APRIL 1995

WRL Technical Note TN-47

I/O Component Characterization for I/O Cache Designs

Kathy J. Richardson

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`.

I/O Component Characterization for I/O Cache Designs

Kathy J. Richardson

April 1995

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

Abstract

When investigating I/O behavior, the tracing environment, simulation systems, and simulation model determine what can be measured and simulated. Workloads generate a variety of disk I/O requests to access file information, execute programs, and perform computation. The system presented here includes information about I/O requests, allowing workload components to be characterized individually.

The component characterization shows that I/O requests for data have vastly different reuse rates and access patterns. Disk files can be classified as accesses to inodes, directories, datafiles or executables. Inodes and directories are small, highly reused files. Datafiles and executables 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 can improve cache performance.

An attribute cache scheme illustrates the importance of I/O characterization to cache design. The scheme uses file information to cache I/O data selectively tailoring the cache scheme to the expected behavior of each file type. For a set of 11 measured workloads, a variable attribute cache scheme reduced the miss ratio 25 to 60 percent 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.

Copyright © 1995 Digital Equipment Corporation

1 Introduction

Workloads generate a variety of disk I/O requests to access file information, execute programs, and perform computation. The characteristics of the set of applications comprising each workload determine the I/O load and final system performance. A characterization shows that I/O requests for data have vastly different reuse rates and access patterns. File information can help split the workload into components with unique properties.

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. characterizing the workload shows what type of requests are the most important to performance, and focuses attention on the most important part of the problem to be solved.

This paper includes a description of the tracing system and workloads which are characterized by type, static, and dynamic resource requirements. An evaluation of the cache behavior of each component shows that each requires a very different cache configuration to best capture its locality. Lastly, an attribute cache scheme, using file information to significantly improve cache performance, shows the usefulness of workload characterization.

2 I/O Workload Traces

Direct comparisons between various I/O configurations requires delivering an identical I/O stream to each configuration. Using an I/O trace guarantees repeatable use of identical I/O streams. The tracing environment, simulation systems and simulation model determine the kind of I/O behavior that can be measured and simulated. Since the tracing environment monitors and records only part of the existing system behavior, the trace constrains the measurements that can be made and thus constrains the resulting conclusions.

2.1 Tracing

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

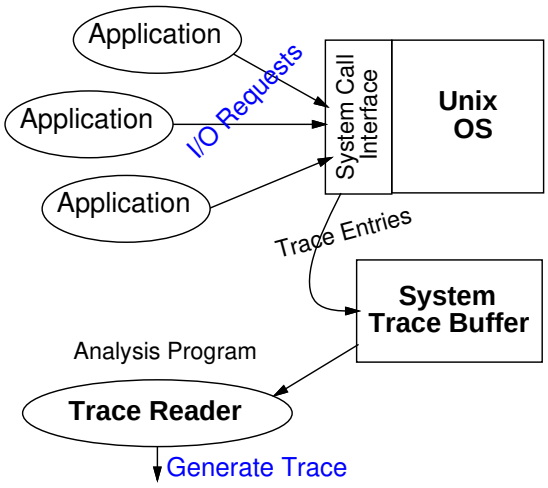


Figure 1: Logical diagram of trace, and simulation configuration.

improvements or application I/O optimizations. This requires tracing application I/O requests and including information about each request.

A new version of the WRL tracing facilities collected the traces on DECstation 5000's running ULTRIX [3, 4]. Its kernel-based approach traces all processes. Figure 1 shows the system configuration. The original system was designed primarily to study processor memory issues in a multi-programming environment. The modified system logs system call information, rather than instruction and data addresses, in the 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 in 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, 6].

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.

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

2.3 Workloads

Figure 2 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 [9].

3 Experimental Characterization of I/O Behavior

3.1 Static and Dynamic I/O Requirements by Type

Static measures of I/O behavior 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 stream must perform for 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.

Static objects are the number of unique files touched by the workload. Figure 3a breaks down, by type, the unique files referenced by each workload. A file is categorized as a datafile, an executable, an inode or a directory. Most of the unique files are either inodes or directories (greater than 75%). Inodes make up more than half the unique files accessed in all workloads, ranging from 51.3% to 86.7%. The file system associates an inode with each directory, datafile, and executable file, and the inode must be accessed to access the other files. Thus, at least half the static files must be inodes. Datafiles make up most of the remaining files, ranging from 6.9% to 41.7% for the workloads. The combination of inodes and datafiles accounts for at least 85% of the unique files in each workload.

a. Static Objects b. Dynamic References

Figure 3: References broken down by type, averaged over all workloads.

a. Static Size b. Dynamic Bytes Transferred

Figure 4: Static and dynamic size broken down by type, averaged over all workloads.

Dynamic references are the number of requests made for files, either whole or partial. Figure 3b breaks down the references according to type. 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). Each reference brings in an entire executable. 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 file count. Inodes are the most frequently referenced type in all but one workload, 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 file contribution. For individual workloads, the datafile reference contribution does not correspond to its static contribution. The relationship varies considerably with workload.

Static size is the total amount of space required to store all the files touched by a workload. Figure 4a 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 files forms an upper bound on the amount of storage space required to capture all re-references to the files. Since the working set changes over time and some parts of datafiles 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 they are less than 5%. Executables contribute much more; their relative static size contribution varies from 1.7% to 41.6%. Datafiles require the majority of the total space requirements.

Dynamic bytes transferred is the total number of bytes requested by the workload. Figure 4b 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 large and each request retrieves the entire file. A small number of requests transfer a large amount of data. Only about 45% of the bytes requested from the operating system are datafiles used as actual computation data.

Conclusions from Static and Dynamic Type Characteristics

Each I/O type differs in importance. Inodes and directories comprise a significant percentage of the files, but they are small and require little space. They also have the greatest average reuse, so they are the most important files to keep in the cache. As Figure 4 showed, executables require considerably more space than do directories and inodes, and datafile files require still more.

To provide the greatest 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 amount of data transferred is not decreased. Because of the large differences in the file 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.

3.1.1 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. Figure 5 shows the distribution of run lengths by type and the distribution of bytes transferred for these runs. The x-axis is divided into run lengths; the maximum run size is shown for each category. For example, the label *32k* includes all runs greater than 16k and less than or equal 32k bytes. The distribution of run lengths tails off quickly. Fewer than 1% of the runs exceed 16 Kbytes. 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,

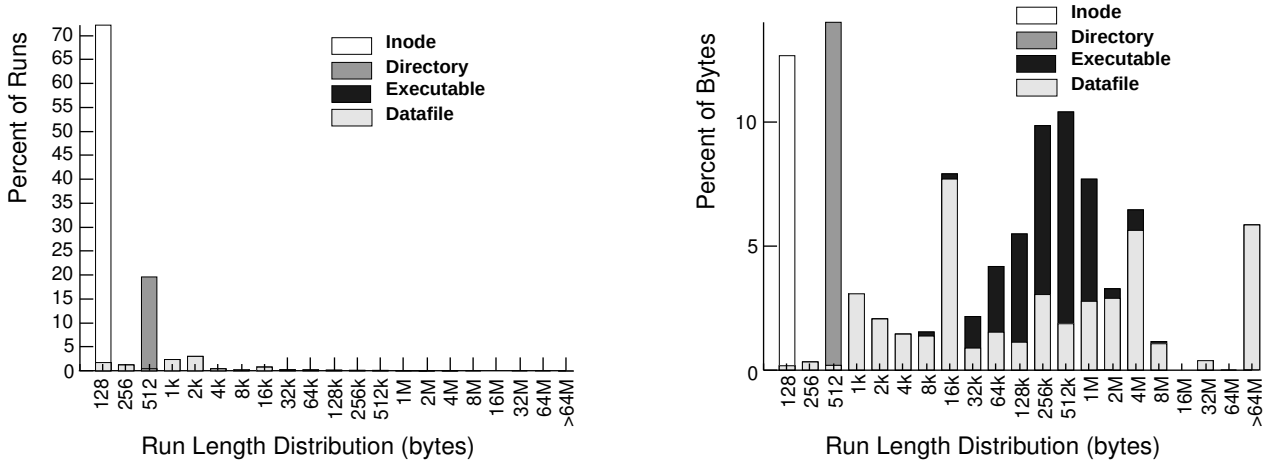


Figure 5: Run length distributions.

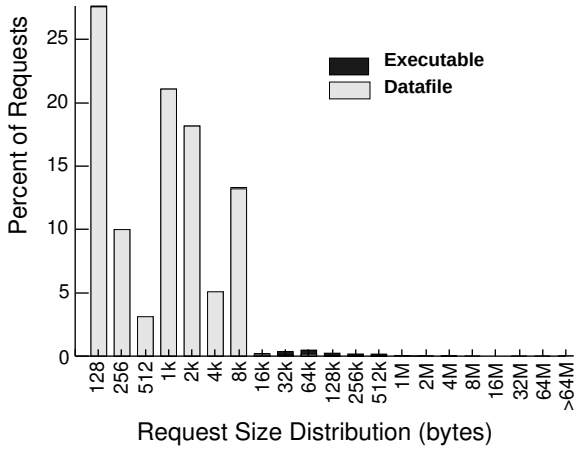


Figure 6: Distribution of request sizes for datafiles and executables.

most of the bytes transferred by the workload occur in these large sequential runs. This type of behavior has been measured previously [7, 2].

Figure 6 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, these runs have a considerable sequential locality that can be exploited with an appropriate 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 re-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 files 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.

4 Performance Modeling and Simulation

4.1 I/O Cache Simulation Model

A single cache simulator, applied and configured in many different ways, was used to study the workload behavior in I/O caches. The **I/O cache simulator** models fully associative I/O caches using an LRU replacement policy. Fully associative caches efficiently simulate multiple caches simultaneously [11].

To reproduce 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 7 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 the request misses based on a supplied condition (COND) and its complement. The condition typically breaks down read and write requests separately (READ and not-READ). A single simulation produces a full range of I/O cache block behavior, request miss behavior, and request component behavior based on the condition.

4.2 Request Model

The request model evaluates I/O requests from the application viewpoint and describes how the requests break down into cache block requests. An application requests file I/O. Each request generates 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 cache 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.

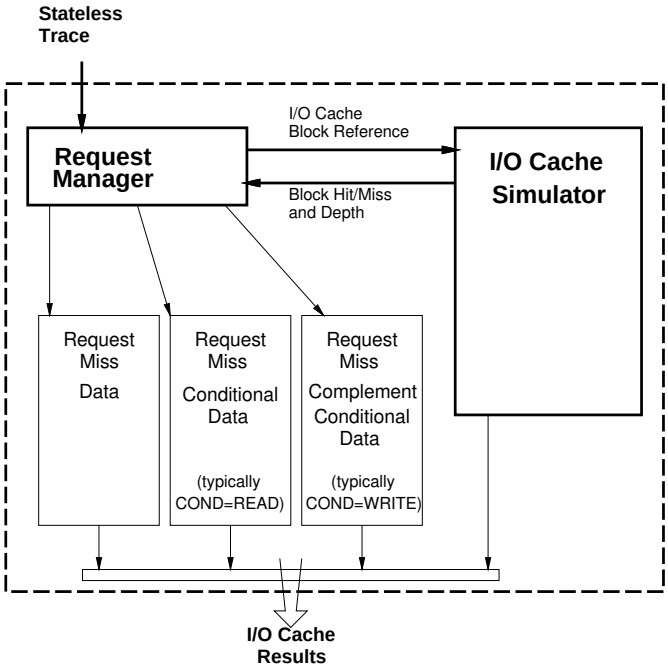


Figure 7: Cache simulator layout.

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. 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 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. Simulating disk blocks requires knowledge of the disk layout; the traces did not contain actual disk block addresses, so this was not modeled.

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. A small cache might therefore have more request misses than actual requests.

The **I/O 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 request misses in the cache if any of the cache blocks it accesses miss. Request misses reflect the disk activity caused by I/O cache misses. 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.

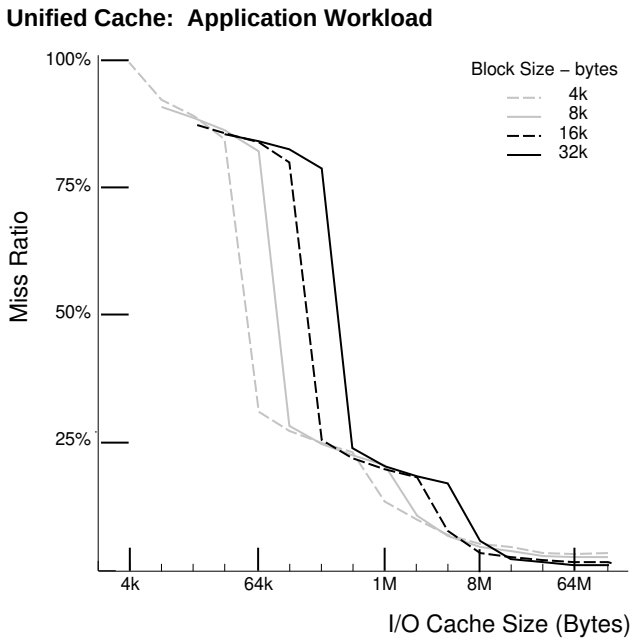


Figure 8: Typical workload behavior in a unified cache.

These are termed write expulsions. Each disk write is a cache block in size. More advanced write expulsion techniques exist [8], and their impact should be similar to other reported results.

5 Workload Component Behavior in I/O Caches

This section examines the cache behavior properties of the various I/O workload components. The behavior differs for each component. Different components require different block and cache sizes to capture locality effectively. Understanding the sources of locality and the cache properties that best capture locality can help to improve I/O cache efficiency. For a more thorough study of I/O cache behavior see [10, 9].

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 8 shows the cache miss behavior of the *Application* workload in a unified cache that uses no information about request types. The miss ratio is the percent of requests not serviced by the cache. Due to the overwhelming number of small requests from inodes and directories, smaller block size choices always win. This behavior is typical of all the workloads. Other work has focused on ad-hoc techniques for separating the temporal and sequential components with cleaver cache configurations [8, 5]. Directly 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

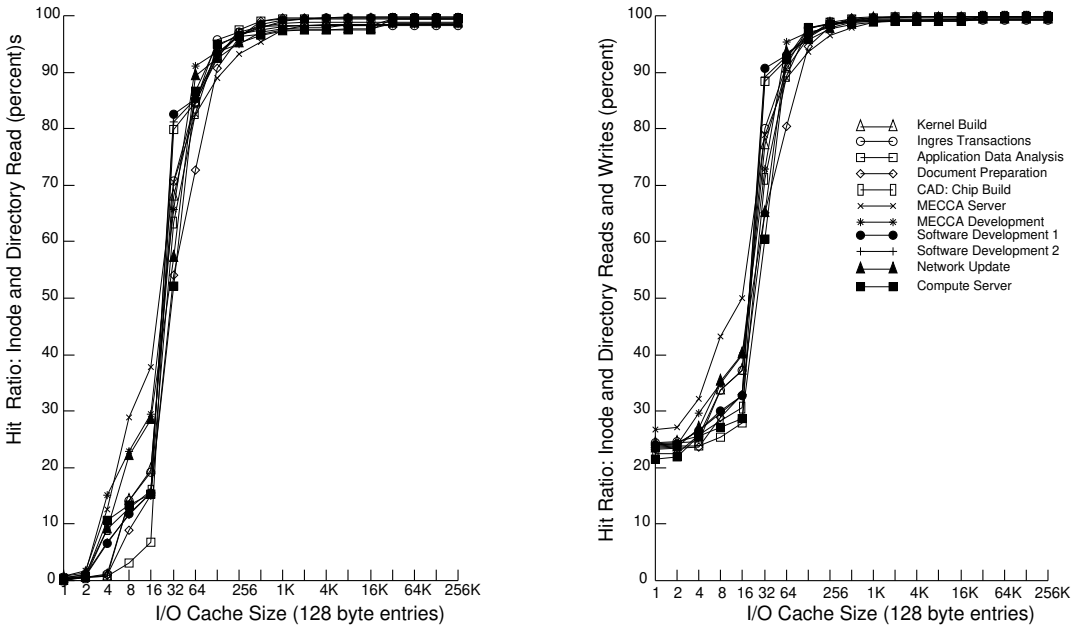


Figure 9: Inode and Directory references in a fully associative I/O cache.

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 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. The hit ratio increases dramatically when the cache captures the workload's set of highly reused files. This is referred to *working set capture*. High hit ratios, low miss ratios, and smaller working sets all mean better cache performance. All the caches presented use a fully associative LRU replacement policy, and use a copy-back write policy.

5.1 Inodes and Directories

Inodes and directories have similar cache behavior. Including both in the same subcache produces uniform behavior across all workloads. Figure 9 shows the read hit ratio and the total hit ratio for inodes and directories together in a cache with 128-byte blocks. The cache size is measured in 128 byte inode *entries*. In this cache, a single 512-byte 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.

5.2 Datafile and Executable Cache Behavior

Block Size Effect on Datafile and Executable Locality Capture

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 short runs, large blocks reduce the cache performance by inefficiently using cache space. If blocks are much larger than the file size, 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.

Executable and 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. Figure 10 shows the block size effect on cache behavior for datafiles and executable files. The four workloads shown 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.

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 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. 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. In a cache considerably larger than the working set, large blocks capture sequential as well as temporal locality. The temporal locality capture is dominated by the number of independent small files that fit in the cache.

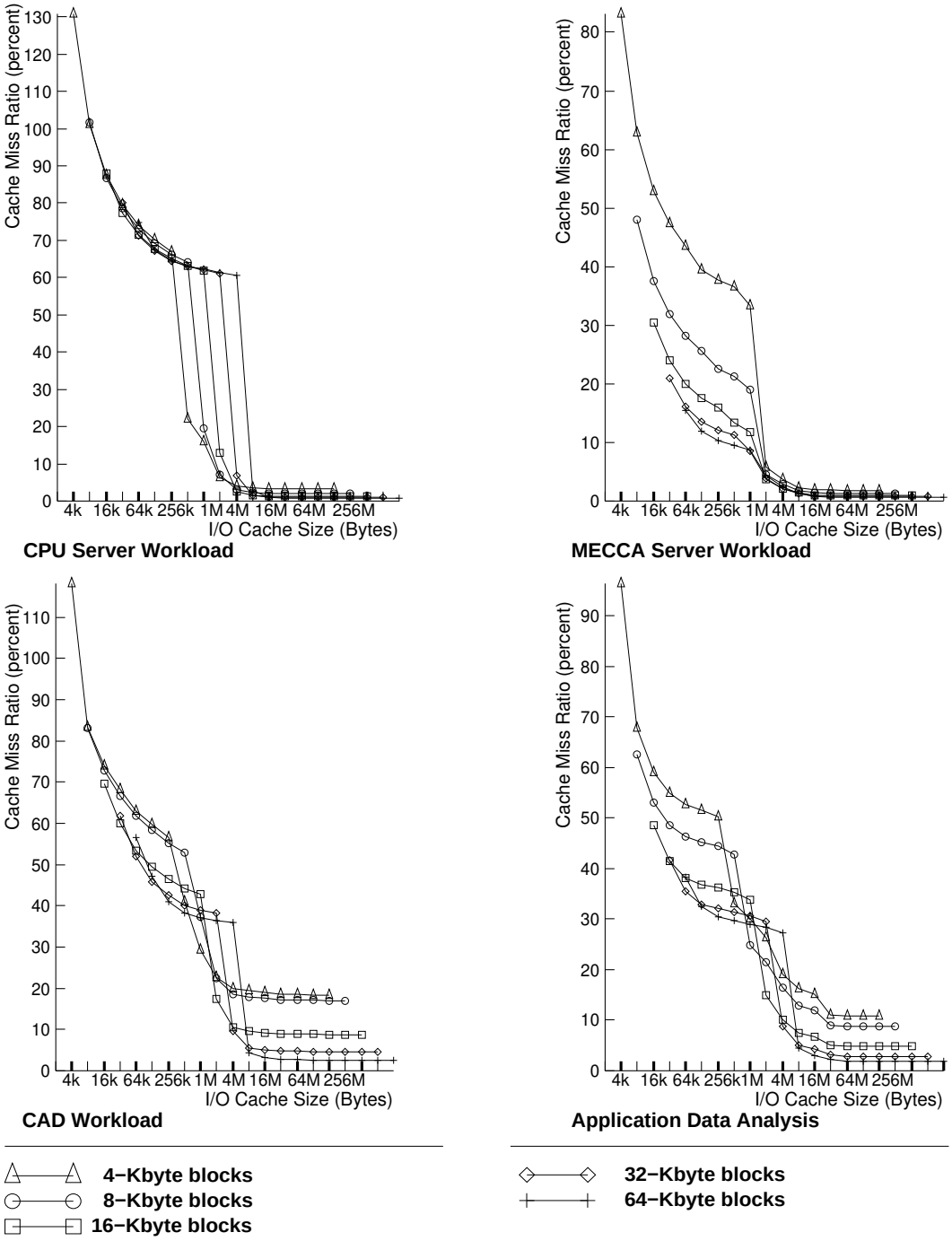


Figure 10: Miss request ratios of datafiles and executables, showing temporal locality (top left), spatial locality (top right), and mixed locality (bottom row).

Increasing the number of blocks beyond that needed to store all the independent files in the working set does little to improve the temporal locality capture, and nothing to improve sequential locality capture. Beyond this point, increasing the block size increases the sequential locality capture. For the measured workloads, an 8-Mbyte cache using 4-Kbyte blocks always captured the working set.

Separating Sequential and Temporal Locality

The statistical breakdown of I/O accesses provides information about workload locality. Most of the I/O bytes transfer sequentially to or from large files of more than half a 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 exhibit 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 the 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. Because of the high reuse rate typical of moderate files it is crucial to capture their working set.

Sequential Cache Properties

Figure 11 shows the sequential cache request miss ratio of the large files in each of four workloads. The Figure shows *512KB cut-off* caches, holding only files of at least 512 Kbytes. The request miss ratio is the fraction of datafile and executable request misses contributed by the large files, not the fraction of large file requests that miss.

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

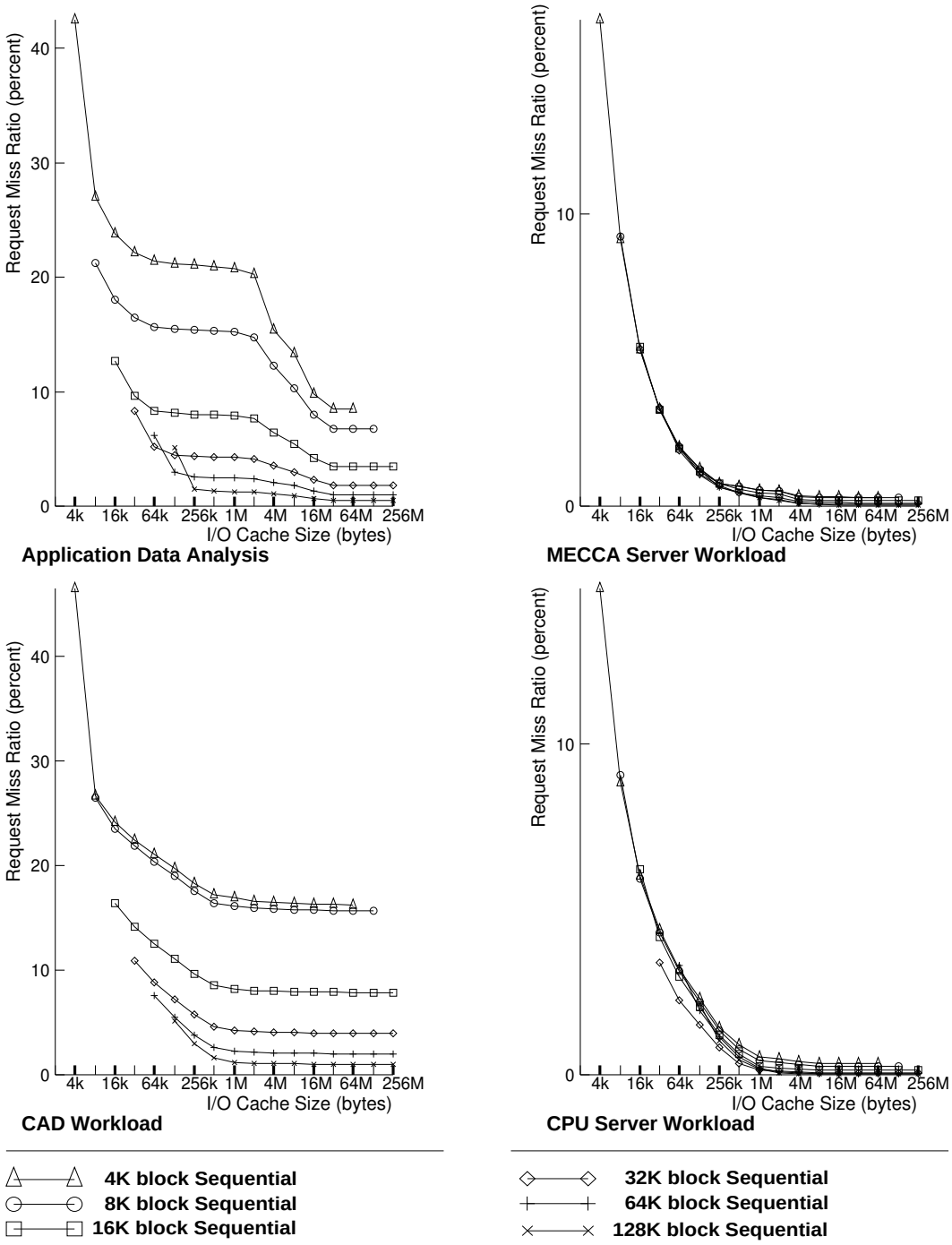


Figure 11: Miss request ratios for a sequential cache holding files larger than 512 Kbytes.

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, contention forces out actively-used blocks. Contention for cache space becomes more pronounced for larger cache blocks. Thus, two half-size blocks perform better than a single large block.

The large files in the *MECCA Server* and the *CPU Server* workloads exhibit almost no sequential locality 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.

Temporal Cache Properties

Figures 12 shows the temporal cache request miss ratio for moderate size files in the four representative workloads. The figure shows the temporal cache behavior for files smaller than 512 Kbytes. As with the sequential cache results, the temporal request miss ratio is the fraction of request misses 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.

Smaller cache blocks increase the usable cache space by reducing the amount of unused space per block, and by increasing the number of independent files 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 capture highly reused data more effectively.

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 Figure 12 show only temporal locality behavior. Eliminating all large files from the cache effectively eliminates all the sequential behavior from the temporal caches for most workloads. Then larger blocks do little to reduce the miss ratio for any cache size.

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. *MECCA* accesses a large set of medium-sized files sequentially. It reuses these files frequently, producing a large working set.

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

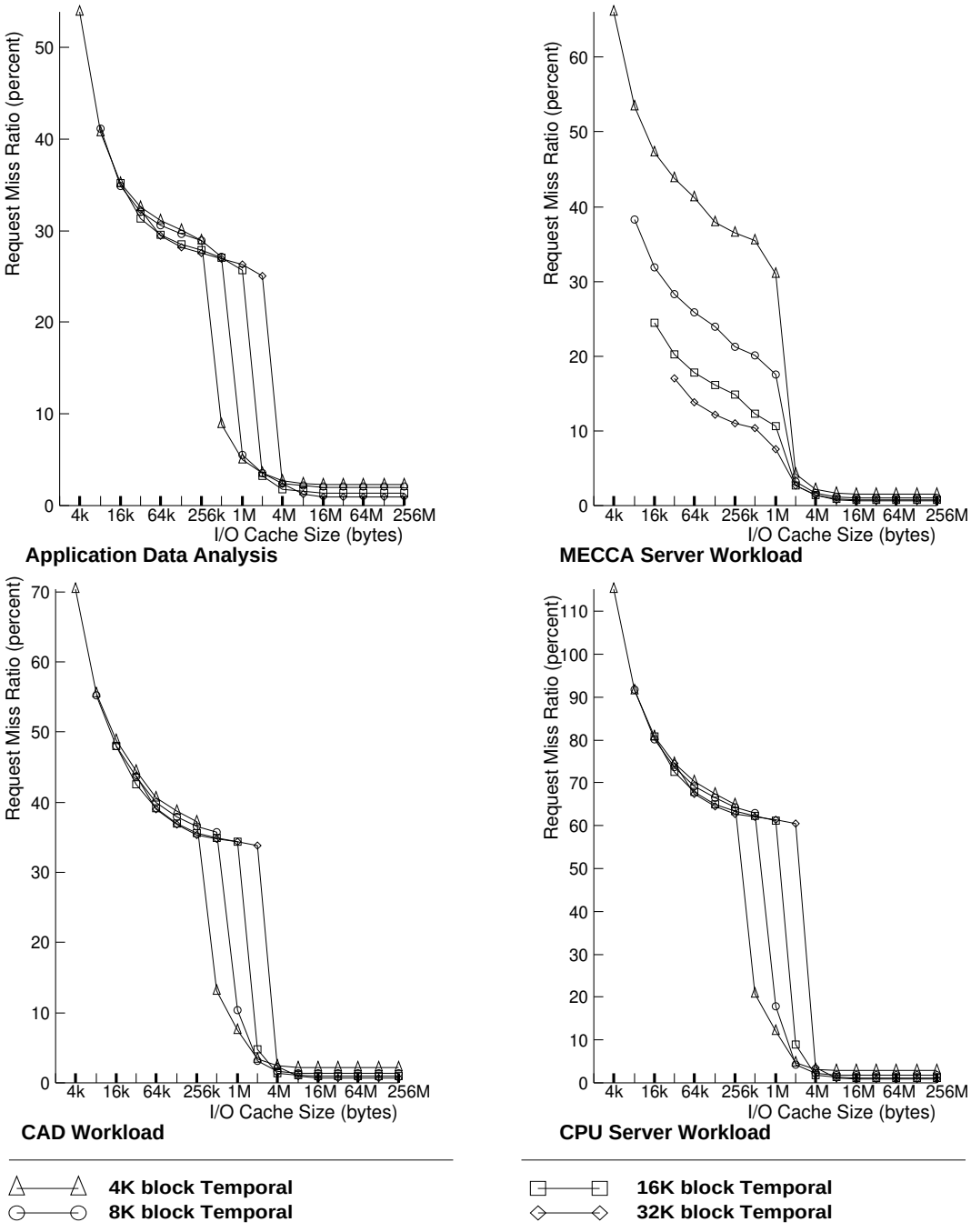


Figure 12: Miss request ratios for a temporal cache holding files smaller than 512 Kbytes.

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.

5.3 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 to hold 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.

6 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 with 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 13 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 Inode and Directory (ID) and sequential subcaches to capture inode and directory

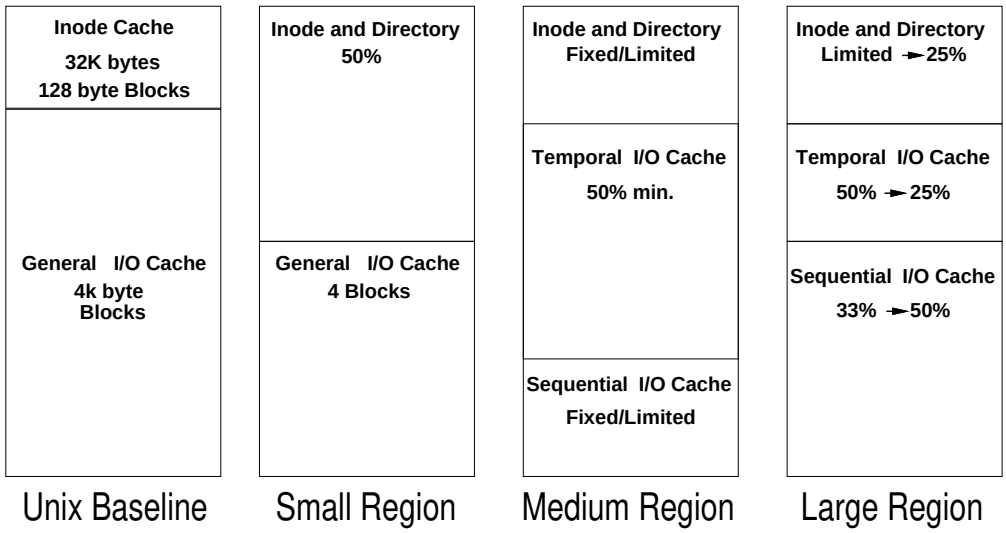


Figure 13: Variable attribute cache configurations for each size region.

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 to 25% of the cache area.

6.1 Read Request Behavior

Figure 14 compares the variable attribute cache scheme with the Unix baseline scheme for the four representative workloads. The variable scheme lowers the read request miss ratio (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 miss ratios comparable to the baseline, 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 15 compares changes in the read requests for the four sample workloads, relative to the Unix baseline cache. It also includes a graph showing results for all eleven workloads. 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

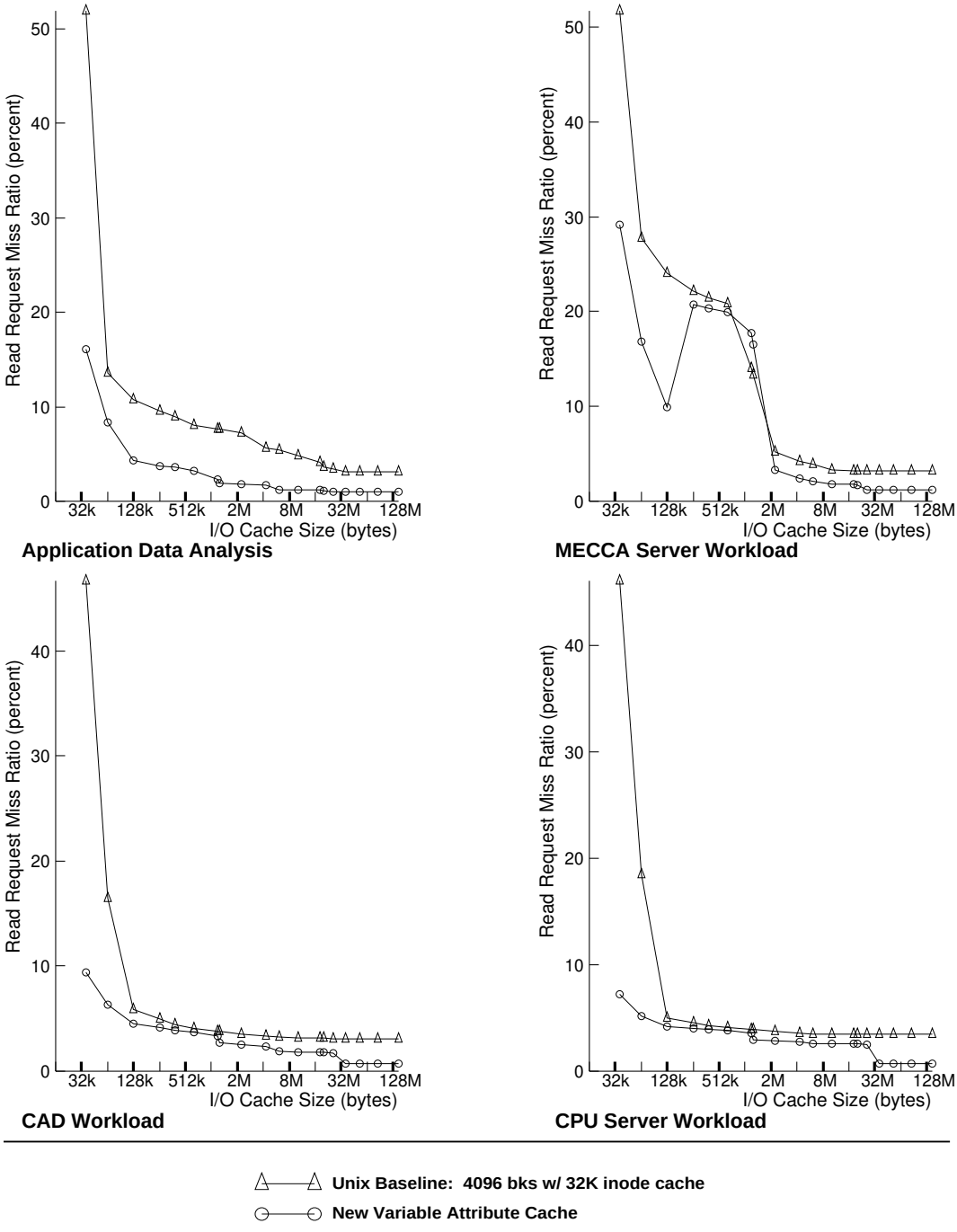
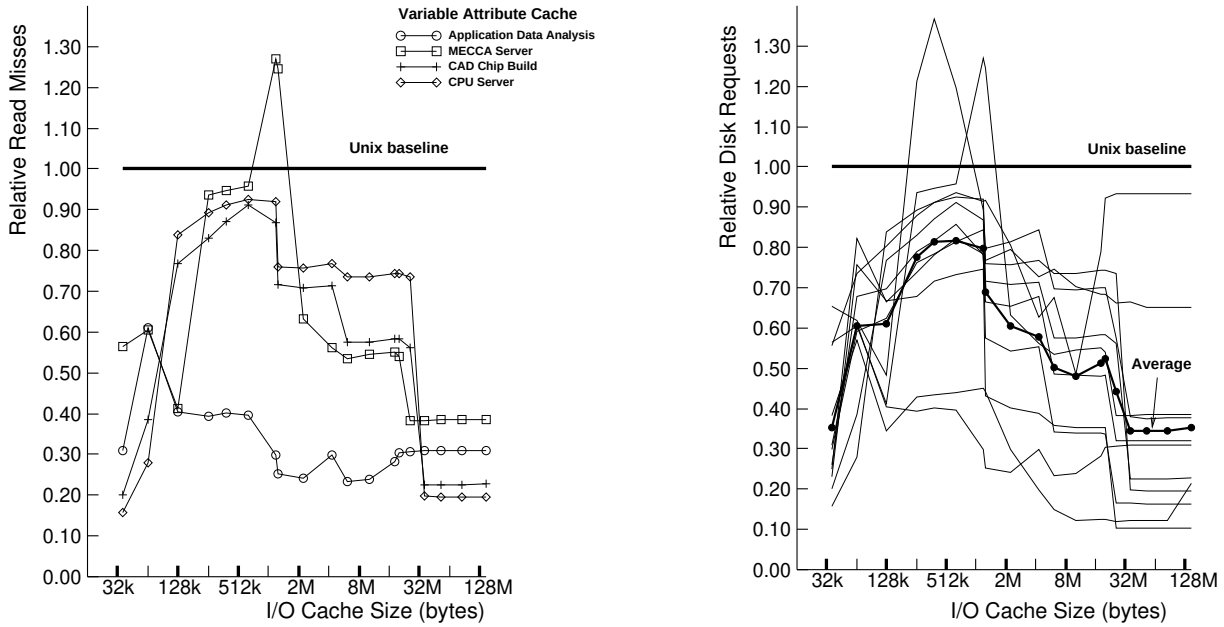


Figure 14: Attribute cache scheme request miss ratios.



Selected Workloads

All Eleven Workloads

Figure 15: Relative Read Disk Requests.

when applications must wait for I/O requests to complete.

Read request misses generate disk accesses, determining the number of 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.

7 Conclusions

Distinct components make up the I/O workload. By including file system information with I/O requests, these components can be separated. For a set of 11 workloads, file type and size information sufficed to distinguish different cache behaviors. 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.

I/O caches should make use of file types to improve their efficiency. 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 file size reduces unused cache space, increases cache utilization, and reduces the number of request misses.

Partitioning the cache prevents large files from filling up the entire cache and forcing out small files. This allows the cache to capture the working set of individual components even if the workload working set exceeds the cache size. Capturing the working set of individual components significantly reduces the total request misses.

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.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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.