APRIL 1995

WRL
Technical Note TN-48

Attribute caches

Kathy J. Richardson and Michael J. Flynn

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and technical notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`JOVE::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Reports and technical notes may also be accessed via the World Wide Web: `http://www.research.digital.com/wrl/home.html`.

Attribute Caches

Kathy J. Richardson and Michael J. Flynn

April 1995

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Abstract

Workloads generate a variety of disk I/O requests to access file information, execute programs, and perform computation. I/O caches capture many of these requests, reducing execution time, providing high I/O rates, and decreasing the disk bandwidth needed by each workload. Workload component characterization shows file type and size information can be used to group requests with similar reuse rates and access patterns.

Attribute caches have various partitions to capture the statistically distinct component behavior of the workload, each tailored to cache files with certain properties or attributes. 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.

The paper develops an attribute cache scheme to improve total I/O cache performance. The scheme relies on workload characteristics to determine the appropriate cache configuration for a given cache size. For a set of eleven measured workloads, it 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.

1 Introduction

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.

Several studies characterize the I/O workload and show that I/O caches reduce I/O traffic in a distributed file system and provide reasonable system performance [9, 2, 8, 12]. The characterizations show 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. The individual data request size is typically determined by the system libraries so long runs are made up of many individual requests.

In Unix systems disk files can be classified as accesses to inodes, directories, datafiles or executables. Each has distinct cache behavior [12]. Inodes and directories are small and highly reused files, while 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 infrequently reused. Properly used, file type and file size information improves cache performance.

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 each request has a 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 [14].

The remainder of this paper focuses on attribute caches. Section2 describes the workloads and the attribute cache terminology. Section3 examines attribute cache design trade-offs in various regions of operation. Section 4 describes and evaluates the performance of an attribute cache scheme that varies with cache size to substantially improve I/O cache performance. Section 5 concludes the paper. Two Appendices describe the trace collection and simulation techniques and the workload component behavior in I/O caches.

2 Background

2.1 Workloads

Kernel Build trace is a configuration and compile of the ULTRIX kernel (2 hours).

Ingres Transactions performs 10,000 banking debit and credit transactions on an Ingres database. The transactions are entirely random, and there are no complicated searches.

Application Data Analysis manipulates a series of traces, simulates caches and displays cache results.

Software Development 1 (8 to 5) develops and tests the ATOM simulation system. 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 (4 days).

Document Preparation records work on a technical report which includes text processing, editing, simulation, drawing, and data manipulation (6 hours).

Mecca Development (24 hr.) records the development and testing of a centralized e-mail system.

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

CAD: Chip Build traces the construction of a CPU layout from a high level description. The workload constructed the layout of the chip and 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 net with new information.

Compute Server (24 hr.) consists of batch simulations running on a machine with an idle console.

Figure 1: Eleven traced workload environments

I/O cache performance evaluation requires collecting I/O traces or installing instrumentation. Traces are needed by all designers to simulate and directly compare various cache alternatives and understand workload locality. Generating traces is difficult, and the type of tracing or instrumentation determines which I/O characteristics are visible [15]. Some I/O cache studies have used disk requests to study cache performance [15, 10, 11], while others have used operating system traces of file system activity [9, 8, 3, 13, 6, 12].

Figure 1 describes the eleven workloads evaluated in this paper. Most of the workloads cover several days of user activity. These long traces capture significant I/O activity and show the interaction of the many large and small files that comprise a workload.

The workload traces cover a wide range of user applications and types of work. Although not necessarily typical of heavy commercial use, such as large database systems, they represent many engineering development and office environments. Appendix A discusses the trace collection methodology and Appendix B shows workload component behavior. More detailed descriptions of the workloads are included in [12].

2.2 Attribute Cache Framework

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

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.

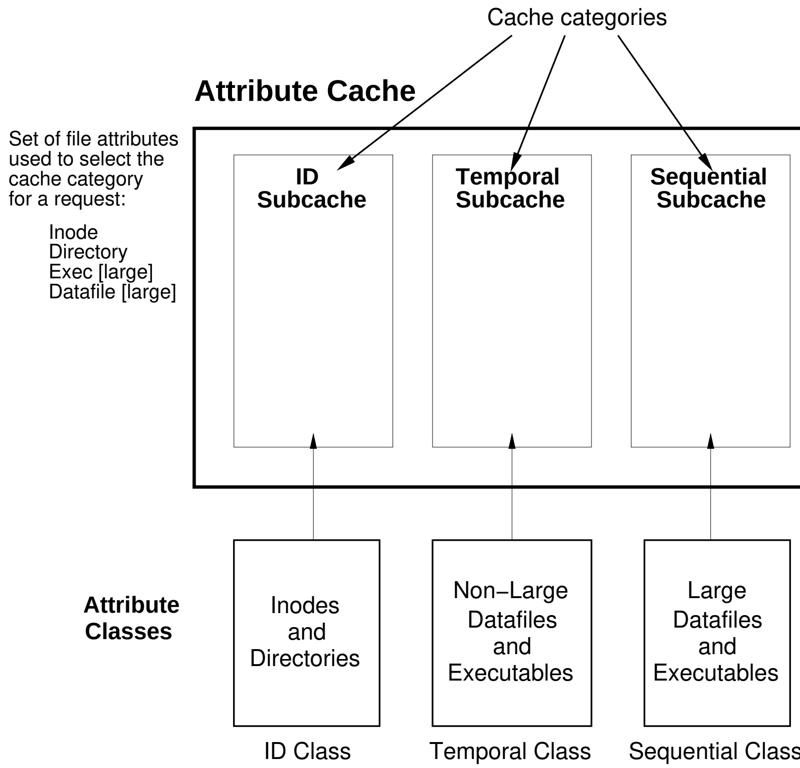


Figure 2: Anatomy of an attribute cache.

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.

Figure 2 shows the attributes, cache categories and attribute classes used by one attribute cache. This attribute cache has three separate subcaches, one for each category of expected cache behavior. 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.

3 Attribute I/O Caches

This section describes the design trade-offs and performance for possible attribute cache schemes. These attribute cache schemes use 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 best partitioning varies with the size of the cache.

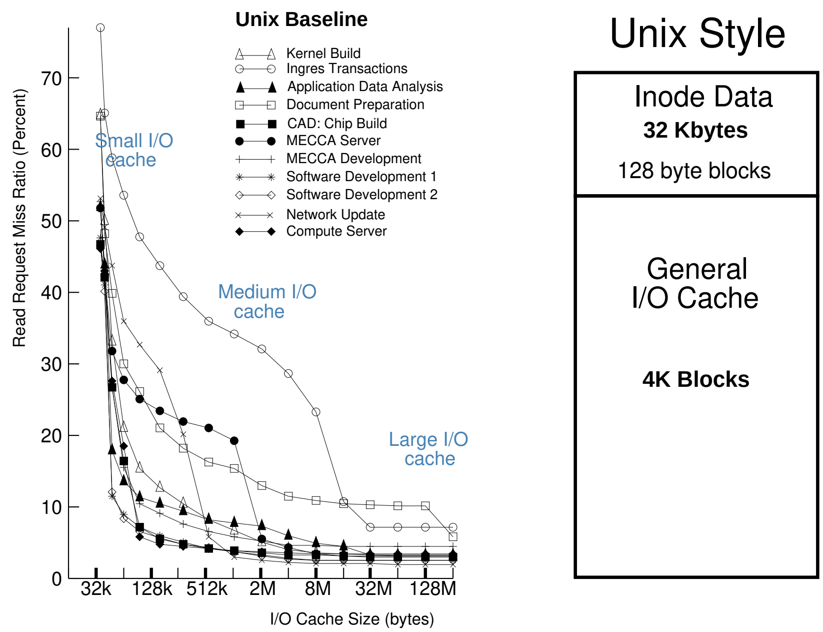


Figure 3: Unix style cache read request miss ratio.

Allocating resources to cache partitions 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. A 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.

3.1 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 3 shows the baseline read request miss ratio for all the workloads. All subcaches partitions are fully associative allocate on write, and perform LRU replacement. For write data security the caches are assumed to be non-volatile.

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

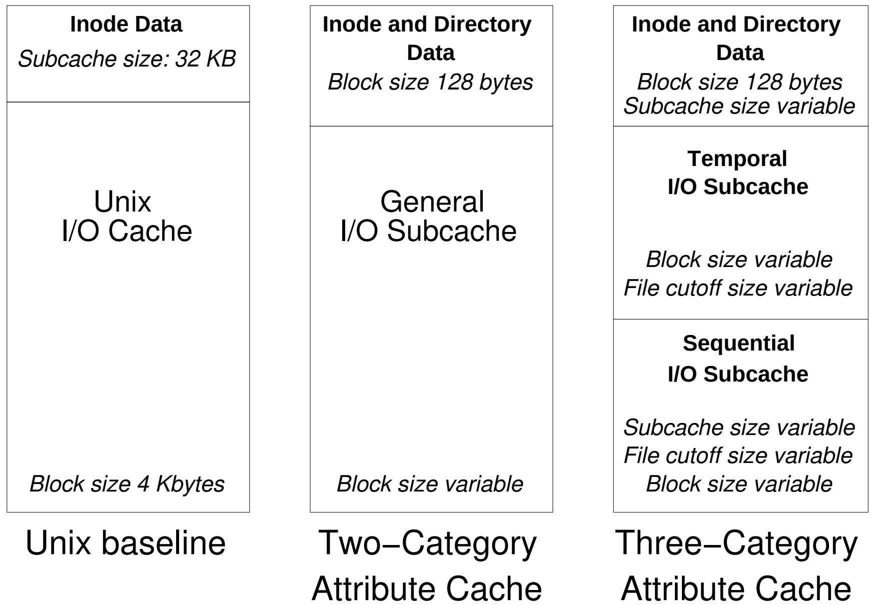


Figure 4: Logical configurations for attribute I/O caches.

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.

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.

3.2 Attribute Cache Design Parameters

Figure 4 shows the cache configurations used to demonstrate the usefulness of attributes. 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 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:

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.

3.3 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. 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 sets typical of inodes and directories.

Datafiles and executables exhibit some locality that a small cache can easily capture. The datafile and executable request miss ratio of Figure 15 shows 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 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

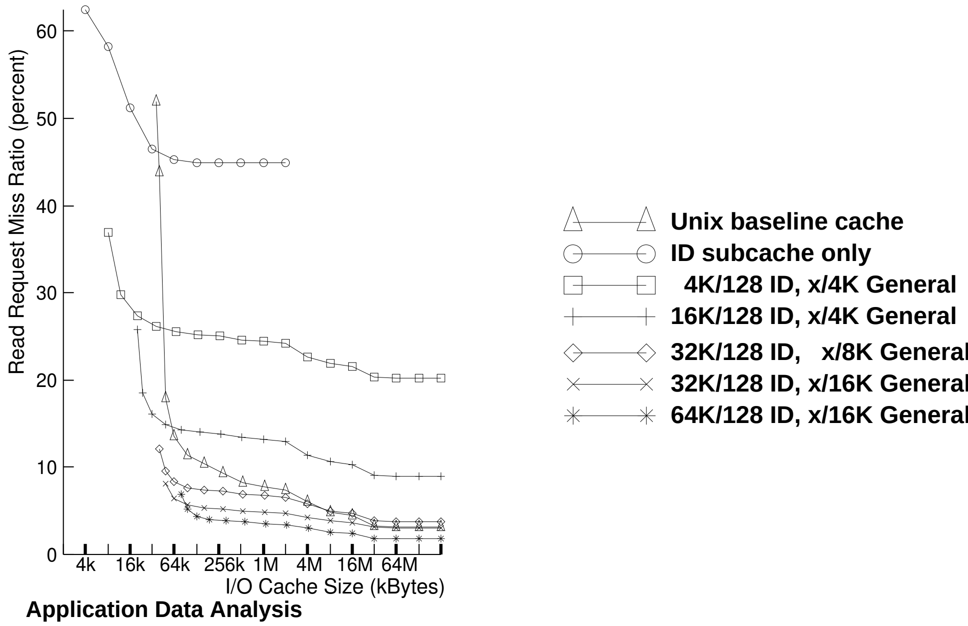


Figure 5: Small cache options.

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. As the cache size increases, so should the inode and directory partition, and the block size of the general subcache.

3.4 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. All medium I/O caches are large enough to capture the inode and directory working set. Medium cache designs need to concentrate on reducing the datafile and executable request misses without increasing the cache size required to capture the

Datafiles and executables have both temporal and sequential 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

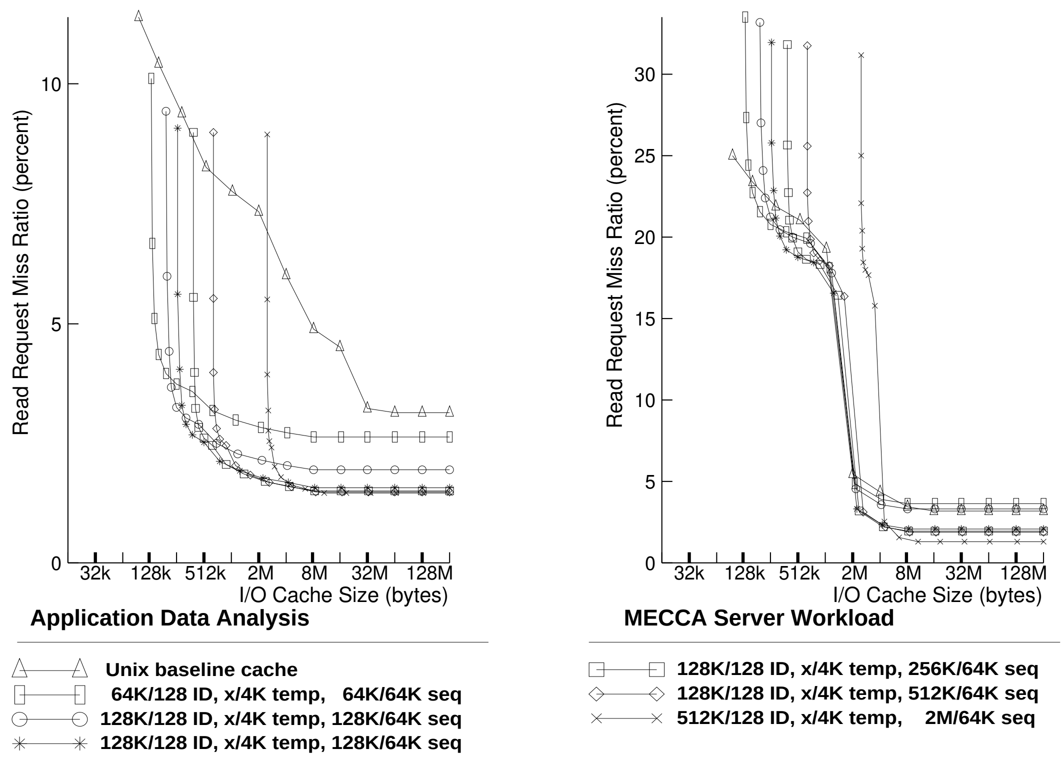


Figure 6: Medium cache options

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.

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.

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 6 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 that 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, however, use the sequential subcache to some extent and show lower read request miss ratios in this middle range.

3.5 Large I/O Cache Region

In the large cache size region, the prime objective is to capture the entire workload with as few request misses as possible. There is adequate cache to capture the working set, and to capture reuse of the large files. Neglecting the large file reuse does not always significantly impact the total requests (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 data transfer time low.

Figure 7 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. In general, little gain accrues from increasing the ID cache beyond 128K, unless goes to at least 1M. A couple of workloads see a significant reduction when the ID subcache increases to 1M, and a few more when it goes to 4M. This results from application that read all the meta-data on a disk, suggesting that an alternative mechanism should really be used to support these applications.

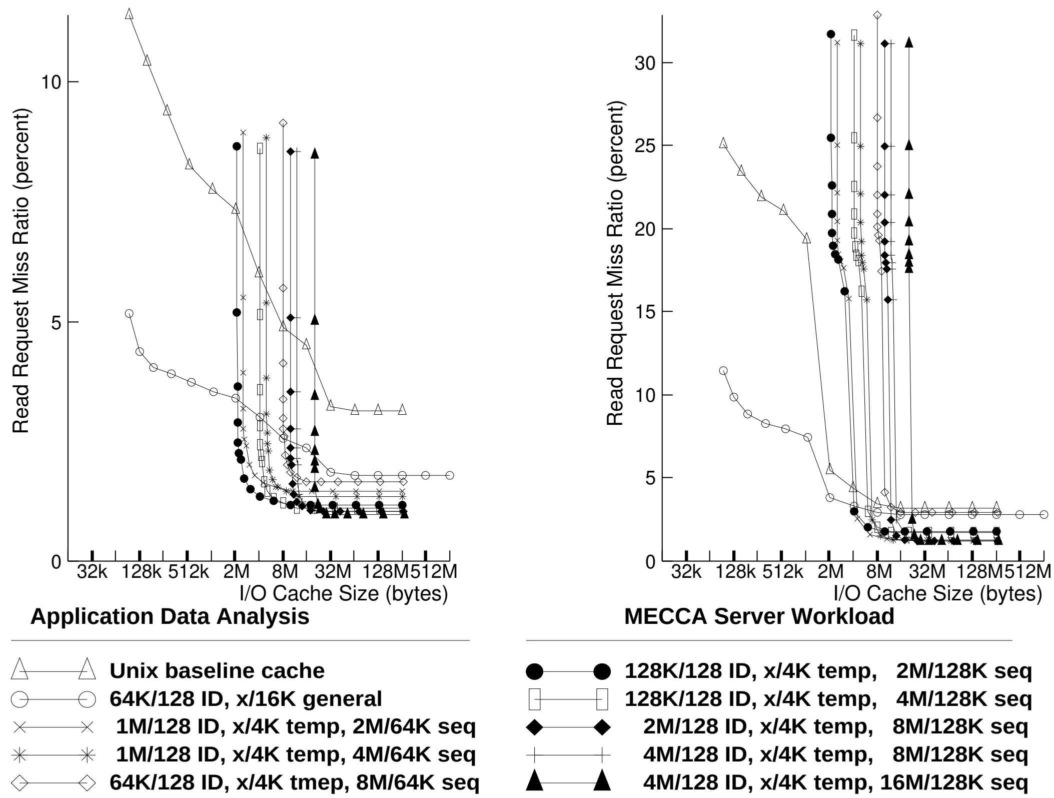


Figure 7: Large cache options

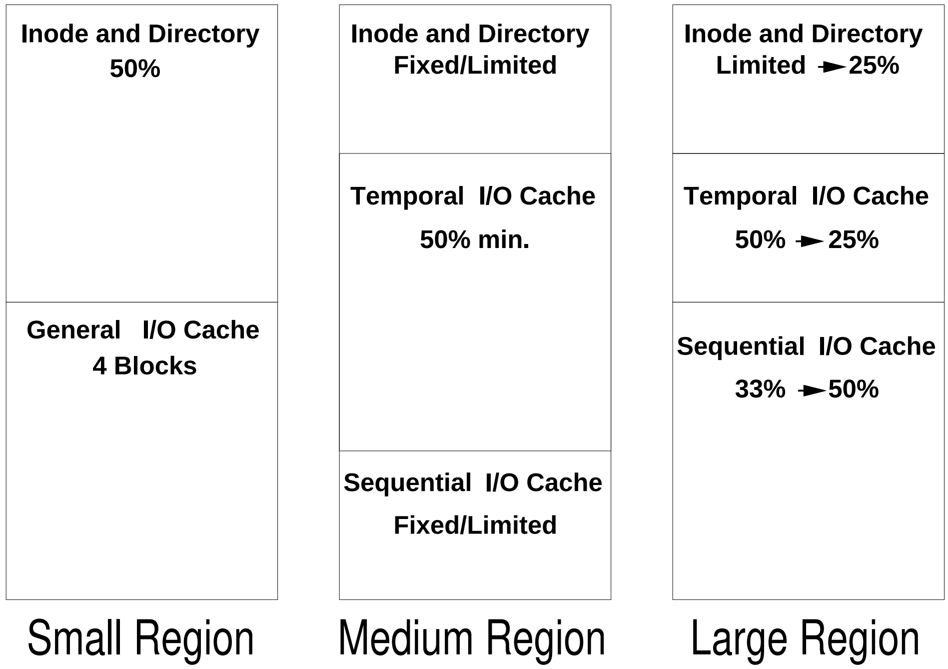


Figure 8: Variable attribute cache configurations for each size region.

4 Variable Attribute Cache Scheme

The *variable attribute cache scheme* uses attributes to substantially reduce 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.

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 8 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

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 1: Variable Attribute Cache Scheme.

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 to 25% of the cache area.

Table 1 describes the exact cache configurations used in each region by the variable attribute cache scheme.

4.1 Read Request Behavior

Figure 9 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 10 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 for I/O requests to complete.

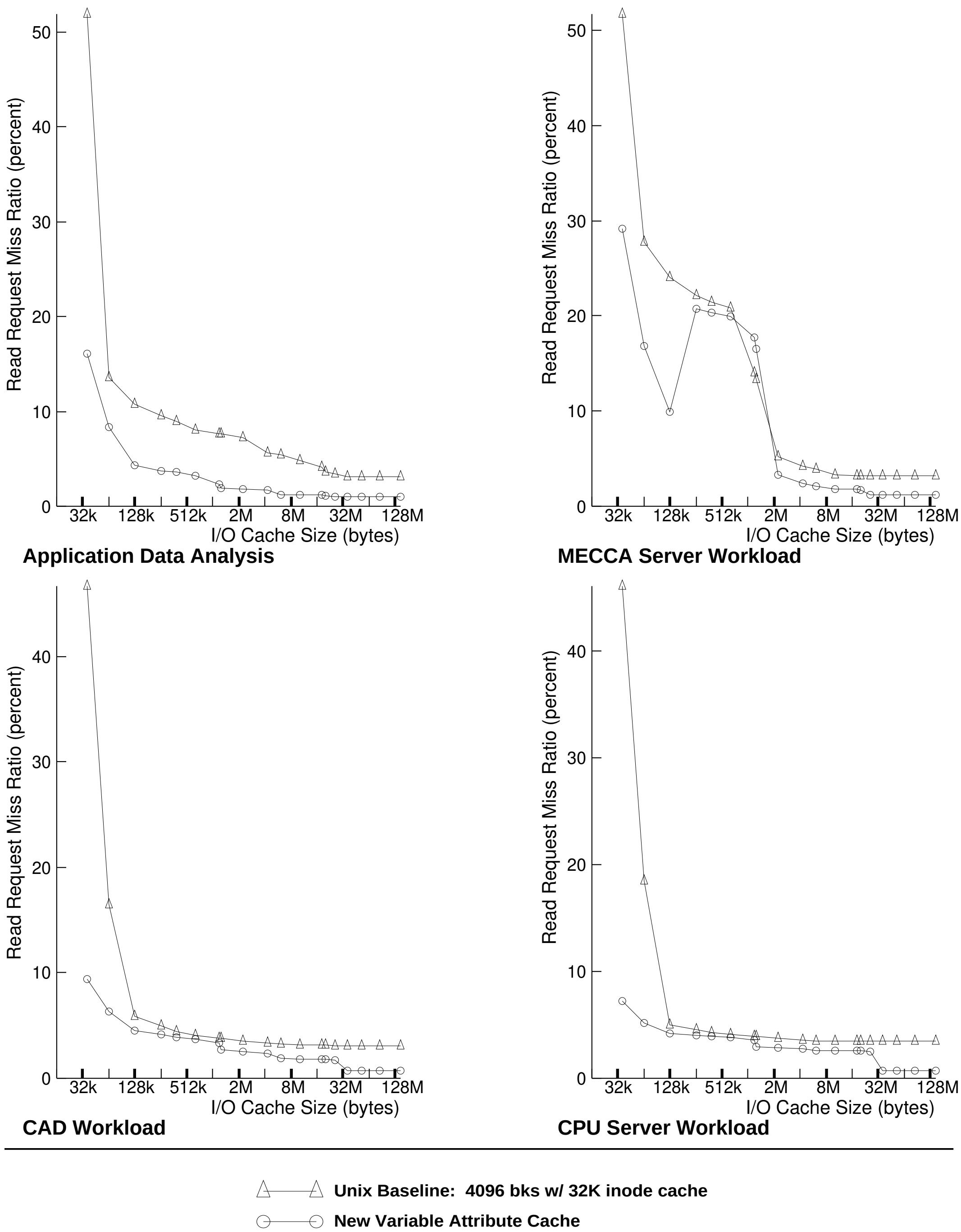


Figure 9: Attribute cache scheme request miss ratios.

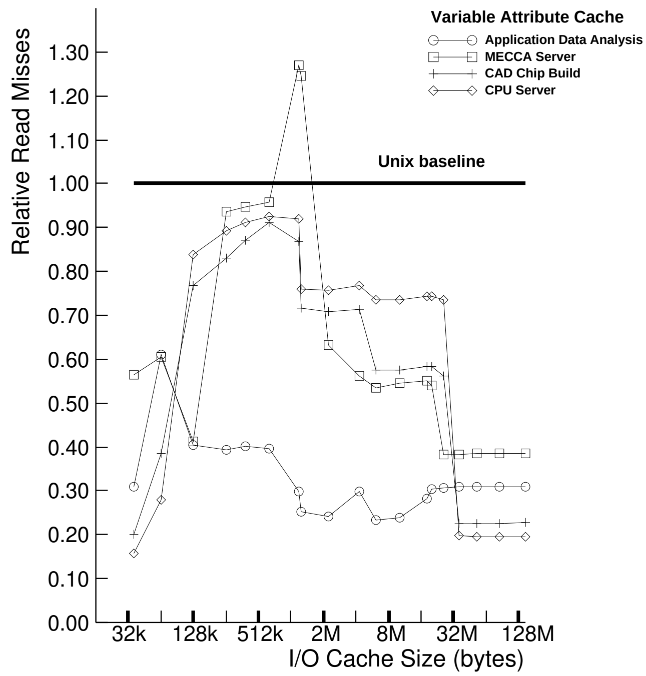


Figure 10: Relative read misses for the four sample workloads.

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 10 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%.

4.2 Write Expulsion Behavior

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. 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 [10] 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 [2]. 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.

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 11 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 11 also 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 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.

4.3 Variable Attribute Cache Compared with Other 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 12 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

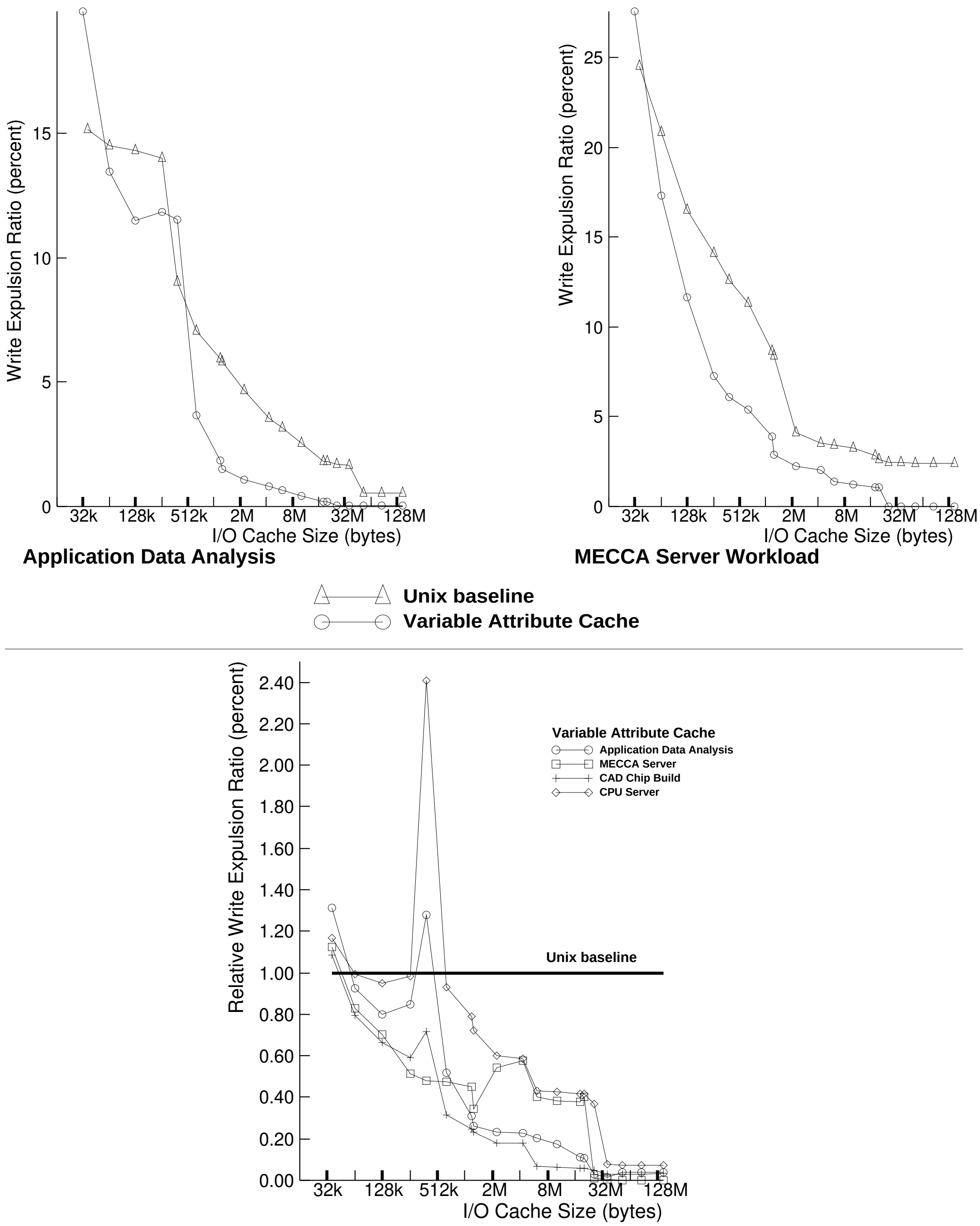


Figure 11: Write behavior

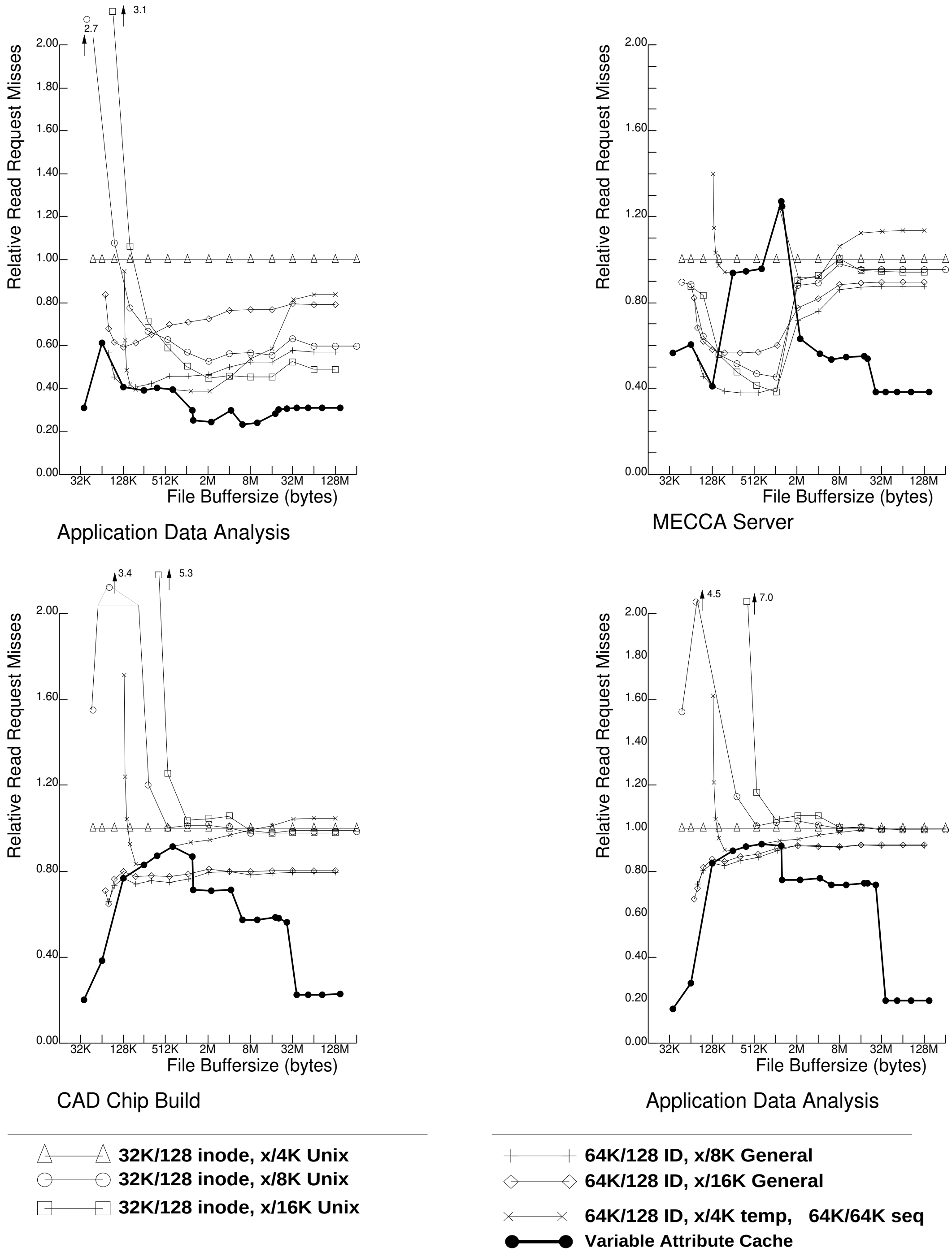


Figure 12: Fixed schemes compared with the variable attribute cache scheme.

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 large files with both a sequential and a temporal component.

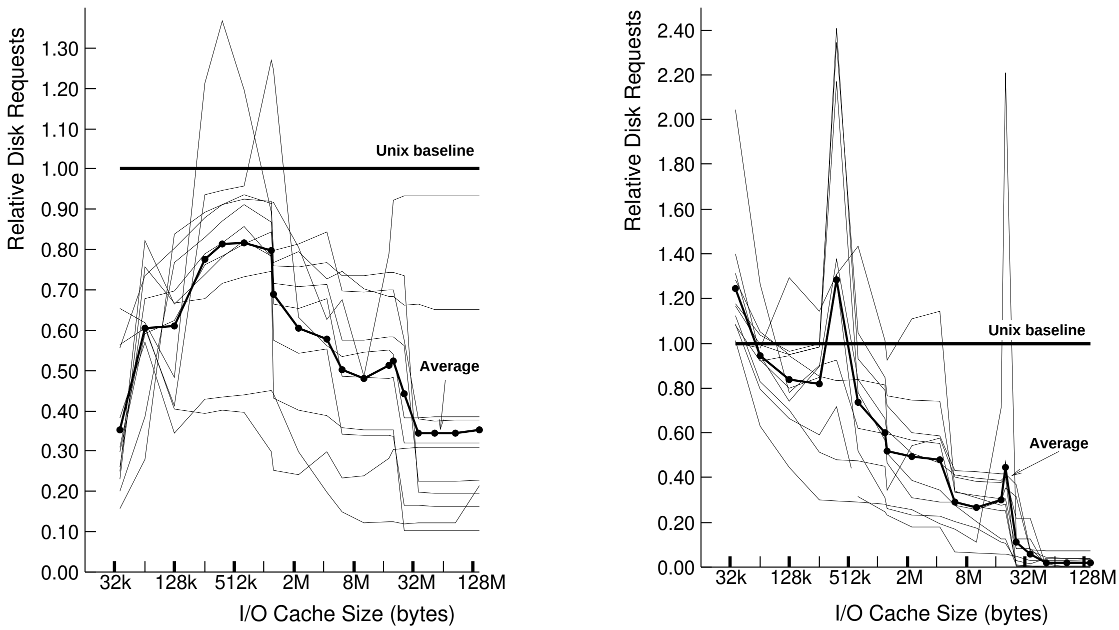
For large caches, the scheme fails for many reasons. (1) The sequential cache is too small to prevent conflicts among large files, or to capture any reuse. (2) The temporal cache cannot capture any sequential behavior. (3) The ID subcache performs the same as the two-category scheme.

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.

4.4 Total Performance Results

Each read request miss and write expulsion generates a disk access. Figure 13 shows the number of disk accesses for the variable attribute cache scheme relative to the Unix baseline. The figure breaks read



Disk Accesses from Read Request Misses

Disk Accesses from Write Expulsions

Figure 13: Read and write disk requests performance for all workloads.

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 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 14 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.

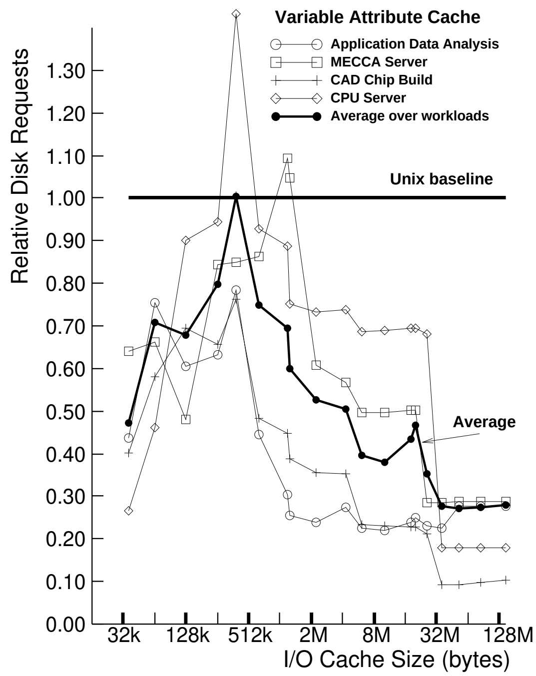


Figure 14: Total relative disk requests.

5 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 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.

Acknowledgments

This research was supported by NASA through the Graduate Researcher Fellowship program and under contract NAG2-248, and by the Digital Western Research Laboratory.

A Trace Collection and Simulation

An I/O workload trace should contain block access patterns as well as file and application information. File access patterns suffice to model broad I/O cache performance, but cannot provide the link between workload activities and cache behavior. Additional information is required to understand the nature of application I/O requests, and improve I/O cache performance in non-ad-hoc ways.

Relating cache performance to application behavior requires file system information along with application I/O requests. Understanding the nature of application I/O requests can drive I/O cache performance improvements or application I/O optimizations.

A.1 Trace Collection

A new version of the WRL tracing facilities collected traces on DECstation 5000's running ULTRIX [4, 5]. Its kernel-based approach traces all processes. The modified system logs system call information in a physically mapped trace buffer. On an I/O system call, the call type, process ID, and call parameters are entered in the buffer. On return from the system call, the return value, error status and call information are entered in the buffer. When the buffer becomes sufficiently full, the kernel schedules a special process called the *analysis program* to read and process the buffer contents. To generate an I/O system call trace, the analysis program matches call and return values, produces a file system event trace, compresses the trace and writes it to a file.

The set of I/O system calls traced includes all file related activity – read, write, open, close, create, reposition, delete, move, and executable execution. The I/O system call traces cannot be directly used for I/O cache simulations since individual I/O requests refer to state information stored within the operating system. A post pass simulates the operating system file management and produces a stateless trace. To generate a stateless trace of file block read and write requests requires keeping track of the current working directories, and simulating the operating system file table information and file descriptors [1, 7].

Unique identification numbers are assigned to each file. 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 requested from the file. For executables the number of bytes accessed equals the file size. The stateless trace drives all I/O cache simulations.

A.2 File Types

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* system 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. References to inodes and directories occur when opening files or evaluating access permissions.

A.3 Workloads

Figure 1 describes the eleven workloads evaluated in this paper. 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.

The traces cover a wide range of user applications and types of work. All traces were collected in a research laboratory with a broad range of activities. Although not necessarily typical of heavy commercial use, such as large database systems, the traces should represent many engineering development and office environments. More detailed descriptions of the workloads are included in [12].

A.4 Workload Characteristics - Dynamic Features

Most requests are to small files. Including the request contribution of inodes and directories in this measure skews the distribution further toward small files. Fewer than 1% of sequential file accesses exceed 16 Kbytes. Most of the bytes transfered to and from applications, however, reside in large files and are accessed as multiple sequential requests. 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 sequential runs of more than 256 Kbytes. The individual data request size is normally determined by the standard libraries; large sequentail runs are composed of many small sequential data requests. Thus, even though large sequential accesses do not make up a significant fraction of the actual file requests, most of the bytes transferred by the workload occur in these large sequentially accessed files. This type of behavior has been measured in other workloads as well [9, 3].

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.

A.5 I/O Cache Simulation

A single cache simulator, applied and configured in many different ways, was used to study the workload behavior in I/O caches. The caches are assumed to be non-volalitle. Writes are allocated to the cache and written back only when evicted from the cache only when they become the least recently used entry. The **I/O cache simulator** models fully associative I/O caches using an LRU replacement policy.

A **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. A single simulation produces a full range of I/O cache block behavior, read request, write request and total request miss behavior.

Request Model

An application requests file I/O. An individual I/O request may encompass several cache blocks, each of which may hit or miss in the cache. The number of cache block requests generated by an I/O request depends on the original request size, the cache block size, and the offset of the request in the file. Each file has a unique file ID. This ID, along with an offset into the file, forms 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. A cache block holds data from only one file at a time. Caches that store actual disk blocks, rather than file blocks, could have pieces of several files in a single block.

If the request size exceeds the cache size, the request 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; this usually only occurs for small caches.

Executable usage is modeled as a single request. In an entirely demand paged system these would be many small page sized requests for the executable program. Here executables are modeled as a single request for the entire executable so they generate large requests.

Read request misses reflect the number and size of read disk accesses for a workload. Since writes occur only when data is evicted from the cache, each disk write is a cache block in size. More advanced write expulsion techniques exist [11], and their impact should be similar to other reported results.

B Workload Component Behavior in I/O Caches

This appendix 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.

If all I/O requests are cached without regard to file type, few options exist for reducing the cache misses

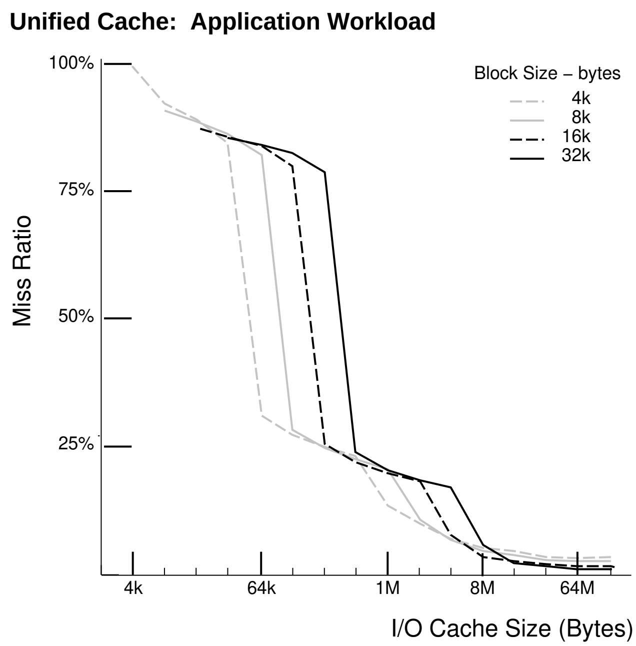


Figure 15: Typical workload behavior in a unified cache.

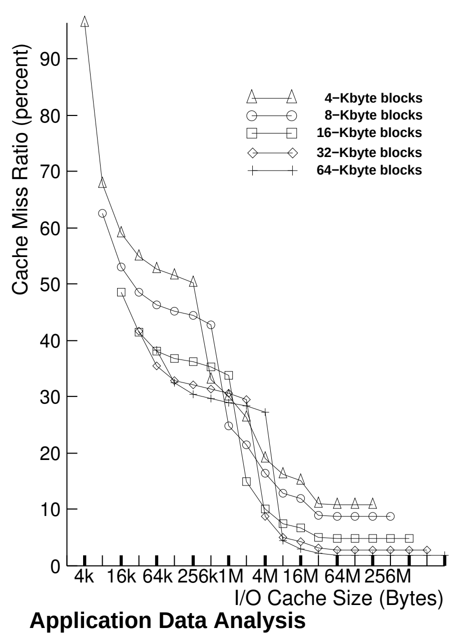


Figure 16: Datafile and Executables: miss request ratios with both temporal and sequential locality.

and improving cache performance. Figure 15 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.

B.1 Datafiles and 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.

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.

The locality in the individual workloads varies, ranging from the primarily temporal locality, to the primarily spatial locality, to both spatial and temporal. Many workloads exhibit both spatial and temporal locality, increasing the block size reduces the request misses, but increases the cache size required to capture the working set. This is the case for the workload shown in Figure 16. For cache sizes between the small and large block working set capture points, the large block size produces substantially poorer performance. The best block size choice depends on the I/O cache size, the workload working set size, and the type of workload locality.

B.2 Inodes and Directories

The cache behavior of inodes and directories are very simular; both are small and highly reused. Including both in the same cache produces uniform behavior across all workloads. Figure 17 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 directory entry occupies four blocks. The highly temporal locality of inode and directory requests greatly outweighs their spatial locality.

Because of the relationship between inodes and directories, workloads often require many inodes and directories at the same time, so the two compete for cache space. However, both working sets are small and 256 entries suffice to capture the inode and directory working sets and eliminate competition. This requires only 32 Kbytes of cache.

The combined read and write miss ratio seen in Figure 17 is about half the read miss ratio predominately due to inode writes. Most inode writes merely *update* file access times and do not modify the file system

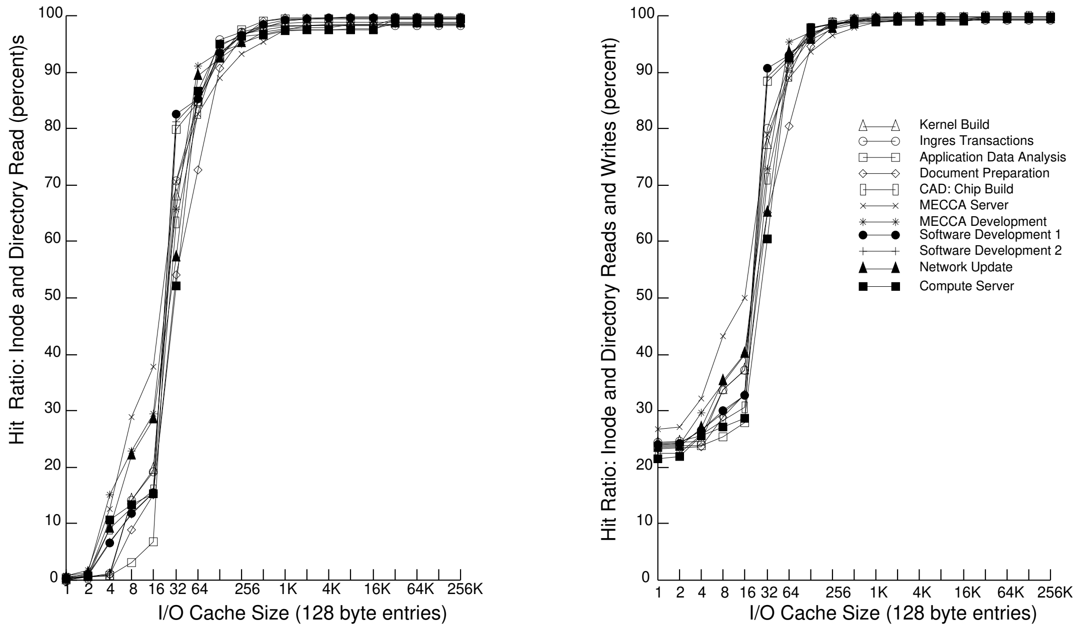


Figure 17: Inode and Directory references in a fully associative I/O cache.

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 back to the disk.

B.3 Separating Spatial and Temporal Locality

For the workloads that have large sequentially-accessed files, large blocks 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.

Sequential Cache Properties

Figure 18 shows the sequential cache request miss ratio (RMR) behavior of the large files - those least 512 Kbytes in size. The workloads exhibit two types of behavior. (1) Large files are sequentially accessed files allowing 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. 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. The *Application* workload illustrates this behavior (2) The second type of cache behavior

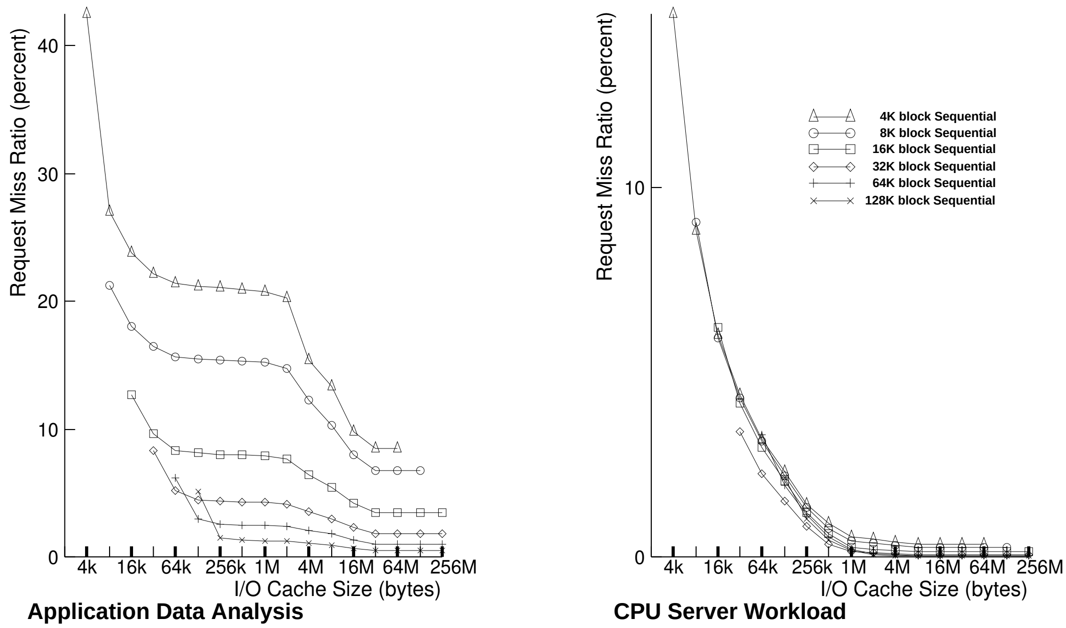


Figure 18: Sequential cache miss request ratio for files larger than 512 Kbytes.

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 *CPU Server* workload falls into this class.

The workloads that exhibit almost ideal sequential locality capture sequentially access the large files and do not reuse individual blocks; increasing the cache size does not capture more locality unless the entire file fits in the cache. A single large block suffices to capture the sequential locality of one active file. The number of blocks needed to capture sequential locality depends on the number of active files and how long the files 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. Thus, two half-size blocks perform better than a single large block.

Workloads that exhibit little sequential locality even among very large files, such as the *CPU Server*, 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.

Temporal Cache Properties

The temporal cache attempts to efficiently capture file reuse (temporal locality) and capture the working set in minimal cache space. 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

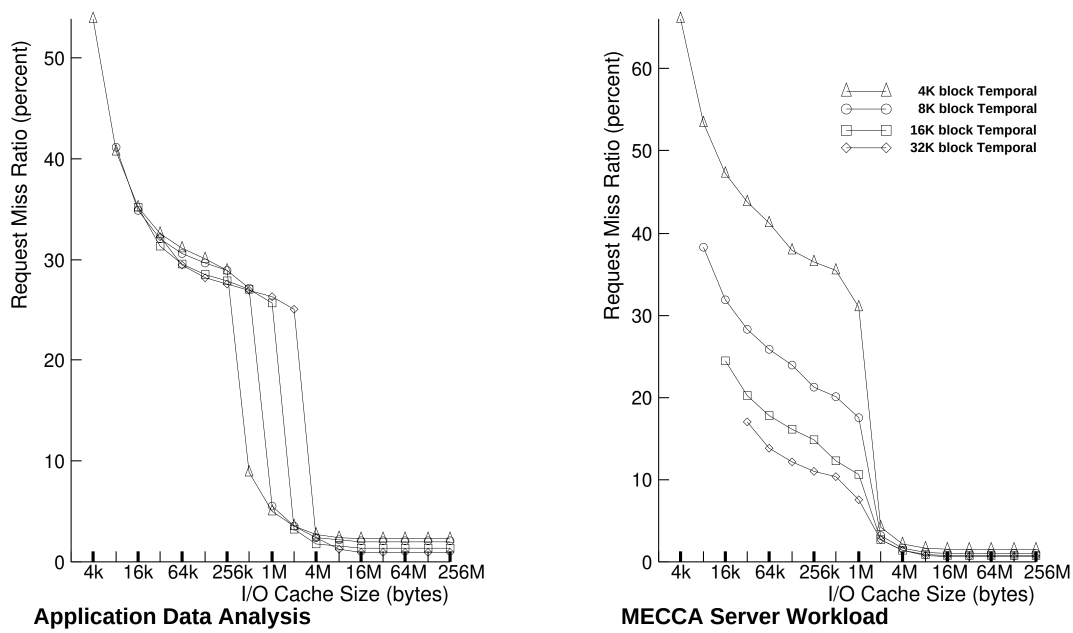


Figure 19: Temporal cache miss request ratio for files smaller than 512 Kbytes.

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* workload shown in Figure 19, like most of the workloads, exhibits 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. 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* (Fig. 19), 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.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Kathy J. Richardson. *I/O Characterization and Attribute Caches for Improved I/O Performance*. PhD thesis, Stanford University, Dec 1994. Also available as Technical Report CSL-TR-94-655.

[13] 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.

[14] 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.

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

WRL Research Reports

''Titan System Manual.''
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

''Global Register Allocation at Link Time.''
David W. Wall.
WRL Research Report 86/3, October 1986.

''Optimal Finned Heat Sinks.''
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

''The Mahler Experience: Using an Intermediate Language as the Machine Description.''
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

''The Packet Filter: An Efficient Mechanism for User-level Network Code.''
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

''Fragmentation Considered Harmful.''
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

''Cache Coherence in Distributed Systems.''
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

''Register Windows vs. Register Allocation.''
David W. Wall.
WRL Research Report 87/5, December 1987.

''Editing Graphical Objects Using Procedural Representations.''
Paul J. Asente.
WRL Research Report 87/6, November 1987.

''The USENET Cookbook: an Experiment in Electronic Publication.''
Brian K. Reid.
WRL Research Report 87/7, December 1987.

''MultiTitan: Four Architecture Papers.''
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

''Fast Printed Circuit Board Routing.''
Jeremy Dion.
WRL Research Report 88/1, March 1988.

''Compacting Garbage Collection with Ambiguous Roots.''
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

''The Experimental Literature of The Internet: An Annotated Bibliography.''
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

''Measured Capacity of an Ethernet: Myths and Reality.''
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

''Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.''
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

''SCHEME->C A Portable Scheme-to-C Compiler.''
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

''Optimal Group Distribution in Carry-Skip Adders.''
Silvio Turrini.
WRL Research Report 89/2, February 1989.

''Precise Robotic Paste Dot Dispensing.''
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’
David Boggs.
WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’
David W. Wall.
WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’
Russell Kao.
WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’
Steven J.E. Wilton and Norman P. Jouppi.
WRL Research Report 93/5, July 1994.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’
Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.
WRL Research Report 93/8, December 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’
Keith I. Farkas, Norman P. Jouppi.
WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’
Jeffrey C. Mogul.
WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’
Robert N. Mayo, Herve Touati.
WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’
J. Bradley Chen, David W. Wall, and Anita Borg.
WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’
Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.
WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’
Keith I. Farkas, Norman P. Jouppi, and Paul Chow.
WRL Research Report 94/8, December 1994.

‘‘Recursive Layout Generation.’’
Louis M. Monier, Jeremy Dion.
WRL Research Report 95/2, March 1995.

‘‘Contour: A Tile-based Gridless Router.’’
Jeremy Dion, Louis M. Monier.
WRL Research Report 95/3, March 1995.

‘‘The Case for Persistent-Connection HTTP.’’
Jeffrey C. Mogul.
WRL Research Report 95/4, May 1995.

‘‘Network Behavior of a Busy Web Server and its Clients.’’
Jeffrey C. Mogul.
WRL Research Report 95/5, June 1995.

WRL Technical Notes

''TCP/IP PrintServer: Print Server Protocol.''
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

''TCP/IP PrintServer: Server Architecture and Implementation.''
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

''Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.''
Joel McCormack.
WRL Technical Note TN-9, September 1989.

''Why Aren't Operating Systems Getting Faster As Fast As Hardware?''
John Ousterhout.
WRL Technical Note TN-11, October 1989.

''Mostly-Copying Garbage Collection Picks Up Generations and C++.''
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

''The Effect of Context Switches on Cache Performance.''
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

''MTOOL: A Method For Detecting Memory Bottlenecks.''
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

''Predicting Program Behavior Using Real or Estimated Profiles.''
David W. Wall.
WRL Technical Note TN-18, December 1990.

''Cache Replacement with Dynamic Exclusion''
Scott McFarling.
WRL Technical Note TN-22, November 1991.

''Boiling Binary Mixtures at Subatmospheric Pressures''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

''A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach''
John S. Fitch.
WRL Technical Note TN-24, January 1992.

''TurboChannel Versatec Adapter''
David Boggs.
WRL Technical Note TN-26, January 1992.

''A Recovery Protocol For Spritely NFS''
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

''Electrical Evaluation Of The BIPS-0 Package''
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

''Transparent Controls for Interactive Graphics''
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

''Design Tools for BIPS-0''
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

''Link-Time Optimization of Address Calculation on a 64-Bit Architecture''
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

''Combining Branch Predictors''
Scott McFarling.
WRL Technical Note TN-36, June 1993.

''Boolean Matching for Full-Custom ECL Gates''
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.

‘‘Ramonamap - An Example of Graphical Groupware’’
Joel F. Bartlett.
WRL Technical Note TN-43, December 1994.

‘‘Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS’’
Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.
WRL Technical Note TN-45, March 1994.

‘‘Experience with a Wireless World Wide Web Client’’
Joel F. Bartlett.
WRL Technical Note TN-46, March 1995.

‘‘I/O Component Characterization for I/O Cache Designs’’
Kathy J. Richardson.
WRL Technical Note TN-47, April 1995.

‘‘Attribute caches’’
Kathy J. Richardson, Michael J. Flynn.
WRL Technical Note TN-48, April 1995.

‘‘Operating Systems Support for Busy Internet Servers’’
Jeffrey C. Mogul.
WRL Technical Note TN-49, May 1995.