to capture the workload's sequential locality by using large blocks and allocating only large files. Requests to sequentially accessed large files constitute a significant fraction of the sequential cache requests. Capturing the reuse of these large files would require considerable cache space, but a cache needs only a small amount of space to capture the sequential access behavior.

A Temporal Cache tries to capture the workload's temporal locality by using small cache blocks for small- and medium-sized, or *moderate* files. The high reuse rate typical of small files contributes greatly to capturing the workload working set.

4.3.1 Sequential Cache Properties

Figures 4.12 and 4.13 show the sequential cache request miss ratio (RMR) of the large files in each of four workloads. The figures show sequential cache behavior for two file size cut-offs. The *512KB cut-off* of Figure 4.12 caches only files of at least 512 Kbytes, and the *128KB cut-off* of Figure 4.13 caches files of at least 128 Kbytes. In each case, the request miss ratio is the fraction of missed datafile and executable requests contributed by the large files, not the fraction of large file requests that miss. The cache behavior varies little between the two cut-offs, even though the load for the two caches differs considerably. Very few files exceed 512 Kbytes. Extending the cut-off down to 128 Kbytes increases both the number of files and the number of requests sent to the sequential cache, thus the *128KB cut-off* sees a greater fraction of the total workload than the *512KB cut-off*.

The workloads exhibit two types of behavior. (1) Sequentially-accessed files allow sequential locality capture. In this case, increasing the block size produces almost ideal reductions in the request miss ratio. Doubling the block size reduces the RMR by almost half. The *Application* workload and the *CAD Chip Build* workload fall into this class. (2) The second type of cache behavior exhibited by some of the workloads shows no sequential locality. In this case, the RMR depends only on cache size, and not block size. The *MECCA Server* workload and the *CPU Server* workload fall into this class.

The *Application* workload and the *CAD Chip Build* workloads exhibit almost ideal sequential locality capture. The workloads sequentially access the large files and do not reuse individual blocks, so increasing the cache size does not capture more locality. The cache cannot capture file reuse until the entire set of files fit in the cache. Doubling the block size from 4 Kbytes to 8 Kbytes produces a much smaller reduction in the number of misses than subsequent doublings because many requests access 8 Kbytes regardless of whether the cache has 4-Kbyte or 8-Kbyte blocks.

A single large block suffices to capture the sequential locality of one active file. The number of blocks needed to capture sequential locality of more than one file equals the maximum number of active files that the cache must maintain. The block size determines the amount of data stored for each file, and thus how long the file block remains active in the cache. Larger blocks stay active for a longer period of time because the workload takes longer to consume the data. If the cache cannot hold all the active files, the cache RMR looks like that of a cache with smaller blocks, because actively-used blocks get expelled.

Some contention for cache space exists between files, and this becomes more pronounced for larger cache blocks. With 32-Kbyte to 128-Kbyte blocks, the workload might not finish reading an entire block before it needs the next block. The cache block then writes over the first block, necessitating a prefetch later to access the unfinished portion. Thus, two half-size blocks perform better than a single large block. The *Application* workload needs only two blocks to eliminate the contention, meaning it has only two active large files at a time.

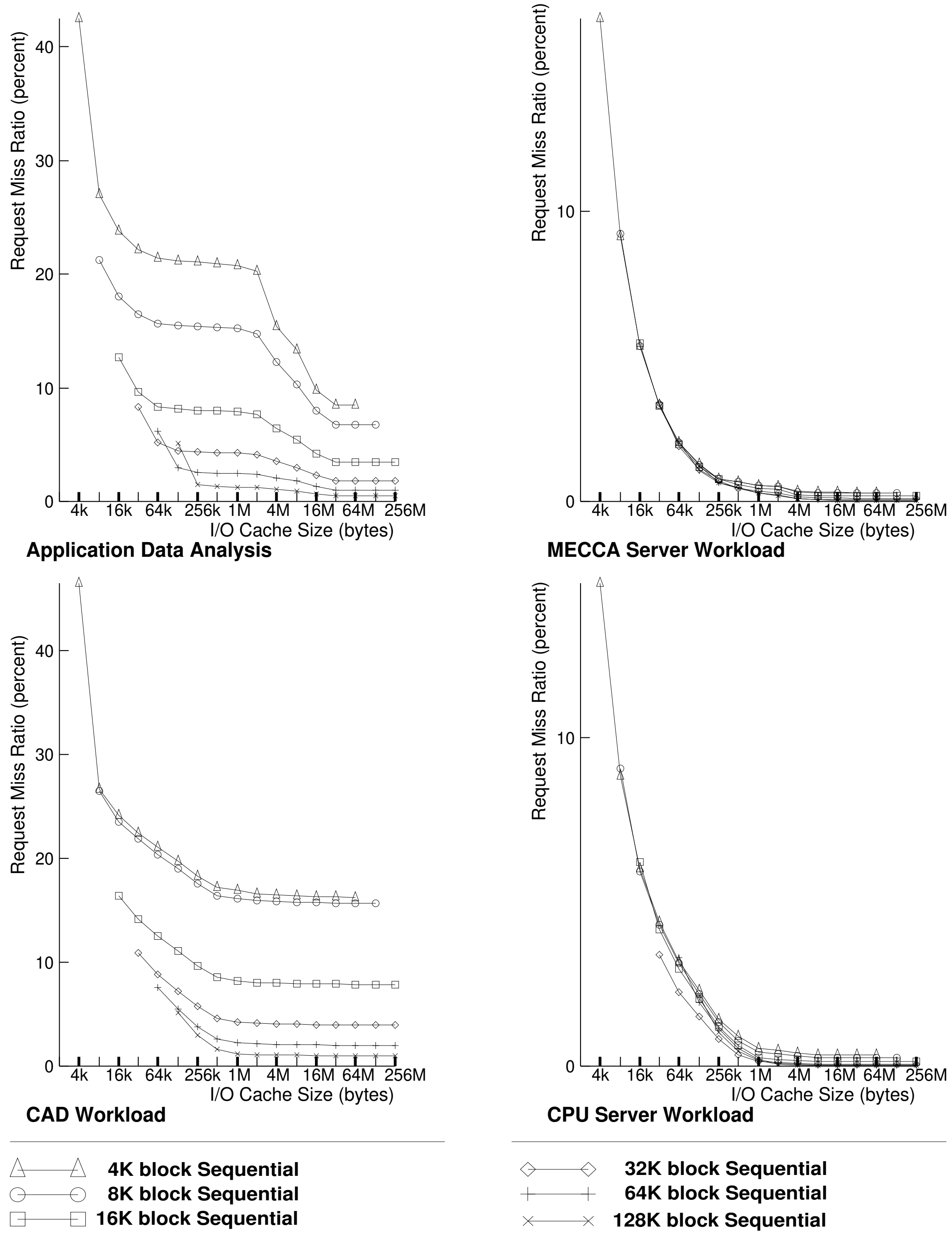


Figure 4.12: Sequential cache miss request ratio for files larger than 512 Kbytes.

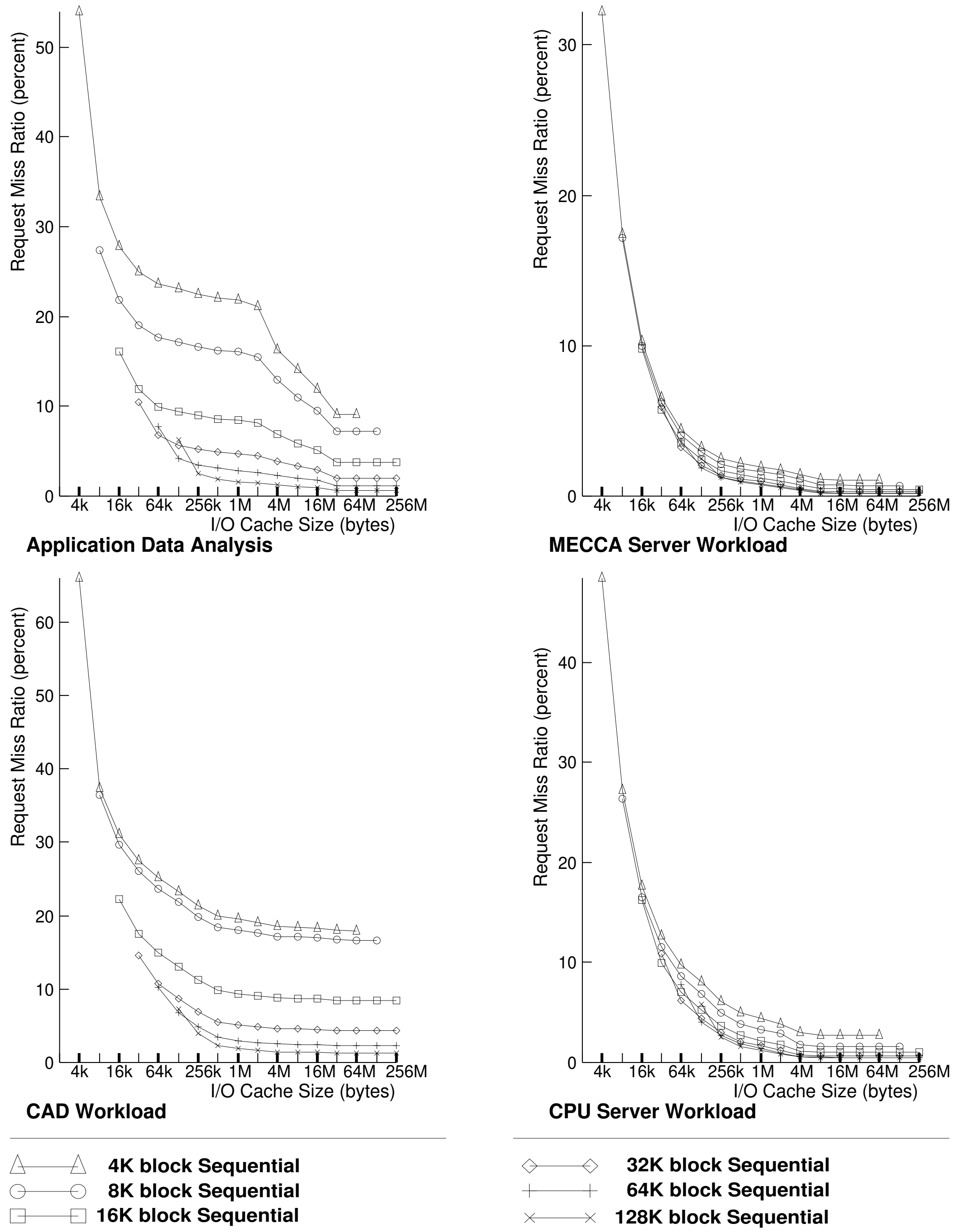


Figure 4.13: Sequential cache miss request ratio for files larger than 128 Kbytes.

Reducing the cut-off from 512 Kbytes to 128 Kbytes increases the number of files and the amount of traffic going to the sequential cache. This has little effect on the actual cache behavior. The only noticeable difference is a slight decrease in the RMR as the cache size increases, and a slight increase in file contention when the block size becomes large enough that the cache only contains a few blocks. With more files of varying sizes, there is likely to be less structured contention, and contention among a larger number of files. The RMR decreases slightly as the cache size increases, because 128-Kbyte files are small enough that the cache reuses of some of the the files before it captures the entire the working set.

The large files in the *MECCA Server* and the *CPU Server* workloads (Figures 4.12 and 4.13) exhibit almost no sequential locality capture, and have almost no correlation between block size and RMR. Increasing the cache size decreases the RMR regardless of block size. The workloads have little sequential locality to capture, even among very large files. This arises when the workload accesses files with very large requests. Most of the large files are executables, which get accessed all at once, rather than datafiles, which tend to be accessed in 8-Kbyte pieces. Because an executable is requested all at once, the number of misses it might generate depends on the cache size rather than the block size. Extending the cut-off down to 128 Kbytes allows the cache to capture somewhat more sequential locality. Now, larger blocks reduce the RMR, but only a few percent. The cache cannot capture significant sequential access behavior, so doubling the block size does not halve the RMR, except for large caches where the block size determines the cold RMR.

The sequential cache cannot capture large file locality for all the workloads. Using large blocks for large files always reduces the RMR, unless there are only a few blocks causing contention. For the workloads that have large sequentially-accessed files, the sequential cache can dramatically reduce the number of misses these files generate. Segregating large files into a separate subcache controls their impact on the overall cache. Large files that are reused less will not push out the many smaller files that are reused more. This segregation works even when the large files are not sequentially accessed.

4.3.2 Temporal Cache Properties

Figures 4.14 and 4.15 show the temporal cache request miss ratio for moderate size files in the four representative workloads. As before, the figures show the temporal cache behavior for each of two file size cut-offs. The *512KB cut-off* allocates only files smaller than 512 Kbytes to the cache, and the *128KB cut-off* allocates files smaller than 128 Kbytes to the cache. As with the sequential cache results, the temporal request miss ratio is the fraction of missed requests contributed by the moderate size files, not the fraction of moderate file request misses.

The temporal cache attempts to capture the greatest temporal locality in the least space. The amount of temporal locality captured depends on the reuse rates and on the amount of data that fits in the cache. While the block size of a cache limits its ability to capture sequential locality, the temporal

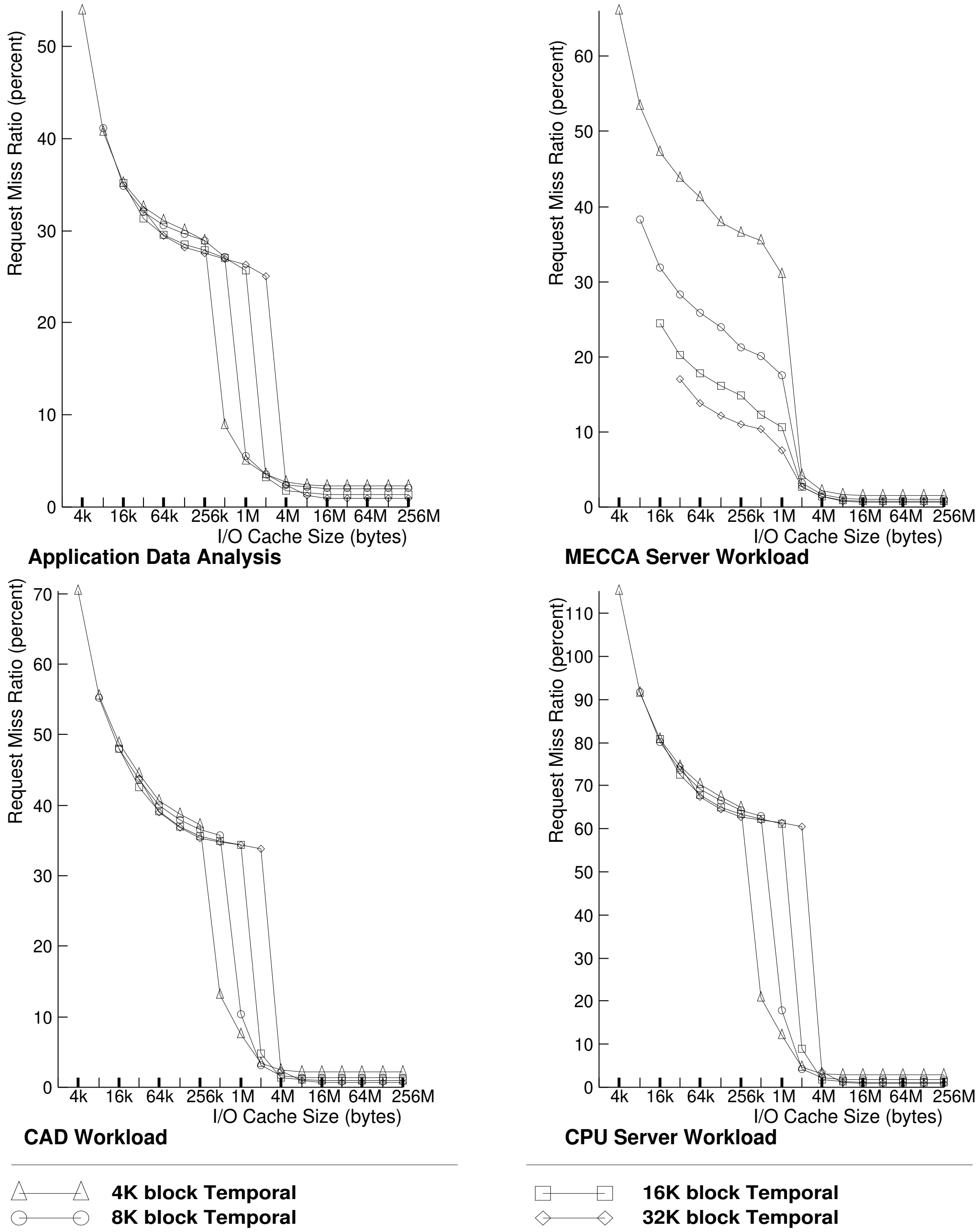


Figure 4.14: Temporal cache miss request ratio for files smaller than 512 Kbytes.

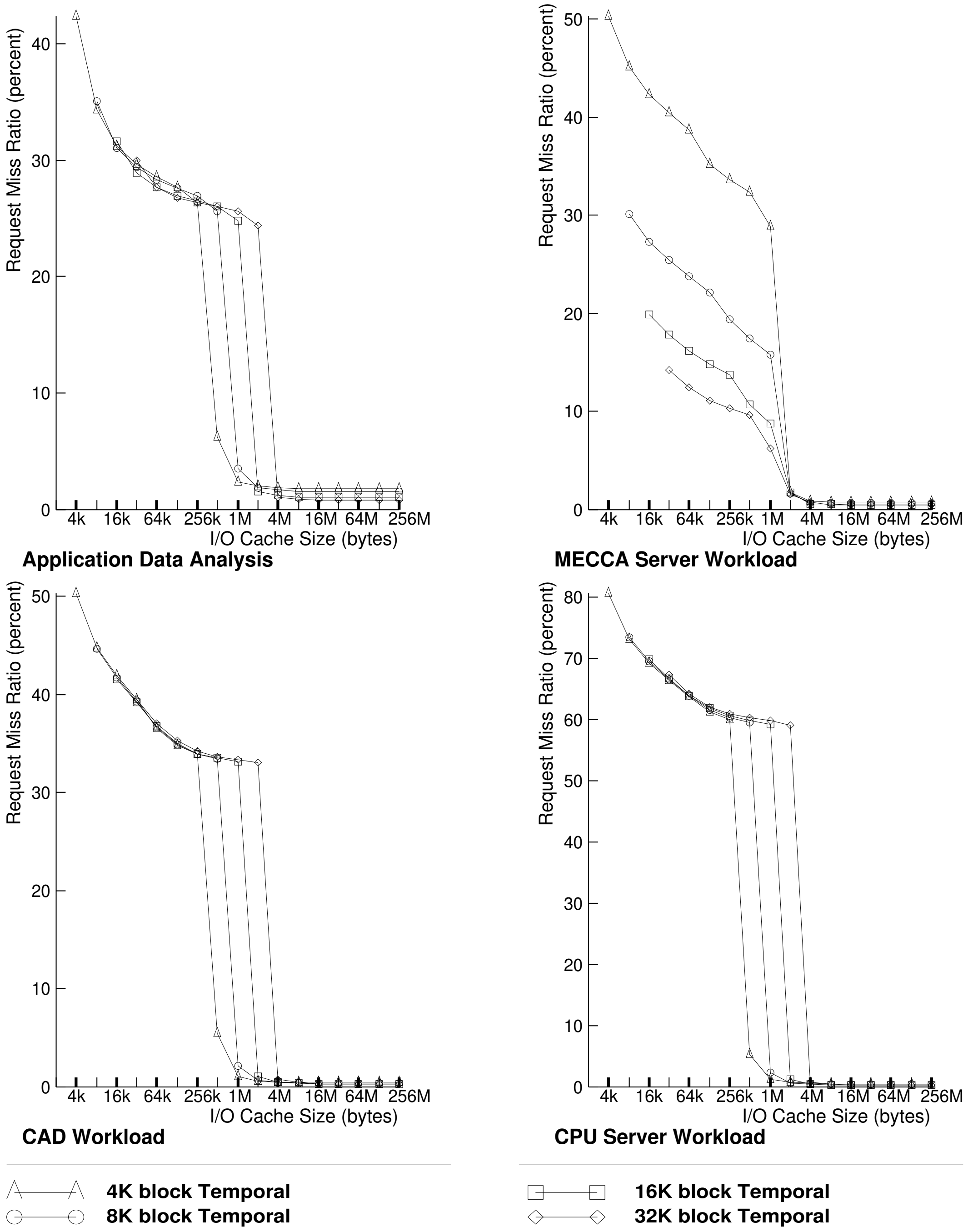


Figure 4.15: Temporal cache miss request ratio for files smaller than 128 Kbytes.

locality has no upper bound.

Smaller cache blocks increase the usable cache space by reducing the amount of unused space per block, and by increasing the number of independent objects that can reside in the cache at one time. Excluding large files eliminates the low-reuse sequential data from the cache, which increases the density of actively used data and allows the cache area to more effectively capture highly reused data.

Moderate Files with Primarily Temporal Locality

As evidenced by their cache behavior, most of the workloads contain primarily temporal locality once the large files have been excluded. The *Application Data Analysis, CAD Chip Build*, and *CPU Server* workloads of Figures 4.14 and 4.15 show only temporal locality behavior. For cache sizes smaller than the working set capture size, increasing the block size produces almost no reduction in the miss ratio. Increasing the block size proportionally increases the cache size required to capture the working set.

Decreasing the file cut-off from 512 Kbytes to 128 Kbytes reduces the number of files and requests to the temporal cache. Eliminating all files larger than 128 Kbytes effectively eliminates all the sequential behavior from the temporal caches for most workloads. Larger blocks do little to reduce the miss ratio for any cache size.

Moderate Files with Both Temporal and Spatial Locality

A few workloads, such as the *MECCA Server*, exhibit both temporal and sequential locality among the moderate sized executables and datafiles. The two may not be easily separable. A large drop in the miss ratio occurs when the cache captures the temporal working set. The working set capture size is independent of block size, indicating the intertwined nature of its sequential and temporal locality. *MECCA* accesses a large set of medium-sized files sequentially. It reuses these files frequently, producing a large working set.

4.3.3 Trading Off Spatial and Temporal Locality

File size provides a simple mechanism for separating the temporal and spatial locality of executables and datafiles. This makes it feasible to tailor the cache management to the expected locality of each request, rather than to the average locality of the entire workload. Sequential data can be cached in large blocks, while small highly reused files can be cached in small blocks.

Managing temporal and sequential locality separately provides several potential advantages. Split management can directly increase locality capture and reduce cache pollution. The temporal cache uses small blocks to reduce wasted space and capture its working set in a minimal area. The sequential cache uses a few large blocks to capture the majority of the sequential behavior in a small area. Limiting the amount of space sequential data can occupy in the cache reduces cache pollution. A separate sequential subcache holding only large files can also prevent large files with little sequential locality, such as executables, from polluting the other subcache.

The cut-off file size for sequential and temporal caches determines the type of locality captured and the sort of workloads for which the caches perform well. A lower cut-off reduces the sequential behavior captured by the temporal cache. However, the lower cut-off introduces more temporal behavior in the sequential cache. Sequential cache behavior is thus more sensitive to cache size. Shifting some of the temporal behavior to the sequential cache does not reduce the amount of cache required to capture the temporal working set.

4.4 Using I/O Characteristics of File Types to Capture Locality

The I/O cache size determines the range of possible locality that it can capture. The choice of cache block size trades off potential sequential and temporal locality capture. The cache configuration and management algorithms need to match the workload locality characteristics. Small caches capture much less locality than larger caches. The type of locality they can capture also differs. A particular cache size can feasibly capture only certain segments of the workload locality. A cache tailored to those segments can achieve better performance. The combined locality of the various file types makes up the total workload request locality.

Inodes and Directories

The highly temporal locality of inode and directory requests greatly outweighs their spatial locality. As noted earlier in Section 4.1, 16 Kbytes of cache arranged as 128 inode-sized 128-byte blocks can capture the inode and directory working sets of the studied workloads. Since inode and directory requests make up almost half of the I/O requests, a large part of the total locality can be captured by properly capturing the inode and directory locality.

Executables and Datafiles

Datafile and executable requests have both temporal and spatial locality. Most workloads have more temporal than spatial locality. The cache captures only a small amount of the temporal locality unless the whole working set fits in the cache. Four to eight entries suffice to capture what temporal locality exists, short of working set capture. Since most of the locality is temporal, capturing the working set is crucial to capturing datafile and executable requests in the cache. The datafile and executable working set size varies from about 512 Kbytes to 16 Mbytes for the given set of workloads. Although less significant than the temporal, spatial locality forms a substantial part of the total locality. Much of the sequential locality can be captured with only a few large blocks.

Choosing a Cache Size

There are three major cache size regions: The small I/O cache region, the working set capture region, and the large I/O cache region. Caches in the small region cannot capture the expected working set of the entire workload. Caches in the large cache region are big enough to capture the working set of most workloads. The working set capture region covers the intermediate sizes.

Each of the three cache size regions needs to capture a different sort of locality: (1) Small caches have very limited space and should be designed to capture locality that requires little space. A small cache can potentially capture inode and directory requests, some small amount of datafile and executable temporal locality and some sequential behavior. (2) Caches in the working set capture region are large enough that they should easily capture the inode and directory working sets. The cache should be tailored toward holding the datafile and executable temporal working set, and then providing adequate support for capturing some sequential locality. (3) Large caches have sufficient space to capture the temporal locality of inodes and directories and of the datafiles and executables. The cache needs to capture the remaining sequential locality and reuse of large sequential files, which have a reasonably long time period between reuse.

Chapter 5

Attribute I/O Caches

This chapter introduces the attribute cache as a mechanism for reducing I/O cache read misses. Attribute caches use directives in the form of file attributes. They improve cache performance by more closely matching the cache with expected workload behavior. Uniform cache schemes try to best capture the access behavior of the entire workload. The resulting cache does capture locality, but designing to the statistical properties of the entire workload limits its effectiveness.

Attributes indicate files with similar cache behavior. Attribute caches more efficiently hold individual data requests because of the narrower range of expected behavior for each request. Attribute caches differ from *attribute caching*. Attribute caches use attributes to guide data management, where as attribute caching refers to storing file attributes in a cache [SO92].

This chapter describes the design trade-offs and performance for one of many possible attribute cache schemes. This attribute cache scheme uses fixed cache partitions for different file attributes. Each partition has a block size designed to capture the locality of files with particular attributes. Because the type of locality a cache captures depends on cache size, the partitioning varies with the size of the cache.

5.1 Attribute Cache Framework

5.1.1 Definitions

Attribute Cache: An I/O cache that uses file information to choose the cache strategy for I/O requests associated with each file.

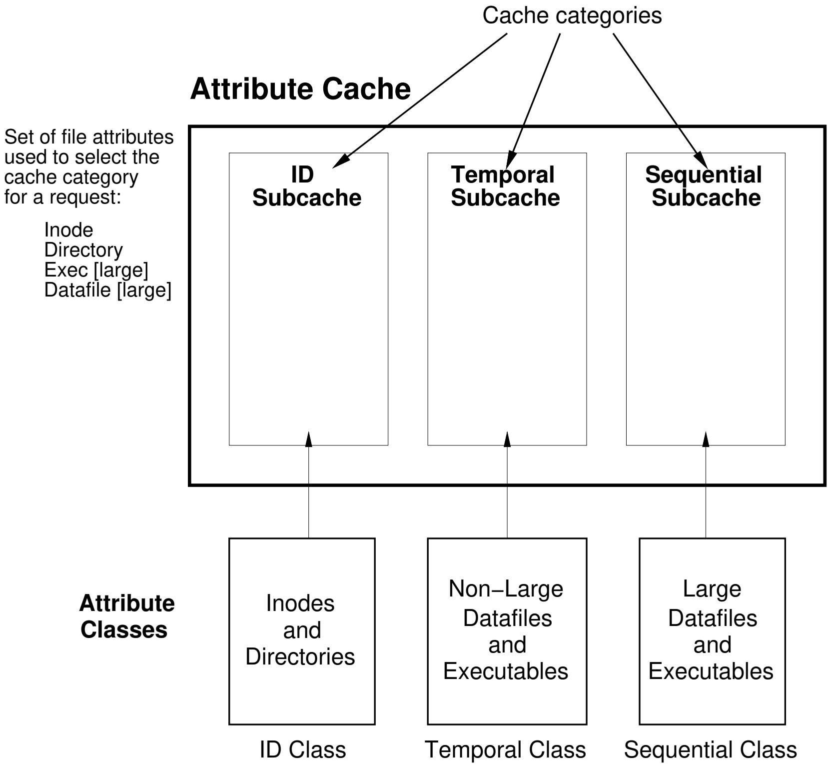


Figure 5.1: Anatomy of an attribute cache.

Attribute: An attribute indicates the expected cache behavior of a file. Attributes may be known features of a file, or they may be explicitly assigned to the file. A file may have multiple attributes that define its expected cache behavior.

Cache Categories: The different cache strategies, or subcaches, used by the attribute cache. The file attribute determines the category for an individual request.

Attribute Class: The set of files that map to a particular cache category.

The definition of attribute caches covers a broad range of caches. This work examines only attribute caches that use a separate partition for each cache category, and that use the file type and the file size for attributes. Figure 5.1 shows the attributes, cache categories and attribute classes used by one attribute cache. This attribute cache uses one of three separate strategies, depending on the category of cache behavior expected. The three cache categories are ID, temporal, and sequential. The file attributes are the four file types *inode, directory, datafile,* and *executable,* and an assigned attribute *large* derived from the file size and the file cut-off value. The attribute classes map attributes to cache categories. For example, files with inode or directory as their attribute belong to the ID class, and get assigned to the ID subcache.

I/O caches can easily incorporate attributes because all I/O requests go through the operating system. User programs request the operating system to look up and retrieve the data from a file or portion of a file. Before retrieving any data from the file, the operating system must fetch information about the file's access permissions, size, and location. Assuming these are all valid, it calculates the address of the requested block and then looks in the cache for the data.

The operating system thus knows a significant amount of information about each file and data block before doing any cache lookup. It can use this information directly as an attribute. Alternatively, other information can be added to the file information with little additional space or time penalty.

System designers typically implement I/O caches in main memory or in the disk subsystem. When implemented in the subsystem, the overhead of sending an additional control word is minimal. Caches implemented in main memory are typically associative LRU caches accessed through hash tables and linked lists. Using attributes to direct the cache scheme adds only a few instructions for selecting between different algorithm sections. The overhead for using attributes is again minimal.

5.1.2 Attribute Cache Goals

File attributes separate workloads into components, each with a distinct size distribution, average reuse rate and typical access pattern. Tailoring the cache to the workload components rather than to the conglomerate lets the cache efficiently capture a wide range of behavior. The designer must try to balance the requirements of the components and thus improve total workload locality capture. First and foremost, the cache designer must try to reduce the number of read requests that miss in the cache and thus require disk access. Read misses reduce performance in many ways:

- Processes wait for read requests to finish before continuing execution. If the cache can not satisfy the request, the process must wait for a disk access. If another process is ready to run, the waiting process is switched out. The original process may not return immediately after the disk access completes.
- Most requests ask for only a small amount of data, which means the request overhead dominates. Reducing the read misses has a bigger impact on the total performance than reducing the bytes transferred.
- Reducing disk accesses by coalescing multiple requests into single disk operations reduces the disk busy time.
- Reducing requests increases the disk efficiency, either by transferring the same amount of data in fewer operations or by transferring less data.

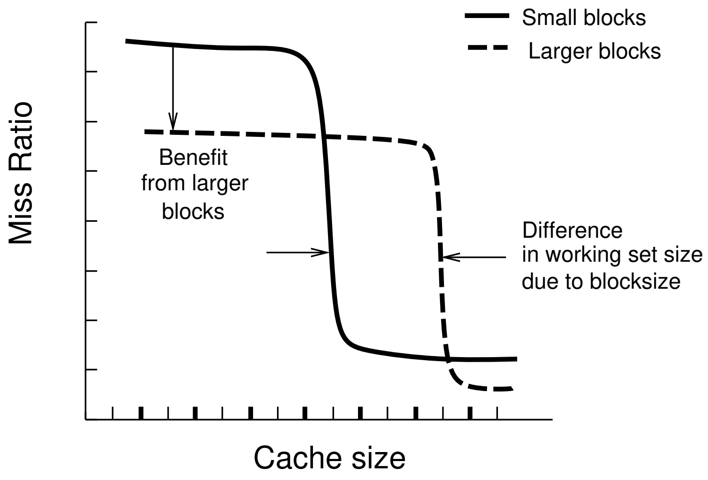


Figure 5.2: Desired cache properties for data and executables.

The primary goal of any I/O cache scheme should be to reduce the number of read request misses. Secondary goals should be to reduce the associated read request service time, and to maintain a competitive disk write load.

Figure 5.2 illustrates the general miss ratio properties of datafiles and executables in caches with differing block sizes. Small caches cannot capture the temporal locality for many workloads, but they can capture sequential locality. Using large blocks for these caches reduces the miss ratio.

As the cache gets larger, it begins to capture the working set of the workload. The amount of cache required to capture the working set is a function of both the actual working set size and the way it maps into the cache. Larger blocks have more unused space, increasing the cache size required to capture the working set. Unix I/O workloads tend to have a temporal working set that is very sensitive to block size and low space utilization within blocks. Most of the I/O requests go to small files. Reusing these small files provides the bulk of the benefit from capturing the working set. Since most of the datafile and executable runs are smaller than 4 Kbytes, increasing the block size beyond that results in fragmentation and low cache utilization. Doubling the block size typically doubles the cache size required to capture the working set. The working set capture size increases linearly with the cache block size. In this region of mid-sized caches, then, smaller blocks provide greater benefit than larger blocks.

Large caches capture the temporal working set regardless of the block size. The majority of the I/O bytes transferred come from long runs with little or no fragmentation. In a large cache that has captured the working set of the workload, the unused space within a block from the small files is still related to the block size. Larger blocks, however, capture the sequential locality of long runs, which constitute the bulk of the bytes transferred. For the largest caches, then, large blocks help more than small blocks.

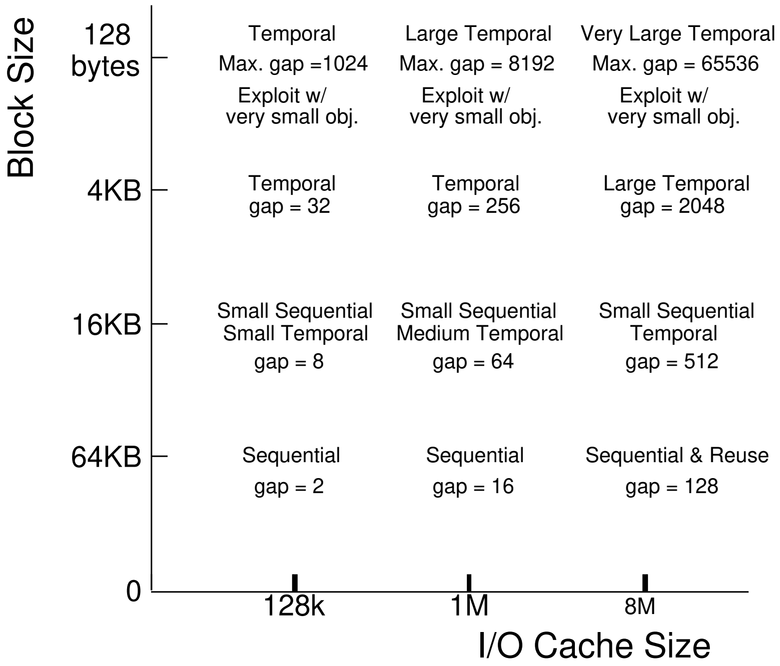


Figure 5.3: Various locality supported by different cache configurations.

This chapter presents an attribute cache scheme that improves the resource utilization of the cache and, in doing so, reduces the read request misses. The scheme was designed based on an evaluation of the overall workloads. It is neither an optimal solution given the cache partition requirement, nor an optimal solution given the set of experiments simulated. Many cache choices depend on the expected workload. The goal was to pick a simple scheme that works well over a broad range of workloads and for many potential disk or network systems.

5.1.3 Balancing Workload Needs

In a uniform cache scheme, all requested data competes equally for cache space. An attribute cache, however, breaks the workload into components, each with a separate attribute class. Each attribute class vies for resources. To improve performance, an attribute cache must balance the resource requirements for the different attribute classes, trying to capture the most locality for a given cache size.

The cache size determines the potential locality that can be captured. Figure 5.3 shows some examples of potential locality capture for various cache sizes and block sizes. Small caches can capture temporal locality when the *gap*[1] between re-references is small, or when all the objects are

[1]The *gap* is the number of unique blocks referenced between successive references to a particular block. For a fully

small enough that many objects fit in the cache [Quo94]. Alternatively, small caches can capture sequential locality, provided the gap between re-referenced objects is very small, say less than four[2]. The locality that a cache actually captures depends on the workload as well as the block size.

Segmenting resources between cache categories involves matching the existing workload locality with the proper cache resources necessary to capture the locality. An efficient allocation of resources gives cache space to the component or components that can capture the most locality in that space. The scheme wastes cache resources if it allocates space to components that cannot effectively use the space to reduce misses. Since the working set size varies for the different components, the cache partitioning can favor components whose working set can be captured at a given size.

A cache with fixed partitions loses some efficiency when it does not equally use all of the partitions at any point in time, but the gains outweigh the losses. Allocating a fixed amount of cache for each cache category guarantees space for each attribute class. This guarantee allows components with short internal gap distances to capture locality even if their gap distance within the whole workload is large. Fixed partitions prevent large objects from sweeping the cache and expelling many small objects. Fixed partitions allow controlled evaluation of allocation options.

5.1.4 Baseline for Comparison

A Unix style cache will be the baseline comparison for attribute caches. The Unix baseline allocates 32 Kbytes for inodes, and divides the rest of the cache into 4-Kbyte blocks. It is fully associative, and has LRU replacement. The cache allocates writes.

Figure 5.4 shows the baseline read request miss ratio for all the workloads. The four sample workloads featured in the remainder of the chapter have solid symbols for highlighting. The data separates naturally into three regions of cache operation. Caches less than 64 or 128 Kbytes have high miss ratios, and increasing the cache size dramatically reduces the read request miss ratio. The region between 128 Kbytes and 8 Mbytes captures the data and executable working sets for most of the workloads. Extending the cache size beyond 8 Mbytes captures very large data sets, like that found in Ingres, and captures reuse of large runs. Each region has unique constraints and workload behavior, and will be evaluated individually. The boundaries between the three regions are not rigid, but for simplicity the regions are defined as follows:

Small I/O Cache:	128 Kbytes or less.
Medium I/O Cache:	256 Kbytes to 4 Mbytes.
Large I/O Cache:	8 Mbytes or more.

associative cache, a block of data hits in the cache if the gap is smaller than the number of blocks in the cache.

[2]That is, fewer than four other objects referenced between each re-reference to the same object.

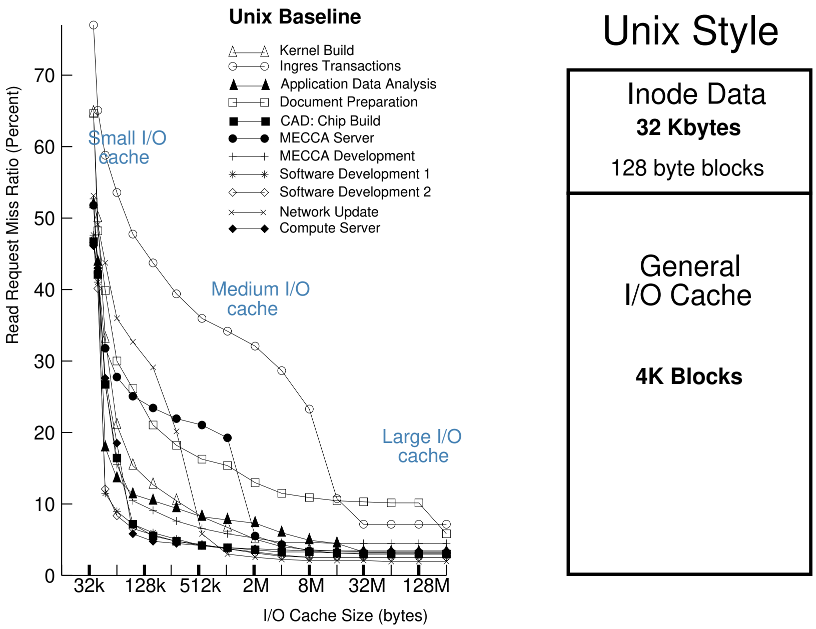


Figure 5.4: Unix style cache read request miss ratio.

5.2 Attribute Cache Design Parameters

Fixed-size subcaches allow the designer to evaluate many potential attribute cache configurations. The simulator handles each subcache independently, producing results for a range of cache sizes on each run. A post-simulation phase combines subcache results to determine the total cache read request misses. Figure 5.5 shows the cache configurations used to demonstrate the usefulness of attributes and to determine how subcache splits should vary with total cache size. Four separate subcaches are used to construct two-category and three-category attribute caches.

The inode and directory subcache, also called the ID subcache, stores inodes and directories in small 128-byte blocks, packing many objects in a small space. The general I/O subcache is designed to capture both the temporal and sequential locality of non-ID requests. The temporal subcache caches the bulk of the datafile and executable references; its size determines whether the cache can capture the whole workload working set. The sequential cache captures large sequentially accessed objects or large objects with very low expected reuse. All the results shown in this chapter use a 512-Kbyte file cut-off to split file references between the temporal and sequential subcaches.

To concisely describe attribute cache organizations requires a notation convention. The important characteristics of each subcache are the cache size, the block size, and the cache category. Each subcache will be described as follows:

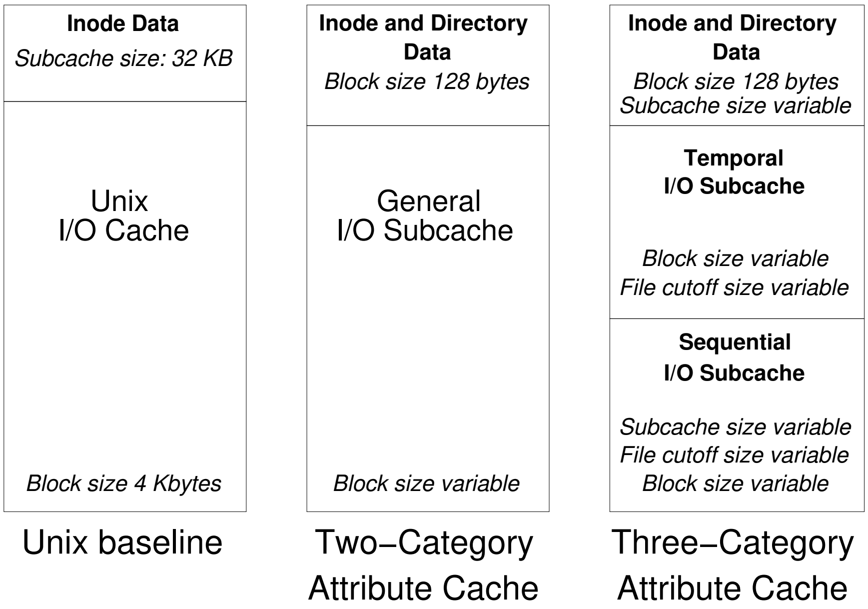


Figure 5.5: Logical configurations for attribute I/O caches.

Cache_size/Block_size Cache Category

A string of subcache definitions describes a complete attribute cache configuration. For example, a two-category attribute cache with a 64K inode and directory cache and a general cache with 8-Kbyte blocks is described as follows:

64K/128 ID, x/8K General

The ID cache is fixed at 64-Kbytes and the General cache size determines the total cache size. A 128 Kbyte I/O cache would have a 64 Kbyte general cache, where as a 1 Mbyte cache would have a 960 Kbyte (1 MB less 64 Kbyte) general cache. From the size and blocks size it is easy to determine the number of cache blocks. The 64K/128 ID cache has 512 blocks.

5.2.1 Small I/O Cache Region

The amount of existing locality any cache captures depends on the cache configuration, especially among small caches. Small I/O caches cannot capture the working set of most workloads, so configurations that use cache area more efficiently capture a greater fraction of the workload behavior.

Several techniques increase the cache utilization, including (1) only storing frequently reused data in the cache; (2) increasing the number of objects stored in the cache by excluding larger objects; and (3) matching the cache size to the cache configuration that best captures locality. The characterization data in Chapters 3 and 4 show that directories are the most reused file type, followed by inodes. Even a small cache can hold many inodes and directories. A small cache can also potentially capture the small working set typical of inodes and directories.

Datafiles and executables exhibit the sort of locality that a small cache can easily capture. The datafile and executable request miss ratio of Figure 4.10 show significant locality capture with only four cache blocks. Beyond four blocks, reductions in request misses diminish until the cache has captured the entire working set. Many workloads have spatial locality that a cache using large blocks can capture. In fact, a single large block suffices to capture this locality.

Figure 5.6 compares several two-category attribute cache configurations against the baseline. The configurations are designed to capture locality in the small cache region; the region over which the configurations are compared is between 4 Kbytes and 256 Kbytes. As the cache size increases, the cache options increase. Allocating the entire cache to inodes and directories can be advantageous if the total cache is very small—four to sixteen kilobytes, depending on the workload. Allocating resources to both the ID subcache and the general I/O subcache captures the highly reused inodes and directories, as well as some temporal locality for the datafiles and executables. Increasing the block size in the general subcache from 4 Kbytes to 16 Kbytes captures some of the sequential locality for the datafiles and executables. The best configuration changes with cache size. As the cache size increases, so should the inode and directory partition, and the block size of the general subcache.

5.2.2 Medium I/O Cache Region

The medium I/O cache region coincides with the range of sizes that capture the working set. Capturing the working set significantly reduces request misses. Thus, medium caches need to provide low request miss ratios without increasing the cache size required to capture the working set.

All medium I/O caches are large enough to capture the inode and directory working set. A 64-Kbyte and 128-Kbyte inode and directory subcache respectively captures 85% and 94% of the inode and directory requests. Medium cache designs need to concentrate on reducing the datafile and executable request misses.

Datafiles and executables have both temporal and sequential locality. Small blocks capture more

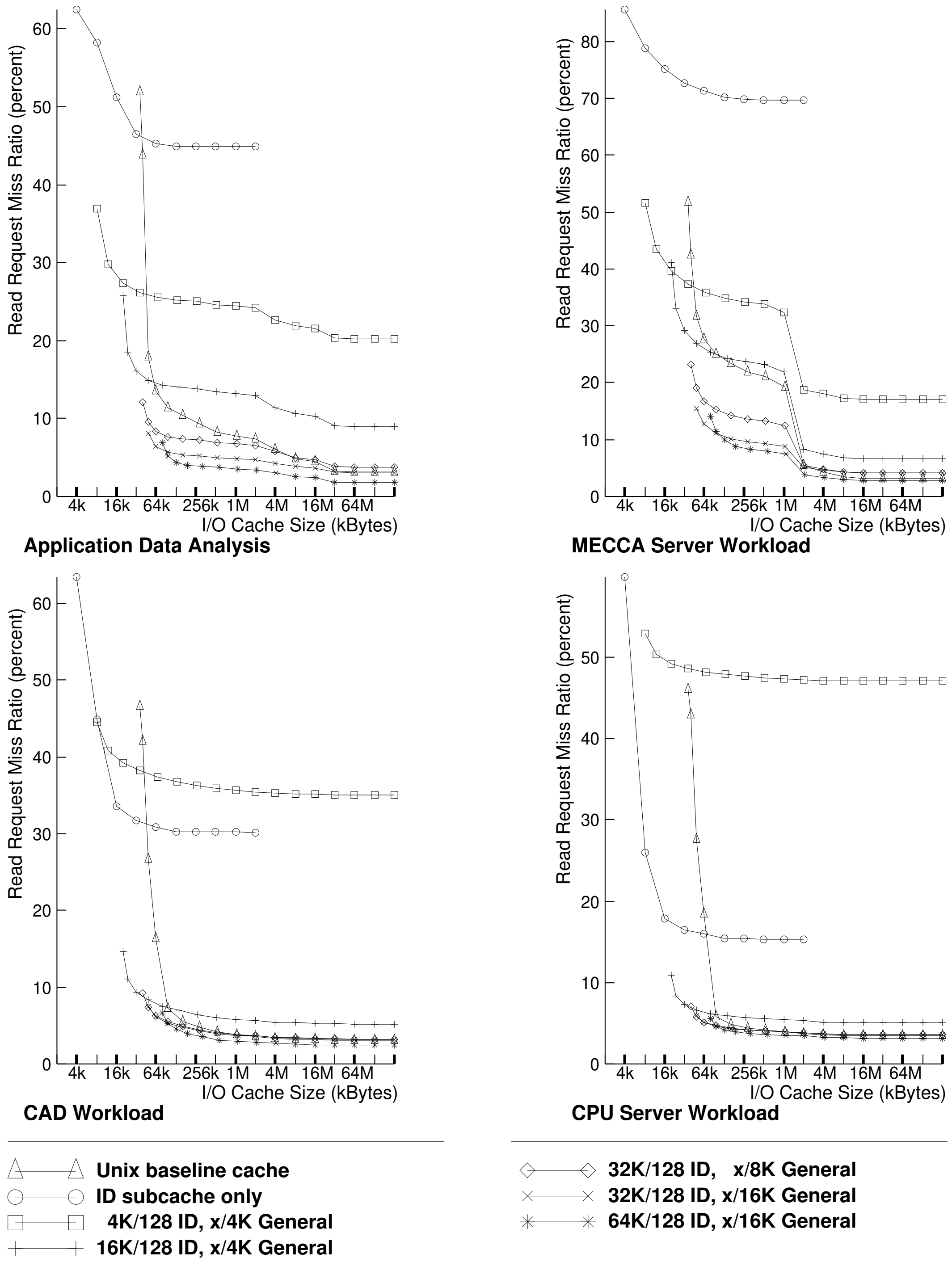


Figure 5.6: Small cache options.

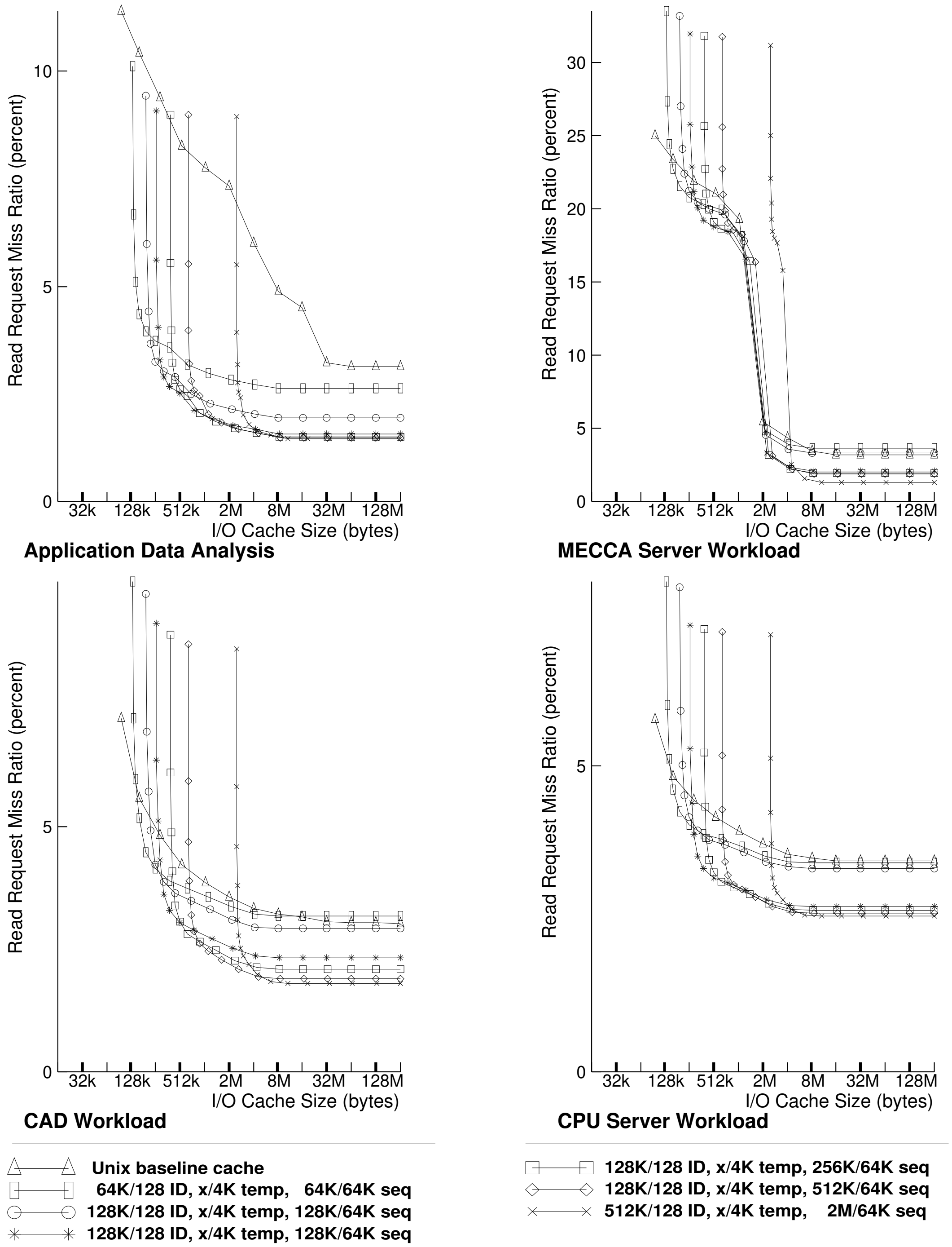


Figure 5.7: Medium cache options

temporal locality, and large blocks capture more sequential, or spatial, locality. Much of the workload's sequential locality results from large files. By separating the large datafiles and executables into the *sequential* attribute class, and the remaining datafiles and executables into the *temporal* attribute class, the attribute cache can capture both the temporal and sequential behavior in a smaller cache.

There are several potential advantages of managing temporal and sequential locality separately. Split management can directly increase locality capture and reduce cache pollution. Packing more useful data in the same cache space increases locality capture. The temporal cache can use small blocks to reduce wasted space and capture its working set in a minimal amount of space. The sequential cache can use a few large blocks to capture the majority of the sequential behavior in a small cache area. Splitting reduces cache pollution by limiting the amount of space sequential data can occupy in the cache. Large sequential data files have a low average reuse rate. Allowing these files to evict useful data pollutes the cache with data unlikely to be reused. This can also prevent the large files with little sequential locality, such as executables, from polluting the cache. The attribute cache works well at handling outlying characteristics, like those displayed by inodes, directories, and large datafiles.

In the medium size region, the goal is to significantly reduce the number of sequential accesses to large files, while not significantly increasing the required working set size for the remaining files. This requires limiting the sequential cache space until the cache has captured the working set of the workload. Since there is no way to determine if a running workload has captured its working set, a conservative approach is necessary. For the workloads studied, the working set size ranges from 512K to 16M. Only Ingres, the synthetic workload, required 16M. Of the remaining workloads, none required more than 2M.

Figure 5.7 compares the read request miss ratio for several three-category attribute caches against the baseline cache. The configurations are designed to capture locality in the medium cache region; the region over which the configurations are compared is 256 Kbytes to 4 Mbytes. The figure shows the full range of behavior for each cache configuration. The plot resolution is such the the smaller caches for each configuration appear to be the same size. For example the

512K/128 ID, x/4K temp, 2M/64K seq

cache configuration has a 512 Kbyte ID subcache, a 2 Mbyte sequential subcache, and a temporal subcache ranging from 4Kbytes to 256 Mbytes. On the log scale, the caches with temporal subcaches from 4K to 128K bytes all appear as 2.5 Mbyte caches even though they vary in size from 2564 to 2688 Kbytes.

The sequential subcache uses 64-Kbyte blocks to significantly reduce sequential misses. Each workload requires a certain size temporal cache to capture the working set. Larger sequential or ID subcaches capture more locality, but also shift the total cache size required to capture the working sets.

The MECCA Server workload has one of the largest working sets. The sequential subcache causes an increase in the cache size required to capture the working set, and produces no reduction in the miss ratio. The ID subcache also increases the cache size for working set capture, but a larger ID subcache reduces the cache miss ratio. Most of the workloads use the sequential subcache to some extent and show lower read request miss ratios in this middle range.

5.2.3 Large I/O Cache Region

In the large cache size region, the prime objective is to try capturing the entire workload in as few request misses as possible. There is adequate cache to capture the working set, and there is enough left over to try capturing reuse of the large files. Neglecting the large file reuse does not always impact the total requests very much (a 1M file requires only 8 128-Kbyte block requests), but it does affect the total bytes transferred. Since large files constitute the majority of bytes transferred, capturing their reuse is critical to keeping the disk transfer time low.

Figure 5.8 compares three-category attribute caches having large ID and sequential subcaches against the baseline cache, and against a two-category cache having 16-Kbyte blocks in its general I/O subcache. The figure shows both 64- and 128-Kbyte block sizes for the sequential subcache. The additional benefit from doubling the sequential block size depends on the amount of sequential behavior in the workload. If the workload has a large sequential component, the larger blocks reduce the miss ratio. With a multi-megabyte sequential cache, the large number of blocks suffices to eliminate any conflicts. At such a size, increasing the block size to 128K does not increase the read request miss ratio for any workload. For most of the workloads, increasing the sequential subcache size beyond two megabytes does little to reduce the read misses, but it significantly reduces the number of bytes transferred to and from disk. A 16-Mbyte sequential subcache captures the large file reuse for all of the workloads.

Increasing the inode cache beyond 128K reduces the misses on several workloads. Some of the workloads see some slight decrease in the number of inode miss requests as the ID subcache size increases. A couple of workloads see a significant reduction when the ID subcache increases to 1M, and a few more when it goes to 4M. In general, little gain accrues from increasing the ID cache beyond 128K, unless goes to at least 1M.

Large-sized caches use attributes to capture the temporal and sequential locality of the workload, including that of very large files, while insuring that no single outlying file or application can destroy already captured working sets. For these workloads, the largest sequential subcache needed is 16M, and the largest ID subcache is 4M. Caches greater than 32M can support these large sequential and ID caches.

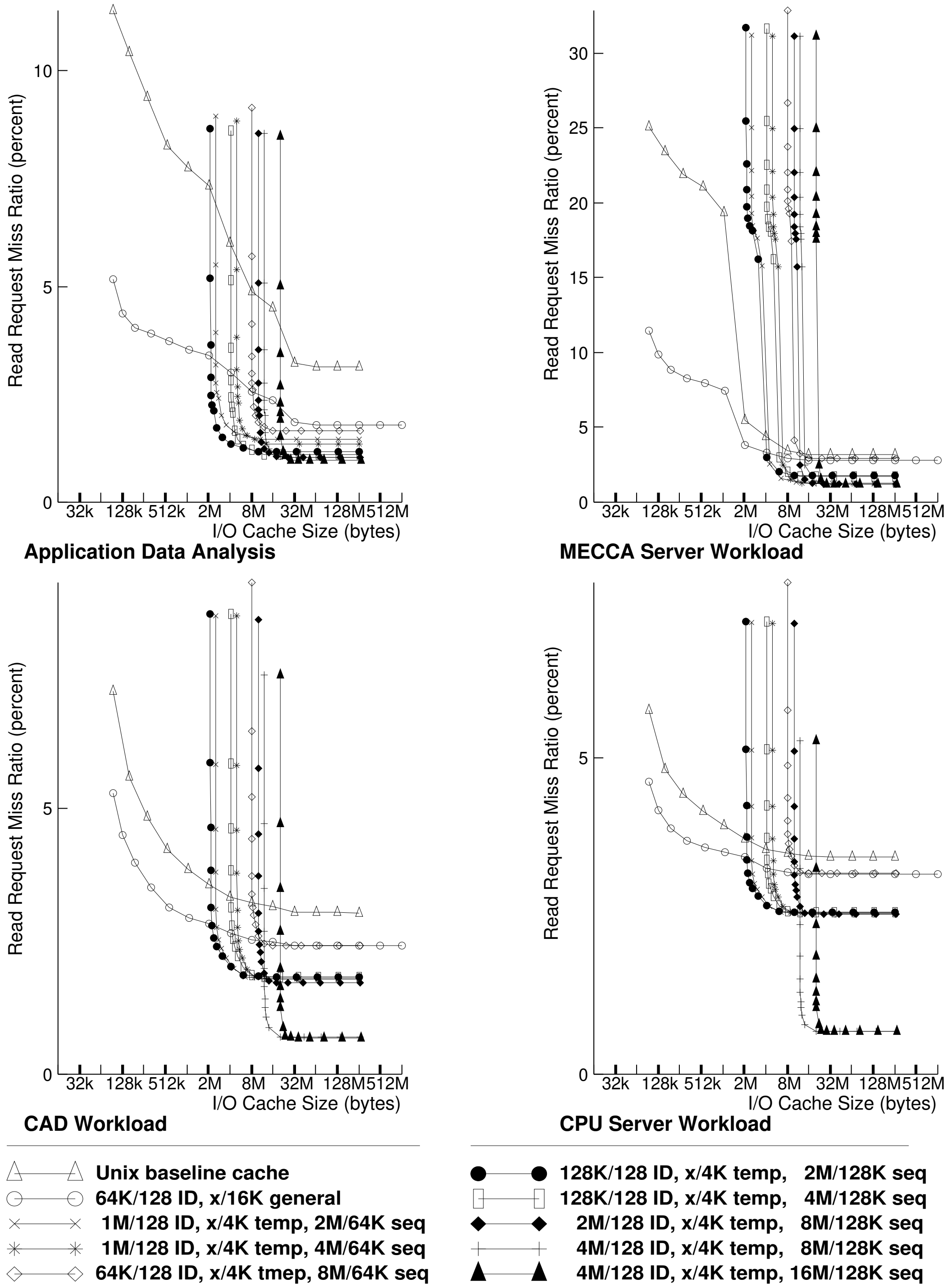


Figure 5.8: Large cache options

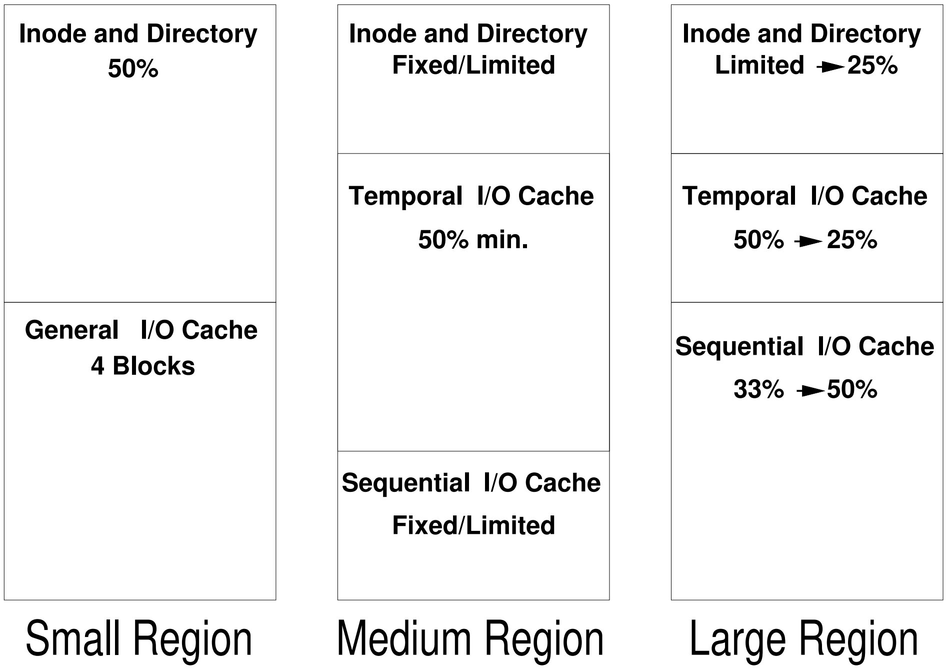


Figure 5.9: Variable attribute cache configurations for each size region.

5.3 Variable Attribute Cache Scheme

The *variable attribute cache scheme* uses attributes to substantially reduce read request misses. No single cache configuration produces low miss ratios over a broad range of caches, hence the scheme varies with cache size. The design exploits common workload behavior and systematically varies the attribute cache configuration along with its cache size to capture the appropriate behavior. The resulting design is not optimal, but it shows the type of benefits that could be expected from a real attribute cache scheme.

Figure 5.9 shows the attribute cache configurations and the general policy governing subcache space allocation for each of three cache size regions. In the small cache region, the scheme uses a two-category attribute cache, allocating half the cache to inodes and directories, and partitioning the other half into a *general subcache* having four equal blocks. The general subcache is designed to capture both temporal and sequential locality. In the medium cache size region, the scheme allocates the bulk of the area to the temporal subcache, which uses 4-Kbyte blocks to capture the temporal working set. It allocates from 64 to 128 Kbytes to each of the ID and sequential subcaches to capture ID requests and sequential requests. In the large cache region, the scheme increases the sequential subcache so that it occupies a large part of the cache, starting at about one-third and increasing to one-half at the high end of the cache region. The ID subcache remains fixed at 128 Kbytes until the cache becomes large enough to support a multi-megabyte ID subcache, at which point it expands

Small Cache Region	Cache Size
16K/128 ID, x/4K General	32 Kbytes
32K/128 ID, x/8K General	64 Kbytes
64K/128 ID, x/16K General	128 Kbytes

Medium Cache Region	Cache Size
64K/128 ID, x/4K Temporal, 64K/64K Sequential	256–640 Kbytes
64K/128 ID, x/4K Temporal, 128K/64K Sequential	1216 Kbytes
128K/128 ID, x/4K Temporal, 128K/64K Sequential	1280–4352 Kbytes

Large Cache Region	Cache Size
128K/128 ID, x/4K Temporal, 2M/128K Sequential	6272–18560 Kbytes
128K/128 ID, x/4K Temporal, 4M/128K Sequential	20608 Kbytes
2M/128 ID, x/4K Temporal, 8M/128K Sequential	26624 Kbytes
4M/128 ID, x/4K Temporal, 16M/128K Sequential	36864–282624 Kbytes

Table 5.1: Variable Attribute Cache Scheme.

to 25% of the cache area.

Table 5.1 describes the exact cache configurations used in each region by the variable attribute cache scheme.

5.3.1 Read Request Behavior

Figure 5.10 compares the variable attribute cache scheme with the Unix baseline scheme for the four representative workloads. The variable scheme lowers the RRMR across the full cache range. The resulting miss ratio usually corresponds with that of caches eight times the size. For many medium and large caches, however, the variable scheme produces RRMR's below that of the maximum Unix baseline cache.

The MECCA Server workload exhibits anomalous variable-scheme cache behavior. As previously noted, the workload locality is only captured by large cache blocks. In the small cache region, the variable scheme uses larger blocks which are ideal for this workload. In the medium cache region, the variable scheme changes to 4K blocks in the temporal cache and partitions space for a sequential cache. The 4-Kbyte blocks produce comparable miss ratios for each of the two schemes, but the MECCA Server workload, benefits little from the sequential subcache. This subcache sits unused, increasing the total space required to capture the working set.

Figure 5.11 compares changes in the read requests for the four sample workloads, relative to the Unix baseline cache. The workloads see dramatic reductions in their read request misses for both small and very large caches, and moderate reductions over a broad range of middle cache sizes. These reductions result in fewer disk read accesses and fewer times when applications must wait

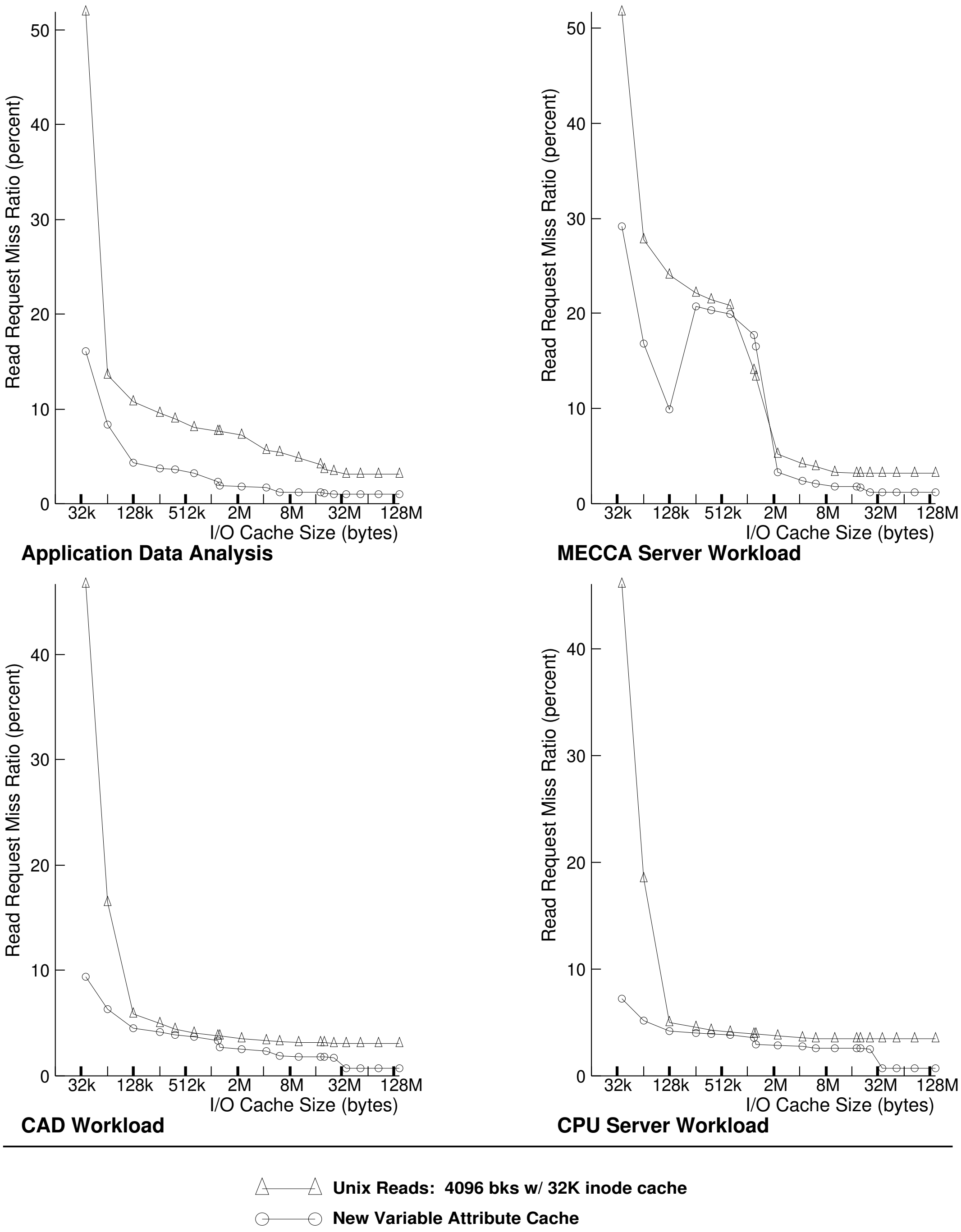


Figure 5.10: Attribute cache scheme request miss ratios.

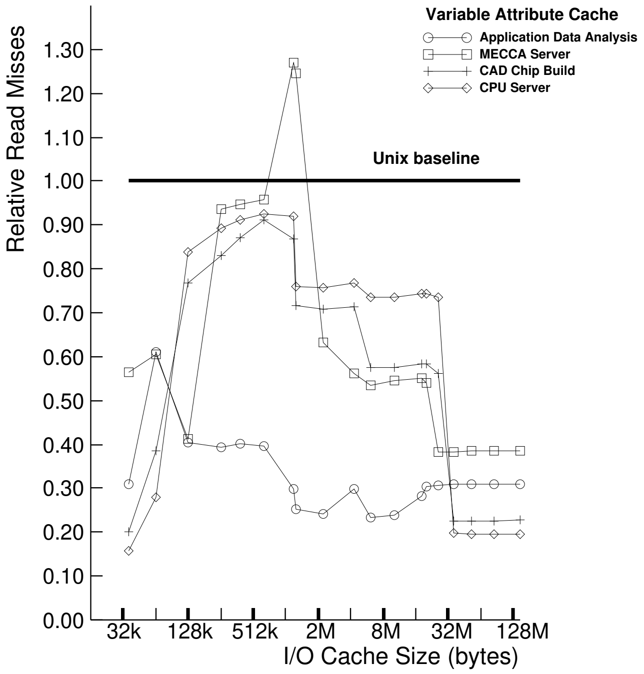


Figure 5.11: Relative read misses for the four sample workloads.

for I/O requests to complete.

Small Cache Behavior

In the small cache region, the variable scheme reduces read request misses by capturing more inode and directory temporal locality and more datafile and executable sequential locality. The ID subcache eliminates unused space within a block for directories. The same cache size thus holds many more directory entries. The ID cache also protects inode and directory data from being evicted by datafiles and executables. In the Unix I/O subcache, directories compete with datafiles and executables. In the small cache region, this competition prevents the directories from capturing their working set. Since the combination of inodes and directories produces roughly 75% of the total requests, increased capture produces large reductions in the total disk requests generated.

Medium Cache Behavior

For medium cache sizes, the variable attribute cache scheme performance depends highly on the exact nature of the workload. Figure 5.11 shows, for certain cache sizes, that the variable scheme reduces misses by 75% in the Application Data Analysis workload, but produces 25% more misses for the MECCA Server workload. In this region, both cache schemes capture similar amounts of inode and directory requests. The Unix subcache is large enough to capture most of the directory working set even with the competition from datafiles and executables. Directories are reused frequently, so only large runs of datafiles or executables throw out the working set. The variable scheme protects the directory working set from being periodically thrown out of the cache, but the benefits here are much less than in the small cache region.

The sequential subcache acts as a liability or an asset depending on workload behavior. Workloads with few large file requests reap no benefit from the sequential subcache, and it goes unused. The unused sequential subcache merely increases the total cache size required to capture the workload working set. Workloads that do have large sequentially-accessed files show large read miss reductions. The sequential subcache directly reduces the sequential read request misses. With typical 8-Kbyte read requests, the 64-Kbyte cache blocks can reduce sequential misses by 88%. Segregating large files to the sequential subcache reduces cache pollution in the bulk of the cache. The temporal subcache protects its working set from large files, which lowers the disk accesses needed to service the temporal read requests.

Large Cache Behavior

Performance trade-offs for large caches smaller than 24 Mbytes resemble those in the middle region, except that both the Unix baseline and the variable scheme captures the temporal working set. The sequential subcache captures locality without occupying space that might otherwise allow capture of the temporal working set. Increased sequential cache size begins to capture reuse for some large files.

Beyond 24 Mbytes, the variable attribute cache scheme significantly outperforms the Unix baseline. Both schemes have read request miss ratios below 5%. Most of the misses come from inodes and directories, or from datafile and executable cold misses. Some workloads touch many inodes and directories. Small ID subcaches still capture 90% of the requests, but at low total cache miss ratios the uncaptured inodes dominate the total misses. The large ID subcache captures reuse for these inodes and directories. Large files generate an excessive number of cold misses, caused by low reuse rates and small requests generating many misses. The sequential subcache fetches large blocks, reducing the number of cold misses generated by large files. 128-Kbyte blocks reduce large file cold misses by more than 90%.

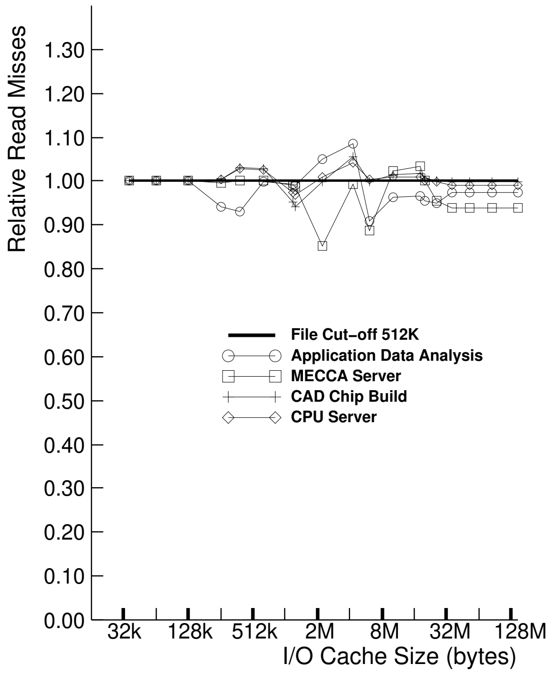


Figure 5.12: Cut-off sensitivity.

5.3.2 Lowering the File Cut-Off Size

The file cut-off directs larger files to the sequential subcache and smaller files to the temporal subcache. Lowering the file cut-off directs more files to the sequential subcache. With a larger fraction of the files, the sequential subcache potentially captures more sequential locality. When the sequential subcache has only a few entries, the additional files will probably conflict with each other. Fewer large files in the temporal subcache reduces the space needed to capture the working set, and lessens evictions from long sequential runs.

Figure 5.12 shows the minimal difference between a variable attribute cache scheme with a file cut-off of 128 Kbytes and one with a 512K cut-off. For workloads with a large sequential component, lowering the cut-off should increase contention in 64- to 128-Kbyte sequential subcaches. For the remaining workloads, segregating more large files to the sequential subcache improves the temporal cache performance. Neither effect significantly impacts the total performance. On average across the workloads, they balance out. For large caches, the lower cut-off reduces the read request misses, allowing the cache to capture more of the workload's sequential locality.

The minor differences between the behavior of the two file cut-off sizes suggests that a precise cut-off is not crucial. New files can be allocated to the temporal or sequential subcache based on their predicted size. And changes in file size need not make the file move between subcaches.

5.4 Write Expulsion Behavior

In a non-volatile cache, writes only generate disk requests when the cache evicts the data. With an early eviction policy, or with sufficient buffering, the operating system schedules disk writes when the disk is otherwise idle. Writes impact I/O performance indirectly, through resource contention. Excessive writes will increase the read request service time, since reads will wait more for write accesses to complete. I/O caches need to have a stable write performance.

Figure 5.13 shows the write expulsion characteristics for the MECCA Server and the CAD Chip Build workloads. Each block of write (or dirty) data evicted from the cache produces an independent disk write request. The *write expulsion ratio* measures the number of disk accesses relative to the number of write requests in the workload. The write cache behavior differs in several ways from the read behavior. (1) In the smallest cache, the variable scheme produces as many write expulsions as the Unix baseline scheme does. Inodes and directories compete for space in the ID subcache. Since most inodes are updated with file access time information, they generate writes when they are evicted. Once the variable scheme's ID subcache reaches a size twice as big as the Unix scheme's inode subcache, the write expulsions drop considerably. (2) Sequential subcaches typically increase the cache size needed to capture the write working set. At the working set capture size, the baseline scheme often requires fewer disk writes. (3) Large caches have few if any evictions, so the write expulsions approach zero. The Unix baseline has a fixed inode subcache that always generates inode writes.

Figure 5.14 shows the typical estimated byte transfer characteristics. The estimated byte transfer ratio assumes that the cache writes entire blocks to disk when it evicts data. Because the estimated write byte transfer ratio relates directly to the block size used by the cache and not to the locality captured by the cache, it makes a poor metric for evaluating write performance. For read requests, however, the estimate can help guide I/O cache configuration choices—high byte transfer ratios typically result from uncaptured reuse of larger files.

In the small cache region, the variable scheme increases the general subcache block size from 4 Kbytes to 16 Kbytes. This increases the transfer ratio by 50 to 100 percent, depending on workload. In the medium cache region, the large sequential subcache blocks continue to generate a high estimated byte transfer ratio. In the large cache region, the sequential subcache begins to capture the write working set, dropping the ratio. When the cache has captured the sequential working set, the estimated byte transfer ratios track reasonably well between the two schemes. In the middle and large cache regions, the behavior suggests that the sequential subcache evicts writes before they have the chance to write an entire block. Sequential reads usually consume the entire block before it is evicted. Write blocks require larger caches with longer occupancy times to capture sequential behavior.

Figure 5.15 compares the expulsion writes of the four sample workloads to the Unix baseline. Except for small caches and narrow cache regions corresponding to the workload working set

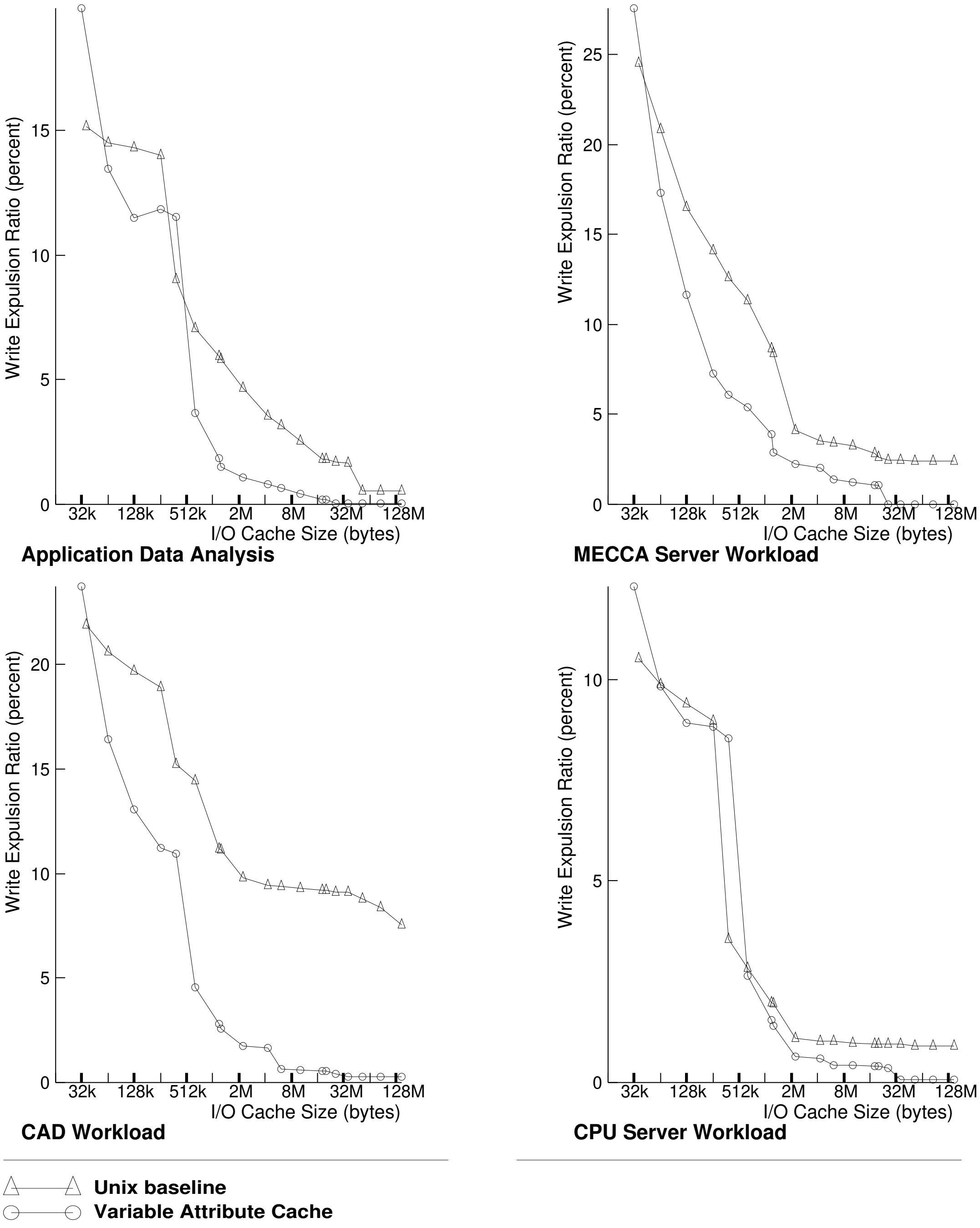


Figure 5.13: Write behavior

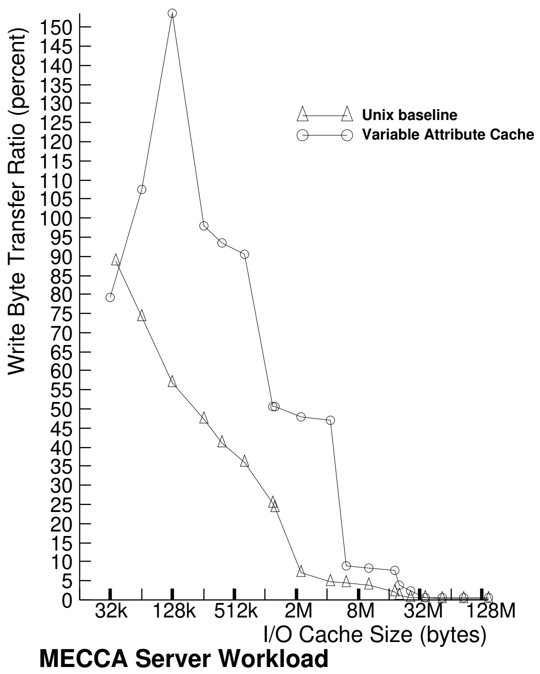


Figure 5.14: Write Byte Transfer Properties.

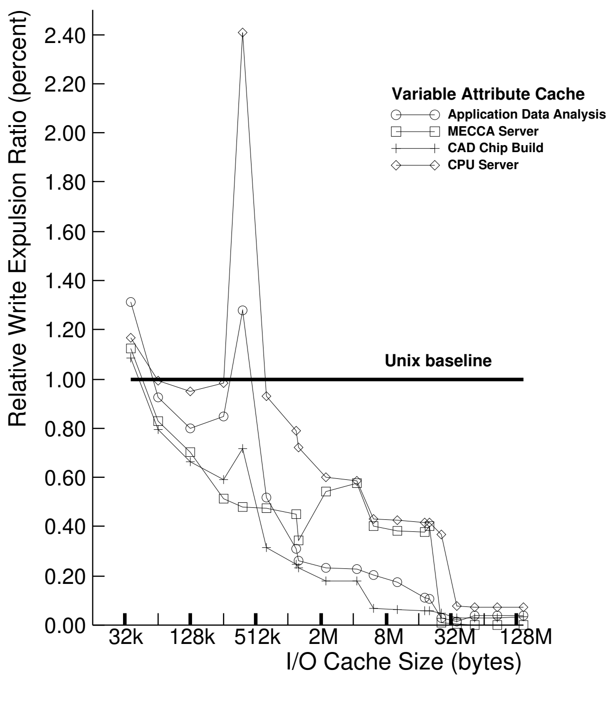


Figure 5.15: Relative write expulsions.

size, the variable scheme produces fewer write expulsions. Increasing the cache size increases the number of inodes in the cache thereby reducing the write expulsions to zero. The variable attribute cache scheme produces competitive write expulsion behavior.

5.5 Attribute Cache Compared with Other Schemes

5.5.1 Fixed Schemes

The variable attribute cache scheme outperforms other schemes by exploiting the distinct cache behavior of workload components, and by varying the cache scheme to best use the cache area for locality capture. Figure 5.16 shows a variety of fixed cache schemes in relationship to the variable attribute cache scheme. Comparing the read request miss performance shows the strengths and weaknesses of each cache scheme, as well as the cache range over which they perform the best.

Unix baseline (32K/128 inode, x/4K Unix)
Unix style (32K/128 inode, x/16K Unix)
Unix style (32K/128 inode, x/32K Unix)

For small caches, these three schemes cannot capture sufficient directory locality. Increasing the block size in the non-inode part of the Unix style I/O cache only exacerbates the situation. For medium and large caches, increasing the block size reduces the request misses, once they have captured the working set. The larger block size increases the cache size required to capture the working set. For many workloads, inadequate inode cache size severely limits the cache performance. These configurations do, however, compete favorably for a couple of workloads. The MECCA Server and Network Update workload have highly spatial requests to many medium-sized files, and relatively few ID requests. The larger block size captures this locality. Once the entire working set fits in the cache, other requests dominate.

Two-category (64K/128 ID, x/8K general)
Two-category (64K/128 ID, x/16K general)

Support for inode and directory caches reduces the miss ratio considerably for small caches, especially when compared with Unix style caches having identical block sizes. Larger block sizes reduce the request misses in workloads with much sequential locality. For workloads with more temporal locality, the larger blocks improve performance in the middle cache range, but not for larger caches. The 64K ID subcache shows performance benefits even for large caches.

Three-category (64K/128 ID, x/4K temporal, 64K/64K sequential)

The three-category attribute cache performs poorly in small caches, because it cannot capture any temporal locality beyond that of the inodes and directories, because it dedicates most of its space to the sequential and the ID subcaches. In the mid-range, it performs very well on workloads having

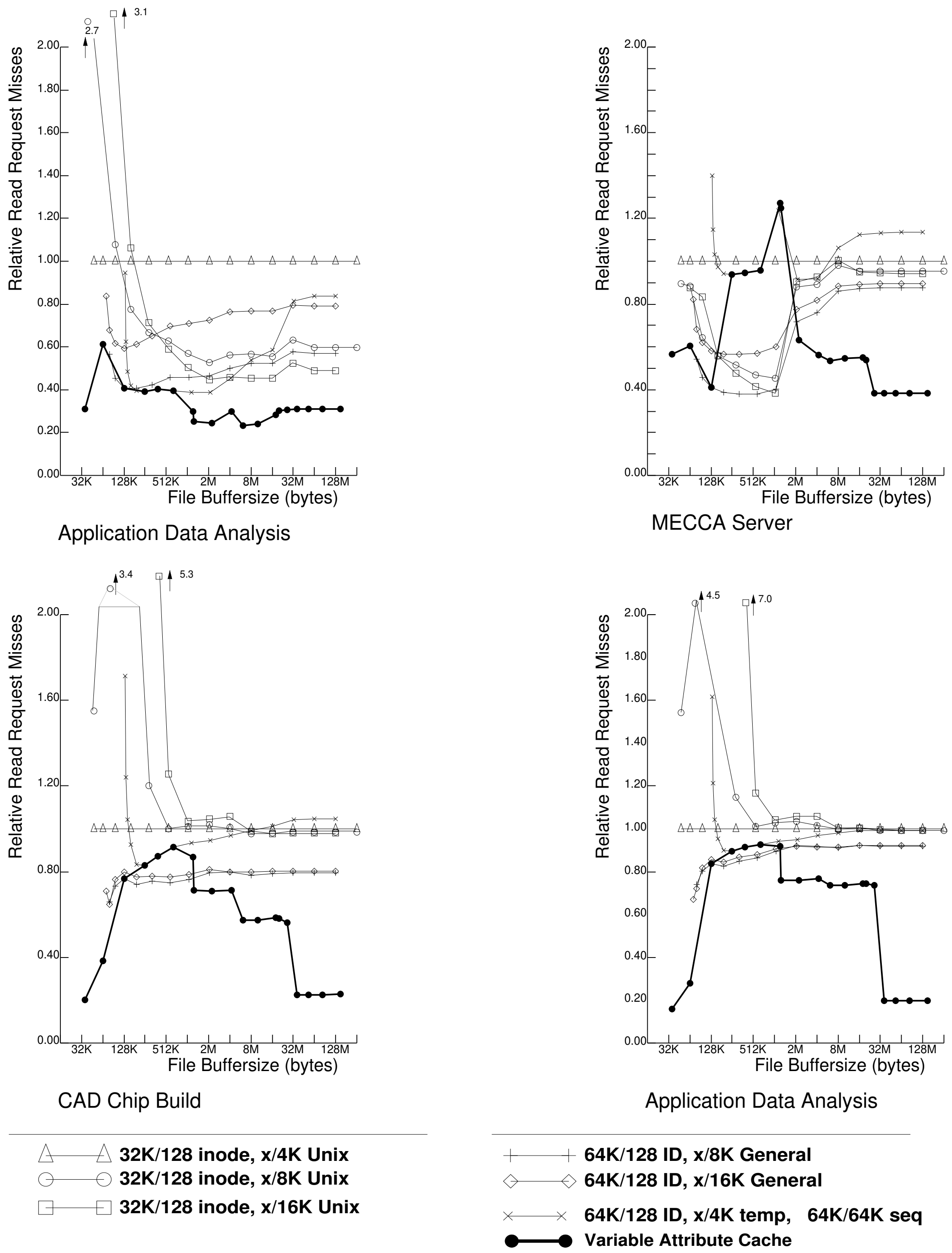


Figure 5.16: Fixed schemes compared with the variable attribute cache scheme.

large files with both a sequential and a temporal component.

For large caches, the scheme fails for many reasons. (1) The sequential cache is too small to prevent conflicts among large files, or to capture any reuse. (2) The temporal cache cannot capture any sequential behavior. (3) The ID subcache performs the same as the two-category scheme.

Variable attribute cache scheme

The variable attribute scheme significantly reduces read misses for both small and large caches by capturing ID locality and large-file sequential behavior. In the mid-size region, it reduces misses best when the workload has significant temporal locality and large-file sequential locality. Here, the temporal working set is protected from sequential sweep behavior, and the sequential cache explicitly captures sequential locality. The scheme fails to capture sequential locality for medium-sized files. These can add significantly to the request misses if the cache size is smaller than the working set capture size. A larger block size for the temporal cache might further reduce the read misses for mid-sized caches.

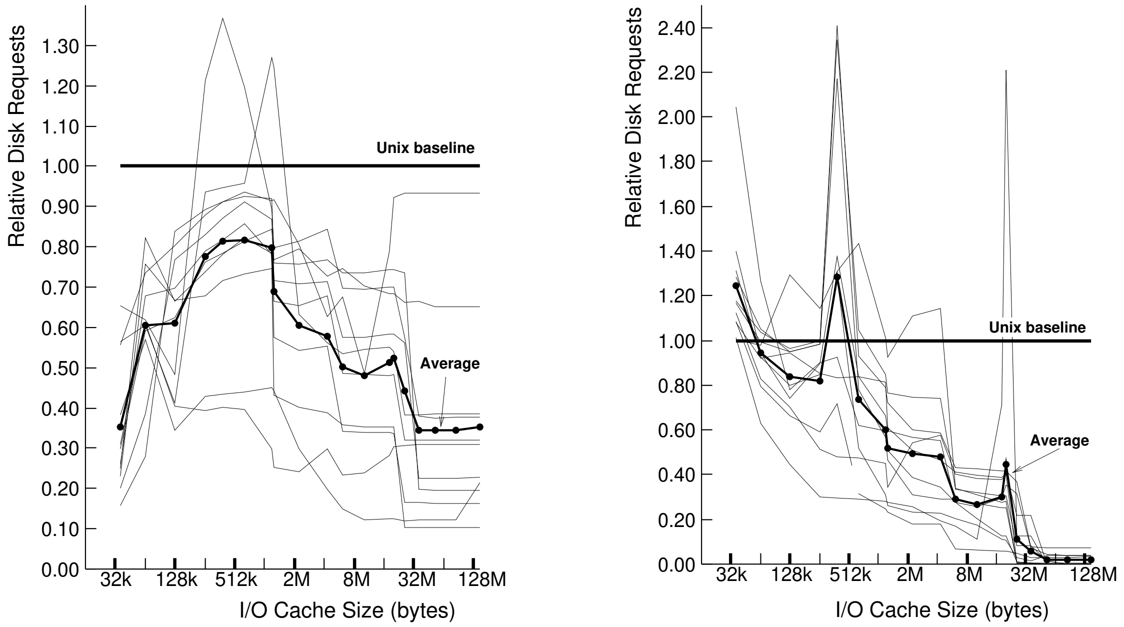
5.6 Total Performance Results

5.6.1 Disk Requests

Each read request miss and write expulsion generates a disk access. Figure 5.17 shows the number of disk accesses for the variable attribute cache scheme relative to the Unix baseline. The figure breaks read and write disk accesses down separately, because they impact system performance in different ways.

Disk accesses from read request misses determine how many times applications wait for I/O to complete, and the minimum number of context switches required to overlap computation with the I/O. The variable attribute cache scheme reduces the number of read disk accesses for almost all workloads over a full range of I/O cache sizes. Averaging over the workloads, it reduces the read accesses by at least 18% and as much as 66% depending on the cache size. The overall reduction averaged 48% in the small cache region, 28% in the middle cache region, and 58% in the large cache region.

Disk accesses from write expulsions increase the disk utilization and the probability that the disk will be busy when a read request miss occurs. Over most cache sizes, the variable attribute cache scheme does little to reduce the write expulsions. With the variable scheme, some workloads generate more writes than the baseline scheme, and others generate fewer writes. The write working set size is somewhat larger than the read working set. The variable attribute cache scheme frequently



Disk Accesses from Read Request Misses

Disk Accesses from Write Expulsions

Figure 5.17: Read and write disk requests performance for all workloads.

requires additional cache space to capture this working set. The spikes in the write expulsion graph correspond to these differences in working set capture.

Figure 5.18 sums the read and write disk access, showing the total disk accesses. For small caches, the writes increase the total relative disk requests, but for caches above 1 Mbyte they reduce the total relative requests. The total disk accesses reduce by an average of 38% in the small cache region, 31% in the middle cache region, and 66% in the large cache region.

5.6.2 Approximate Disk Service Time

The total disk time for a workload determines the amount of time individual applications wait on I/O, the amount of computation that must be available to fully utilize the CPU, and the amount of resources available for other simultaneously executing workloads.

Figure 5.19 shows the approximate relative disk service time for the workloads. The disk service time is calculated from the number of disk accesses and the number of bytes transferred. Recall

$$\text{read service time} = (\text{read request misses} \times \text{disk access overhead}) + (\text{read bytes transferred} \times \text{disk transfer rate})$$

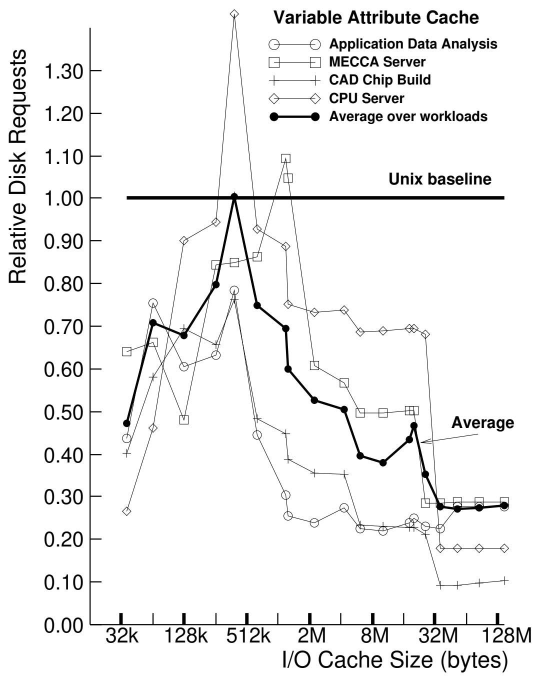
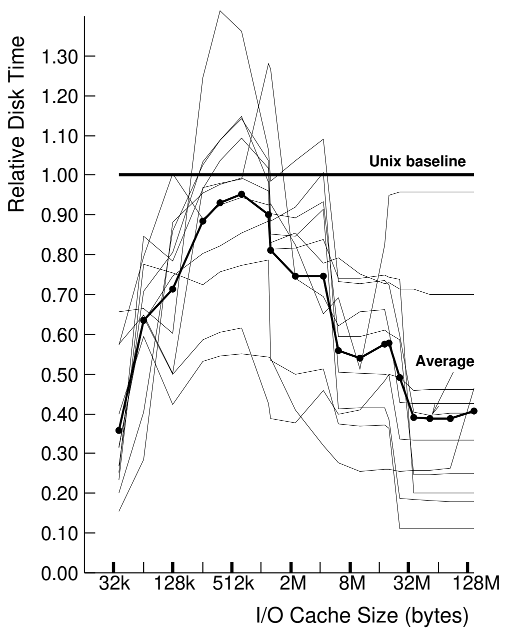
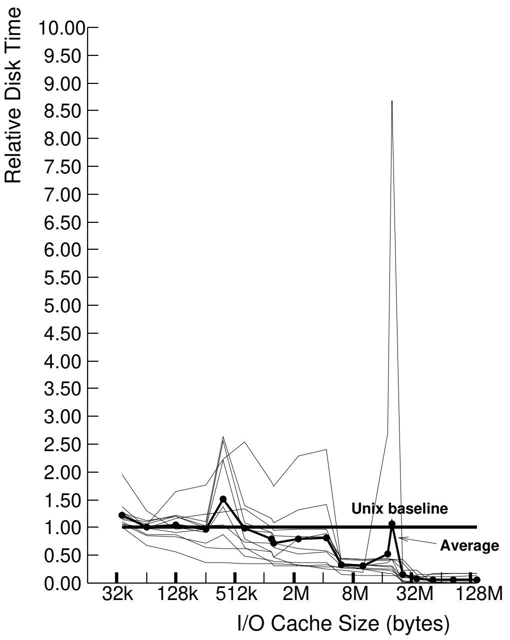


Figure 5.18: Total relative disk requests.



Read Time



Write Time

Figure 5.19: Estimated read and write disk access time performance for all workloads.

$$\text{write service time} = \text{write expulsions} \times \left[\text{disk access overhead} + (\text{cache block size} \times \text{disk transfer rate})\right]$$

assuming 20 ms overhead and a 0.5 ms/KB transfer rate. The read request misses and write expulsions come from direct simulation, but the number of bytes transferred on a miss or an expulsion must be estimated from the cache block size. This always overestimates the number of bytes transferred. The overestimation increases with larger cache blocks. The variable cache scheme thus tends to overestimate more, as its block size often exceeds that of the baseline scheme. Thus, the approximate disk times show the minimum expected improvement, or the maximum degradation in total disk time.

The read disk time directly affects application execution. The disk read time represents the total time that applications wait on disk reads. This is also the amount of time the operating system must overlap other process computation with I/O to prevent idle CPU cycles. On average, the variable attribute cache scheme reduces the average read disk time across all cache sizes. The reduction averaged 44% in the small region, 15% in the middle region, and 52% in the large region.

The write service times are comparable for the two schemes at most cache sizes. The variable scheme only significantly lowers the write service times for large caches. The variable scheme increased the average write service time by 7% in the small cache region, and decreased the service time by 6% in the middle region and 71% in the large region.

Figure 5.20 shows the combined read and write disk service times. The total service time determines the disk utilization. Lowering the service time required by individual workloads allows a disk to support a greater number of workstations and simultaneous workloads. For almost all cache sizes, the variable scheme reduces the total disk service time.

5.7 Conclusions

Attribute caches capitalize on the distinct access patterns of different file types and sizes. Each subcache uses a different block size to capture the locality of its attribute class. Allocating small files to small blocks increases the number of independent files stored in the cache. Allocating medium files to mid-sized blocks captures temporal locality with a smaller cache. And allocating large files to large blocks captures sequential locality. Matching the block size to the attribute class reduces unused cache space, increases cache utilization, and reduces the number of request misses.

The requests form large files do not fill up the entire cache, forcing out small files. The allows the cache to capture the working set of individual components even if the workload working set is

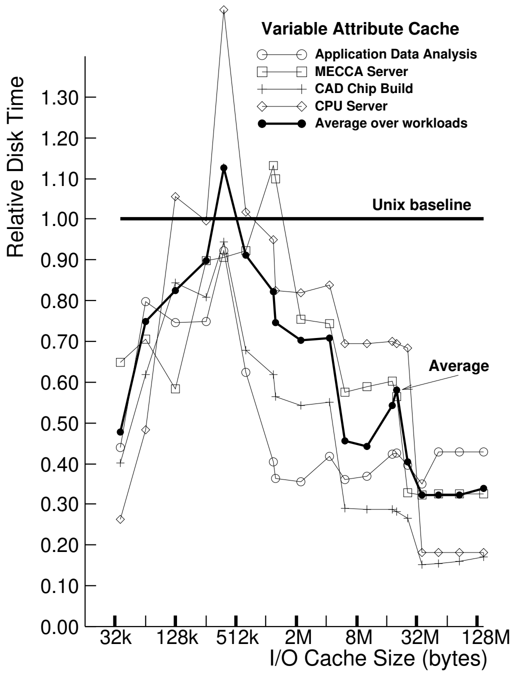


Figure 5.20: Total relative disk time.

larger than the cache. Capturing the working set of individual components significantly reduces the total request misses.

The subcache partitions must change with cache size to capture the greatest locality. By defining a different the cache partition based on cache size, the *variable attribute cache scheme* performs well over the full cache size range.

When compared with a Unix Style cache the variable attribute cache reduces the read disk requests by at least 18% and as much as 66% depending on cache size. Writes to disk decrease as the individual subcaches partition size increases. Large attribute caches have very few writes to disk. The reduction in read accesses reduces the total time required to service disk requests with any size cache, whereas, reductions in disk writes mainly reduce the disk service time for when the cache size is large.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

I/O cache s should make use of file types to improve their efficiency. Each file type has a distinct size distribution and reuse ratio. File type and size information can direct I/O requests to different parts of an attribute cache. Inodes and Directories account for 80% of all file requests; they are small and have a highly temporal access pattern. The attribute cache should allocate a significant amount of space to capturing these requests. This portion of the cache should have small blocks to effectively capture the temporal nature. Small executables and datafiles tend to have temporal access patterns, whereas large executables and datafiles tend to have sequential access patterns. A cache should direct large files to a sequential subcache, and smaller files to s temporal subcache. The sequential subcache has large blocks to capture sequential locality while the temporal subcache has small blocks to minimize unused space.

Small-sized caches, 128 Kbytes or less, should concentrate primarily on capturing the crucial inode and directory references, and use the remainder of their area to capture executable and datafile requests. Allocating half of the cache to an inode and directory subcache and half to a general I/O subcache reduces the average request miss ratio by 48% when compared to a standard Unix style cache.

Medium -sized caches, 256 Kbytes to 4 Mbytes, easily capture most of the inode and directory references. Now they must also capture the temporal portion of the executable and datafile requests. Separating the general I/O subcache into a large temporal and a smaller sequential portion allows each portion to be tailored to capture the appropriate locality. The temporal subcache captures references to many small files with small 4 Kbyte blocks while the sequential subcache captures sequential accesses with large 64 Kbytes blocks. Splitting the two caches also prevents large files from sweeping the cache. Unix style caches with the same miss ratios typically require a cache at least 8 times as large.

Large-sized caches, 8 Mbytes or more, easily capture both the inode and directory, and temporal executable and datafile references. To reduce the byte traffic, they must also capture reuse of the sequential executables and datafiles as well as any residual temporal locality. The sequential subcache occupies half of the cache to capture the reuse of the large files. Both the ID and temporal subcaches occupy the other half to capture residual temporal locality. The miss ratios are typically lower than any size Unix style cache; for the same size the read miss ratio is 66% percent lower.

6.2 Future Work

More work needs to be done in the general field of I/O cache performance. Many I/O cache implementations use ad hoc techniques to achieve acceptable system performance. Performance is measured either by the I/O time required to execute particular benchmarks, or by the number of user complaints. The system is tuned until few users complain about I/O performance. This might produce efficient I/O caches, but it does not extend the body of knowledge and understanding required to systematically design I/O caches to meet a targeted workload environment.

Specifically, the design space for using attributes to improve I/O cache performance remains vast. Much more work needs to be done to characterize and understand the I/O workload, including the interaction between applications and files. The variable attribute cache scheme is just one of many possible cache schemes based on the attributes of file type and file size. For any given cache size, several attribute caches configurations yield similar performance over a set of workloads. The variable attribute cache scheme, is just one set of attribute cache configurations.

6.2.1 Extended Workload Characterization

Extending the use of attributes beyond simple existing file properties requires an in-depth understanding of workload properties. While this work shows that file type and file size can effectively split the I/O request stream into distinct components, other file characteristics might do as well.

Attributes could be assigned by the application creating a file, or could possibly be determined dynamically. Either of these techniques requires understanding the relationship between applications and the files they use. Much work remains to understand and ultimately predict the cache behavior of individual files.

6.2.2 Attribute Cache Extensions

Framework for Directed Caches

The variable attribute cache scheme provides a framework for implementing either user or system directed caches. The subcaches provide a mechanism for capturing different file access and reuse. New attributes can allocate individual files or sets of files to appropriate subcaches.

System directed caches would involve monitoring file and application behavior. Based on historical behavior, the system could either assign attributes on the fly, or it could modify existing attributes. With proper monitoring tools, users could alter file attributes to improve individual workstation performance. More work remains to evaluate ways of monitoring systems to determine what sorts of behavior occur, and to understand what properties of files or applications effect total workload performance over long time periods.

In several of the eleven workloads studied, a few files determined the total performance of the workload. Remapping these files to a more appropriate subcache dramatically improved the cache performance.

Adaptive Schemes

Fixed subcache partitions invariably introduce inefficiencies. As with any split cache, the individual partitions are never equally utilized. Possible attribute cache schemes could use an adaptive partitioning scheme to vary the subcache sizes based on the workload behavior. This might further reduce the read request miss ratio, as well as prevent increases in the working set capture size for workloads that cannot use a particular subcache. In both the middle and large cache regions, this technique could make under-utilized subcaches available to the rest of the workload requests. Methods need to be developed for detecting low workload utilization of subcaches, and for growing and shrinking them appropriately.

6.2.3 Implementation Issues

Extend to Volatile or Limited Non-Volatile Cache

This work investigated only caches with non-volatile memory. Although many future I/O caches will probably have at least some non-volatile cache, few will use entirely non-volatile memory.

For many I/O cache systems, writes dominate the disk traffic. The I/O cache captures read requests, but caches writes only a short time before writing them to disk. The disk securely stores the data for future use, and keeps them against loss in the event of a system failure.

Write attributes could signal which data must be secured and which data may remain volatile. For example, many applications create temporary files that would be regenerated if a system failure occurred; these files could remain volatile. Current systems provide uniform data integrity for all user data. Workload characterization is needed to determine the potential write reduction from such a scheme, and to determine criteria for assigning write attributes.

For systems with some non-volatile I/O cache, workload write characterization could determine how to most effectively use or partition the non-volatile segment.

Dynamic I/O Cache Allocation

In the past few years, workstation operating systems have adopted a memory management policy that dynamically varies the I/O cache size based on the virtual memory requirements of the workload. The I/O cache grows when the workload memory requirements are low, and shrinks when they are high. Much research still needs to be done to evaluate the effectiveness of dynamically varying cache size. Applying an attribute cache scheme to such an environment would utilize the full cache range. More work needs to be done to smooth the transition from one cache size to another.

6.3 Other Areas

Attribute caches perform best in environments with diverse behavioral components. I/O related areas offer the most promise, simply because the required attribute information is minimal in both size and time when compared with the amount of data requested. Obvious areas to apply attribute caches include multi-media and network environments.

True multi-media environments incorporate pictures, video, and sound along with traditional text, data, and programs. Video requires sequentially reading large files from disk or over the network. Continuous video requires prefetching and buffering. Prefetching large rather than small blocks reduces the overall disk utilization by removing the overhead associated with multiple accesses; this allows the I/O system to support up to twice as many independent video streams [RW94a]. Video differs from other large sequentially accessed data as it has no expected reuse and it requires prefetching since misses are not tolerated. In this environment, a video and audio subcache might smoothly incorporate proper I/O buffering for the two.

Information access through the network has increased dramatically. Sites all around the world are making documents, pictures and sound readily available. To support continued growth, especially into the business sector, requires more efficient cache mechanisms. In addition to distinct reuse and size distributions, network data has a range of integrity requirements. Some data requires security, some requires accuracy, and some requires neither. Credit card transactions require security. Hotel vacancy notices need to be up-to-date. But restaurant menus or product catalogues may be a bit out of date, and freely accessible. Making the national information infrastructure a medium for commerce will require much more sophisticated cache mechanisms. The attribute cache principles can be applied to network caches.

Bibliography

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[BAD+92] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Fifth Int. Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27 of *SIGPLAN Notices*, pages 10–22, Boston, MA, Sept 1992. SIGPLAN Notices, ACM.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM symposium on Operating Systems Principles*, SIGOPS, Special Interest Group on Operating Systems, pages 198–212, Pacific Grove, CA, October 1991. SIGOPS, ACM.

[BKLW90] Anita Borg, R. E. Kessler, Georgia Lazana, and David Wall. Long address traces from RISC machines: Generation and analysis. In *The 17th Annual International Symposium on Computer Architecture*, pages 270–279. IEEE Computer Society Press, May 1990.

[Bor92] Anita Borg. MECCA: a message-enabled communication and information system. Technical Report Technical Note TN-10, Digital Network Systems Laboratory, November 1992.

[BRW89] Andrew Braunstein, Mark Riley, and John Wilkes. Improving the efficiency of UNIX file buffer caches. In *Symposium on Operating System Principles 12, 1989*, pages 71–82, 1989.

[CBJ92] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of TLB performance. In *The 19th Annual International Symposium on Computer Architecture*, pages 114–123. IEEE Computer Society Press, May 1992.

[CGK+88] Peter Chen, Garth Gibson, Randy H. Katz, David A. Patterson, and Martin Schulze. Two papers on RAIDs. Technical Report UCB/CSD 88/479, University of California, Berkeley, December 1988.

[Che89] Peter M. Chen. An evaluation of redundant arrays of disks using and Amdahl 5890. Technical Report UCB/CSD 89/506, University of California, Berkeley, 1989.

[Che92] Peter Ming-Chien Chen. *Input/Output Performance Evaluation: Self-Scaling Benchmarks, Predicted Performance*. PhD thesis, University of California, Berkeley, Nov 1992.

[Cop] Digital Equipment Coproration. Configuration file maintenance. In *USENIX System and Network Management*, volume 2.

[CP92] Peter M. Chen and David A. Patterson. A new approach to I/O benchmarks - adaptive evaluation, predictive performance. Technical Report UCB/CSD 92/679, University of California, Berkeley, March 1992.

[DM92] Jeremy Dion and Louis Monier. Design tools for BIPS-0. Technical Report Technical Note TN-32, Digital Western Research Laboratory, December 1992.

[Fly94] Michael J. Flynn. *Studies in Processor Design*, chapter Memory System Design. Review copy, 1994.

[GHK+89] Garth A. Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz, and David A. Patterson. Failure correction techniques for large disk arrays. In *Third Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston Massachusetts, April 1989. ACM, ACM Press.

[GMS88] Hector Garcia-Molina and K. Salem. Disk striping. Technical Report Computer Research Report, Princeton University, 1988.

[Gra91] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, 1991.

[Gro89] C. P. Grossman. Evolution of the DASD storage control. *IBM Systems Journal*, 28(2):196–226, 1989.

[GWHP93] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk subsystem load balancing: Disk striping vs. convential data placement. In *26th Proceedings of the Hawaii International Conference on System Sciences*, volume 1 : Architecture and Biotechnologies, pages 40–49. IEEE Computer Society, IEEE Computer Society Press, January 1993.

[GWHP94] Gregory R. Ganer, Bruce L. Worthington, Robert Y. Hou, and Yale N. Patt. Disk arrays: High-performance, high reliability storage subsystems. *Computer*, 27(3):30–37, March 1994.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, chapter Chapter 9: Input/Output, pages 498–568. Morgan Kaughmann Publishers, Inc., 1990.

[Kim86] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[KLW94] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, March 1994.

[KOP+89] Randy H. Katz, John K. Ousterhout, David A. Patterson, Peter Chen, Ann Chervenak, Rich Drews, Garth Gibson, Ed Lee, Ken Lutz, Ethan Miller, and Mendal Rosenblum. A project on high performance I/O subsystems. *Computer Architecture News*, 17(5):24–31, September 1989.

[KT87] Michelle Y. Kim and Asser N. Tantawi. Asychronous disk interleaving. Technical Report RC 12497 (No. 56190), IBM Thomas J Watson Research Center, January 1987.

[KT91] Michelle Y. Kim and Asser N. Tantawi. Asychronous disk interleaving: Approximating access delays. *IEEE Transactions on Computers*, 20(7):801–810, July 1991.

[LMKQ90] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.3BSD UNIX operating system*. Addison-Wesley Publishing Company, 1990.

[Lyn72] W. C. Lynch. Do disk arms move? *Perform. Eval. Review*, 1:3–16, Dec 1972.

[LZCZ86] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenpoel. File access performance of diskless workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[MB91] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Fourth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, volume 19 of *SIGARCH Computer Architecture News*, pages 75–84, Santa Clara, California, April 1991. ACM.

[Men87] Jai Menon. The IBM 3990 model 3 disk chace. Technical Report RC 5994 (No. 59593), IBM Thomas J Watson Research Center, December 1987.

[MJLF84] Marshall K. McKusick, William N. Joy, Samual J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MK89] Y. Manolopoulos and J.G. Kollias. Performance of a two-headed disk system when serving database queries under the scan policy. *ACM Transactions on Database Systems*, 14(3):425–442, September 1989.

[MSG89] Richard R. Muntz, Edmundo De Souza E Silva, and Ambjuj Goyal. Bounding availability of repairable computer systems. *IEEE Transactions on Computers*, 38(12):1714–1723, December 1989.

[Nel88] Michael Nelson. Physical memory management in a network operating system. PhD. Thesis UCB/CSD 88/471, University of California, Berkeley, 1988.

[Nel90] Michael N. Nelson. Virtual memory vs. the file system. Technical Report Research Report 90/4, Digital Western Research Laboratory, October 1990.

[Ng88] Spencer Ng. Some design issues of disk arrays. Technical Report RC 6590 (No. 63550), IBM Thomas J Watson Research Center, December 1988.

[Ng91] Spencer W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, 40(1):22–30, January 1991.

[NWO87] Michael Nelson, Brent Welch, and John Ousterhout. Caching in the Sprite network file system. Technical Report UCB/CSD 87/359, University of California, Berkeley, February 1987.

[OD88] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: A case for log-structured file systems. Technical Report UCB/CSD 88/467, University of California, Berkeley, October 1988.

[ODCH$^+$85] John Ousterhout, Herve Da Costa, David Harrison, JohnA. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. Technical Report UCB/CSD 85/230, University of California, Berkeley, April 1985.

[Ols89] Thomas M. Olson. Disk array performance in a random IO environment. *Computer Architecture News*, 17(5):71–77, September 1989.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). Technical Report UCB/CSD 88/477, University of California, Berkeley, December 1988.

[Quo94] Russell W. Quong. Expected I-cache miss rates via the gap model. In *The 21th Annual International Symposium on Computer Architecture*, pages 372–383. IEEE Computer Society, IEEE Computer Society Press, April 1994.

[RB89a] A. L. Narasimha Reddy and Prithviraj Banerjee. An evaluation of multiple-disk I/O systems. *IEEE Transactions on Computers*, 38(12):1680–1690, December 1989.

[RB89b] A.L.Narasimha Reddy and Prithviraj Banerjee. A study of parallel disk organizations. *Computer Architecture News*, 17(5):40–47, September 1989.

[Red92a] A. L. Narasimha Reddy. Reads and writes: When I/Os aren't quite the same. In *25th Proceedings of the Hawaii International Conference on System Sciences*, volume 1 : Architecture and Emerging Technologies, pages 84–92. IEEE Computer Society Press, January 1992.

[Red92b] A. L. Narasimha Reddy. A study of I/O system organizations. In *The 19th Annual International Symposium on Computer Architecture*, pages 308–317. IEEE Computer Society Press, May 1992.

[RF92] Kathy J. Richardson and Michael J. Flynn. TIME: Tools for input/output and memory evaluation. In *25th Proceedings of the Hawaii International Conference on System Sciences*, volume 1: Architecture and Emerging Technologies, pages 58–66. IEEE Computer Society, IEEE Computer Society Press, January 1992.

[RF93] Kathy J. Richardson and Michael J. Flynn. Strategies to improve I/O cache performance. In *26th Proceedings of the Hawaii International Conference on System Sciences*, volume 1 : Architecture and Biotechnologies. IEEE Computer Society, IEEE Computer Society Press, January 1993.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Symposium on Operating System Principles 13, 1991*, pages 1–15, 1991.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[RW94a] A. L. Narasimha Reddy and James C. Wyllie. I/O issues in multimedia system. *Computer*, 27(3):69–74, March 1994.

[RW94b] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–29, March 1994.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. Technical Report Research Report 94/2, Digital Western Research Laboratory, March 1994.

[Smi81] A. J. Smith. Bibliography of file system and input/output optimization and related topics. *Opererating System Review*, 15(4):39–54, October 1981.

[Smi85] Alan Jay Smith. Disk cache - miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[SO92] Ken W. Shirriff and John K. Ousterhout. A trace driven analysis of name and attribute caching in a distributed system. In *USENIX Winter 1992 Technical Conference*, pages 315–331, San Francisco, CA, Jan 1992. Usenix Association.

[STH83] R. A. Scranton, D. A. Thompson, and D. W. Hunter. The access time myth. *IBM Res. Rep. RC 10197*, Sept 1983.

[Sto86] M. Stonebraker. *The Ingres Papers*. Addison-Wesley, 1986.

[SV68] D. A. Stevenson and W. H. Vermillion. Core storage as a slave memory for disk storage devices. In *Proceedings of the INFORMATION PROCESSING '68 Conference*, pages F86–F91. IFIP, 1968.

[Tho87] James Gordon Thompson. *Efficient Analysis of Caching Systems*. PhD thesis, University of California, Berkeley, Oct 1987.

[TS87] James G. Thompson and Alan Jay Smith. Efficient (stack) algorithms for analysis of write-back and sector memories. Technical Report UCB/CSD 87/358, University of California, Berkeley, June 1987.

[WO89] Barry Wolman and Thomas M. Olson. IOBENCH: A system independent IO benchmark. *Computer Architecture News*, 17(5):55–70, September 1989.