# FACTORING FILE ACCESS PATTERNS AND USER BEHAVIOR INTO CACHING DESIGN FOR DISTRIBUTED FILE SYSTEMS

by

SHARUN SANTHOSH

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2004

MAJOR: COMPUTER SCIENCE

Approved By:

_____

Advisor Date

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Staying connected anywhere, anytime isn't any more a foreseeable eventuality but has become a present day reality. To get a sense of how connected the world is growing some statistics are presented. According to the International Telecommunications Union, in the year 2002 there were 580 million Internet users, 615 million personal computers, and 1,155 million mobile phones(a little more than the total number of telephone subscribers in the world). Forecasts for 2003 show an increase by *at least* 50 million in each of these categories. About 20 million Americans will have broadband Internet access and yet the US will rank only 11th in terms of broadband penetration. Taiwan is believed to have more mobile phones than people. The numbers are impressive and growing.

The need to stay connected to people and data is the driver. It has lead to the development and emergence of numerous new technologies. There are as many ways to use data as there are ways to connect to it. A person may stay connected via half a dozen different networks throughout the course of a day, from a cable

modem or DSL connection at home, to a high speed ethernet network at work or school, to a bluetooth network in the car, to a WiFi network at the airport or the neighborhood coffee shop or using their latest 3G mobile phones.

Along with the freedom to roam through these diverse environments the user expects to be connected to personal files and data where ever he or she may be. Availability of data and ease of access to it is crucial to end user satisfaction, but providing it still remains a big challenge. Each method of connectivity has its own characteristic requirements, services and limitations. To be effective, an underlying system should be able to adapt to the diverse nature of the connectivity available. A distributed file system provides one way of tackling this complicated problem.

Distributed file systems have been extensively studied in the past [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], but they are still far from wide acceptance *over heterogeneous network environments*. Most traditional file systems target high speed reliable homogenous network environments and *therefore do not work well in a wide-area heterogenous context.* Providing seamless, secure file access to personal documents by adapting to diverse network connections is *the* desirable goal.

The *Cegor(**C**lose**E** and **G**o **O**pen and **R**esume) file system* [12] has been proposed to address the important issues of availability, security, transparency and adaptability across the different environments that exist today. It tries to address these issues in the following ways:

- Reducing communication frequency to adapt to varying network conditions using semantic based caching
- Type-specific communication optimization to reduce the bandwidth require-

ment of synchronization between client and server

- A Connection-view based transparent and secure reconnection mechanism

This thesis focuses on semantic based caching. In order to design and implement such a system effectively, a good understanding of user behavior and file access patterns is important. Studies have been conducted on existing file systems [13, 14, 11, 15, 16] but not periodically enough to reflect or predict present day workloads. Publicly available file system traces are almost a decade old.

Trace-driven file system analysis is necessary to collect useful information to support the design process. This thesis had two objectives. First, to investigate the important factors that influence the design of such a system, through the analysis of existing and current file system traces. Second, to use the results to study caching as a means of reducing communication overhead and hence reliability on varying network conditions.

Based on the results, a semantic-based cache replacement algorithm has been proposed. We show it is more suited to heterogenous environments than existing caching approaches. The results also provide an insight into how access patterns have changed over the years, allowing us to speculate on workloads of tomorrow and their influence on future file system design.

## 1.1 Distributed File Systems and Caching

Traditionally, the three basic processes involved in *anywhere, anytime* data access are, retrieve the files from the server, work on them locally and write the changes

back to the server. Almost all clients in distributed file systems have a client-side cache [1, 5, 8, 17] to *minimize communication with the server*, improving system performance on the whole. It can not be assumed that a fast network connection to a file server *always* exists and thus policies must be designed to have a minimal reliance on communication, resulting from cache misses (fetching) and cache replacement (update synchronization).

In the very popular network file system NFS [1] delayed write backs are implemented, writes are flushed back to the server from the cache after some preset time or when a file is closed. The Andrew file system AFS [4] uses a write back on close policy to reduce write traffic to the server. When the cache is full and programs request more data from AFS, a Cache Manager must flush out cache chunks to make room for the data based on a LRU type algorithm. So in general, each time a cache gets full, data has to be either *written back* to the server or dropped (generating an *additional exchange* with the server, the next time it is requested). Replacements can be expensive.

In the case of high-speed networks this may not be noticable. But caches of distributed file systems, that operate across *heterogeneous*, especially low-bandwidth, networks, must not only provide the hit ratios of conventional caches that operate over homogeneous high-speed networks, but must do so performing as few replacements as possible. This thesis reexamines caching in this context.

## 1.2 User behavior and File access patterns

As stated earlier to design an efficient system a good understanding of user behavior is required. We are interested in *single user* needs so as to provide the close and go open and resume experience across an heterogenous environment. Though traces of user activity have been collected and studied in the past, there are several reasons they cant be used in the context of our problem.

- Traces haven't been collected periodically enough to reflect present day usage activity.
- The traces available today have been collected to evaluate systems such as BSD, AFS, Sprite or CODA after the implementation phase rather than before the design phase. Traces are more usefull in the design stage, as modifying existing systems based on trace driven evaluation is often complex. An example is the recently proposed translucent caching mechanism for CODA [18]. It cannot be incorporated into an already complex system without a complete reimplimentation of the entire system.
- Traces that are available such as those collected at the disk driver level [15] or web proxy traces [19] don't give designers relevant information on single user activity and access patterns.

The above reasons provided us with a strong motivation, to collect a new set of traces that reflected current user behavior. The analysis of the new traces provided information to design new caching algorithms. It served as workload over which

to test the proposed caching techniques and also showed how user behavior and access patterns have changed over the past decade.

## 1.3 Semantic-based Caching

File access patterns aren't random. Many algorithms exist that utilize the inherent relationship between files(inter-file semantics and intra-file semantics) to perform pre-fetching [20, 21, 22, 23] or hoarding [7]. Problems related to these approaches are discussed in the next chapter.

The proposed approach, on the other hand *does not* use this relationship information to prefetch files or hoard them, rather it concentrates its efforts on preserving these relationships in a cache. An *eviction index* is defined and calculated based on these relations. 'Strong' relations are preserved and 'weak' ones replaced. Not only does this approach deliver effective hit ratios but it also decreases the communication overhead as compared to other replacement algorithms such as LRU, LFU and Greedy-dual size [19] when run against the DFStraces [11] from CMU and our own MIST traces. This approach could be important for two reasons:

- It takes the relationships between files into consideration that conventional approaches ignore.
- It minimizes the need for replacement which could mean increased file availability or reduced synchronization overhead.

User patience depends on importance of the file and expected fetch delay [24]. Importance of a file is specified by user in some systems [18, 24]. We try to remove

this burden by building file relationship information based on which caching is done. This produces better hit rates than LRU while in some cases performing half the number of replacements. It is felt by allowing more files to remain in the cache the user can be given more options using translucent mechanisms [24] when in a weakly connected or disconnected state.

## 1.4 Contributions

User behavior and file access patterns are constantly changing. Knowledge of system workload is essential in optimizing existing systems and in designing new ones. File system activity must be traced periodically to provide us with this information. But publicly available file system traces are almost a decade old. The traces collected during the course of this study are important for the following reasons

- The traces help us better understand *present day* user behavior and file access patterns
- They provide a realistic and practical workload over which experimental systems may be developed and tested
- The new traces allow us to perceive change and the reasons for it, when compared against older studies

A trace collection utility has been built that is transparent to the user and does not affect system performance. Some important results from the trace analysis are

- Most files were opened for less than a hundredth of a second
- The majority of files are accessed only a few times. There is a small percentage of very popular files
- The majority of files are less than 100KB in size. Large file can be very large
- Almost half the accesses repeat within a short period of initially occuring
- File throughput has greatly increased due to presence of large files
- Majority of files accessed have a unique predecessor

While several file systems exist to support mobility, cache management policies haven't been examined in detail. All most all systems in existence today use LRU as the replacment policy. This thesis considers the problem of caching in file systems that need to operate across heterogenous environments. Caching has been reexamined with new data and new goals. Files are cached based on inter or intra-file relationships. This semantic based caching approach not only yields better hit rates but does so performing almost half the replacements, compared to conventional approaches.

## 1.5 Outline

This thesis is organized as follows. Chapter 2 provides an overview of previous work, describes caching in existing distributed file systems, factors that effect caching and issues that need to be addressed in future systems. Chapter 3 describes the process of trace collection, the observations made on file charateristics

such as size, popularity and semantic relationships. Comparisions with existing studies on the DFSTraces are also made that reflect changes in user behavior and file access patterns over the past ten years. The design of the semantic based cache replacement algorithm based on inter-file and intra-file relations is described in Chapter 4. Chapter 5 describes the replacement policies and metrics used in to study the performance of the proposed approach. Chapter 6 summerizes the work done and results obtained.

# Chapter 2

# Background

In view that this thesis involves both the study of file systems through tracing and the study of caching, as a means of performance optimization to such systems, this chapter has accordingly been divided into two parts. The first part describes the workload characterization studies on existing file systems. Factors influencing the design of caching and the caching techniques used in existing distributed file systems are described in the second part.

## 2.1 Workload of Existing Distributed File Systems

The goal of most trace driven studies involve tracing existing distributed file system activity to evaluate the existing system or to design techinques to optimize performance [13, 11, 14, 15]. The problem being focusing on here involves a single user moving across a heterogenous networking environment. The goal here is to understand the user behaviour and access patterns on single user file system and

design a system that allows a user to emulate this behavior across a heterogenous environment. Some existing studies on user behavior and file acces patterns are presented in this section.

One of the earliest studies by Ousterhout [14] in 1985 on the UNIX 4.2 BSD file system showed that average file system bandwidth needed per user was a few hundred bytes per second. Files were open only a short time (70-80% were open less than 0.5 seconds) and accessed sequentially. 80% of all acesses were to files less than 10KB long.

Six years later a study [13] was conducted along similar lines on the Sprite network file system [8]. They found similar results; most files were short, opened for brief periods of time and accesses were sequential. Significant changes observed were that large files had grown larger and file throughput had increased by a factor of 20 and was very bursty with peak throughput reaching 9MB per second. They found that increases in cache sizes resulted in increased read hit ratios but not as expected due to the increase in large file sizes. There was no improvement in reducing write traffic using caching. They felt that as cache sizes increase more read requests would be handled by the caches, leading to the conclusion that writes would dominate file system performance.

Rosenblum [10] too predicted that future workload would be dominated by write traffic leading to the design of log structured file system. Here all modifications to files in main memory would be written to disk to speed up write operations. A latter study at HPlabs [16] found this prediction did not hold as the number of reads in newer traces have been seen to have greatly increased. The above studies involved tracing at the file system level. Tracing at the disk level [15] revealed that

only 13-41% of accesses are to user data. All these studies are nearly a decade old and motivated us to make new measurements.

## 2.2 Caching in Existing Distributed File Systems

Distributed systems consisting of workstations and shared file servers typically have main memory disk block caches at both the workstations and the file servers. These caches form a two-level hierarchy in which file I/O requests generated by applications running on the clients may be satisfied at the local cache or if not then possibly at the file server cache. These caches improve system performance by reducing the frequency with which costly disk operations are performed and in the case of the caches at the workstations the frequency of requests to the file servers, in addition to reducing network load.

Crucial to the efficient functioning of a file system cache is a good replacement policy. The replacement policy specifies which disk block should be removed when a new block must be entered into an already full cache and should be chosen so as to ensure that blocks likely to be referenced in the near future are retained in the cache. A lot of work has been done on distributed systems and caching. In this section we look at existing approaches, the issues they address, and their advantages and shortcomings.

A discussion of client side caching in different distributed file systems follows. The section following that lists the issues that need to be looked at during design.

### 2.2.1 NFS

In Sun's very popular network file system [1] file block as opposed to whole file caching is used on the client side. NFS performs file, attribute, and directory caching. Attributes and directory information are cached for a duration determined by the client. At the end of a predefined timeout, the client will query the server to see if the related filesystem object has been updated. When a file is *opened*, the copy in the cache needs to be revalidated with the one in the server. A query is sent to the server to determine if the file has been changed. Based on this information, the client determines if data cache for the file should kept or released. When the file is *closed*, any modified data is flushed back to the server. Repeated reference to the server to find that no conflicts exist is expensive. The communication produced each time a file is opened or closed, or for invalidating attributes can result in serious performance drawbacks. A common case is one in which a file is only accessed by a single client, where sharing is infrequent and constant invalidation is unneccessary. Another problem is if clients are geographically distributed there is an increase in the latency for cache revalidation requests. In a heterogenous environment this must be reduced as much as possible to minmize reliance on the underlying link.

The NFS version 3 [25] client significantly reduces the number of write requests it makes to the server by ”collecting” multiple requests and then writing the collective data through to the server's cache. Subsequently, it submits a commit request to the server which causes the server to write all the data to stable storage at one time.

NFS version 4 [25] uses *delegation* to reduce communication overhead. At open, the server may provide the client either a read or write delegation for the file. If the client is granted a read delegation, it is assured that no other client has the ability to write to the file for the duration of the delegation. If the client is granted a write delegation, the client is assured that no other client has read or write access to the file. Delegations can be recalled by the server if another client requests access to the file in such a way that the access conflicts with the granted delegation. This requires that a callback path exist between the server and client. If this callback path does not exist, then delegations can not be granted.

The NFS version 4 protocol does not provide distributed cache coherence. However, it defines a more limited set of caching guarantees to allow locks and share reservations to be used without destructive interference from client side caching. All this said, NFS remains one of popular file systems in use today.

### 2.2.2 AFS

The Andrew file system [17] developed at CMU originally used whole file caching but now large files are split into 64K blocks. Write back on close semantics reduces write traffic to the server. Cache consistency is achieved through a callback mechanism. The server records who has copies of a file. If the file changes, server is updated (on close). The server then immediately tells all the clients having the old copy. The callback scheme allows clients to cache data until the server tells them it's invalid, while in NFS clients must continually revalidate their cached data. When the cache is full and application programs request more data from AFS, a

*Cache Manager* must flush out cache chunks to make room for the data. The Cache Manager considers two factors:

- 1. How recently an application last accessed the data?
- 2. Is the chunk is dirty?

A *dirty* chunk contains changes to a file that have not yet been saved back to the permanent copy stored on a file server machine. The Cache Manager first checks the least-recently used chunk. If it is not dirty, the Cache Manager discards the data in that chunk. If the chunk is dirty, the Cache Manager moves on to check the next least recently used chunk. It continues in this manner until it has created a sufficient number of empty chunks. Chunks that contain data fetched from a read-only volume are by definition never dirty, so the Cache Manager can always discard them. Normally, the Cache Manager can also find chunks of data fetched from read/write volumes that are not dirty, but a small cache makes it difficult to find enough eligible data. If the Cache Manager cannot find any data to discard, it must return I/O errors to application programs that request more data from AFS.

### 2.2.3 Coda

AFS and similar systems are vulnerable to server and network failures. Coda [5] was developed based on AFS to provide reliability as well as the availability of AFS while supporting disconnected operations. It was initially felt that server replication was the solution to providing higher reliability and fault tolerance. Disconnected operations, however involved hoarding, emulation and reintegration upon recon-

nection. The need for server replication seemed unnecessary. But in fact both these mechanisms compliment one another [18].

A distinction is made between first class replicas on servers and second class replicas(cached copies) on clients. While first class replicas are of better quality and are the reason for system reliability, second class replicas allow supporting disconnected operations, even as quality suffers. This is implemented as a user level cache manager Venus on the client side. Whole file caching is performed at the clients on their local disks with block caching being performed in their main memory buffer caches. It uses callback based cache coherence where the server maintains what objects have been cached by the client and notifies it when another client updates on of those objects. Disconnected operations involved hoarding which can be done manually or using tools such as SEER [7].

### 2.2.4 Sprite

This system was developed at Berkeley with the goal of improved file sharing. Server detects when two users are concurrently writing to the same file and disables caching of that file on both machines. When files are not shared, writes are delayed 30s before writing back to the server. Sprite's file caches change size dynamically in response to the needs of the file and virtual memory system. Baker [13] showed about 60% of all data bytes read by applications are retrieved from client caches without contacting file server. Sprite's 30 second delayed-write policy allows 10% of newly written bytes to be deleted or over written without being written back from the client cache to the server. Sprite's network file system

provides a single system image: there is a single shared hierarchy with no local disks.

The file system uses large file caches in both clients and servers and ensures cache concurrency even in case of concurrent write access. A least-recently-used mechanism is used to choose blocks for replacement in Sprite. According to the Baker study on average blocks have been unreferenced for almost an hour before they get replaced. On server side can consume entire physical memory but on client side(the side that concerns us) can consume between one third to one fourth of physical memory. This is significantly larger than size used by the unix kernel at the time(1991). The Baker study says 50% of traffic is filtered out thanks to the client cache.

### 2.2.5 xFS

Berkeley's eXperimental file system [2] optimizes AFS. When a file is modified, it is not written back to the server. The server is informed of the modifications and the client becomes the owner of the file. The next request for this file is forwarded to new owner. In such systems server caches don't need to be relied upon much, though there exists the overhead of maintaining state on who has what.

### 2.2.6 Ficus

Ficus [26] was developed at UCLA and supports disconnected operations, which means a client can continue operating when totally disconnected from the rest of the network. This is achieved using a sophisticated predictive caching [7].

## 2.3 Factors Effecting Caching in Distributed File Systems

Having looked at the way most of todays distributed file systems implement caching the following list of issues needs to be considered while designing an effective caching strategy.

**Cache on disk or memory?** While a main memory cache implies reduced access time, a disk cache implies increased reliability and autonomy of client machines. The creators of Sprite [8] give the following reasons why main memory caches are better. Main memory caches allow clients to be diskless hence cheaper. Quicker access from main memory than disk. As memories get larger main memory caches will grow to achieve even higher hit ratios.

**Cache on server or client?** When faced with a choice between having a cache on the server node versus the client node, the latter is always preferable because it also contributes to scalability and reliability. Replacement strategies that rely on temporal locality such as LRU suffer when implemented on server caches. Fileserver caches differ, both in comparison to the client caches with respect to the characteristics of the disk reference streams that they handle. These reference streams differ greatly from those generated by the applications themselves as all references satisfied by client caches have been removed. This filtering of the reference stream destroys much of its temporal locality [27]. LFU performs better except for small client caches which dont remove all temporal locality. Thus the choice of replacement pol-

icy at fileserver as well as at client caches will continue to be an important issue. If different policies on client and server are choosen, their effects on one another must be studied. Some have argued that client caches are now becoming so large that they can hold the complete disk block working sets of their users and thus that the ability of a fileserver cache to hold portions of these working sets is irrelevant. But at the same time, increased file sizes seen in current day workloads could overwhelm smaller client caches.

**Whole file caching or block caching?** Whole file caching has several advantages. Transmitting an entire file in response to a single request is more efficient than transmitting it block by block in response to several requests as dependency on network conditions is minimum. It has better scalability as servers receive fewer access from each client. Disk access is faster when dealing with a whole file than with random blocks. Clients supporting whole file caching maybe more immune to server and network failures. The main drawback is sufficient space is required on client nodes to store large files in their entirety. Therefore it may not be the best strategy when clients are diskless workstations. Ameoba [9] and AFS-2 [17] are examples of systems that implement whole file caching. Whole file caching makes more sense in mobile environments when clients are in weakly connected or disconnected states. But block transfers can complete more quickly than file transfers. The less time a transfer needs to complete the sooner a user can resume working. This becomes very advantageous in low bandwidth scenarios where file transfers are very time consuming. Additionally the use of blocks as the caching unit

means that it is not necessary to transfer the entire contents of a file to the client if the whole file is not referenced. This also means client nodes need not have large storage space. NFS [1], LOCUS [28] and Sprite [8] implement block based caching. Byte-level caching has also been investigated. Its main drawback is the difficulty in implementing cache management due to the variable length data for different access requests.

**Variable or fixed size caches** Fixed size caches are easier to implement but dont make use of the free memory available on the disk or memory. Sprite [8] implements a variable size memory cache that occupies main memory along side the virtual memory system. The virtual memory system receives preference. A physical page used by virtual memory cannot be converted to a file cache page unless it has been unreferenced in the last 20 minutes [13].

**Performance Measurement?** The theoretical optimal replacement policy provides lowest miss ratios. Replacing blocks which will not be used for the longest period of time requires knowledge of the future and maybe impossible to implement but provides an upper bound on attainable performance. This would provides a measure of the goodness of other policies. Since we have information of future accesses in trace data this can be done to see how other policies measure up to an optimal strategy. Random replacement chooses all blocks with equal probability. This scheme would work well if all blocks were equally accessed. If ‘intelligent informed’ replacement policies cannot perform better, then there almost certainly are some underlying flaws in decision strategy. It provides a lower bound on performance. There is no reason

to use any policy which performs worse than this one. Most caches are designed and tested based on file system traces. Trace collection is almost always done in research environments, is this good enough?

**Caching and mobility** Cache management in a mobile environment is a significantly different problem and management policies need to be examined in that context if good performance is to be achieved. It can no longer be assumed that a fast network connection to a file server exists and thus policies must be designed to have a minimal reliance on communication. Policies should be designed to be adaptive in nature. That is for performance reasons it may be necessary to dynamically modify policies in response to changes in the operating environment. Finally since user behavior is strongly affected by perceived performance it might be expected that the access patterns of a disconnected user will be different from that of a strongly connected user. Since any good cache management policy relies on the knowledge of the patterns of requests which occur, it may be advantageous to reject this change in file request traffic in cache management policies.

**Transparent and Translucent caching** Almost all current file systems perform cache management without user interference. Exposing too much detail to the user requires the user to learn more about the systems functioning, to be able to control it well. But the aim in almost all cases is to make system functioning as transparent to the user as possible. This would allow the user to focus on his or her work rather than system management.

**Hoarding and Prefetching** Hoarding is scheme to allow mobile client to continue

working in case of disconnection. Hoarding is a relatively infrequent operation performed only at clients request prior to disconnection, timing is not critical. Prefetching is mainly concerned with improving performance and timing is important. In prefetching, file server is assumed to be still accessible although network connectivity may be weak. A cache miss is much more catastrophic in disconnected operations hence hoarding is typically willing to overfetch more in order to enhance availability of files.

File prefetching [20, 21, 22, 23] is done using semantic structures such as access trees that capture potentially useful informaion concerning the inter-relationships and dependancies between files. Two advantages of prefetching are applications run faster as they hit more in the cache, and second less burst load is placed on the network as prefetching is done when bandwidth is available. The disadvantages are CPU cycles are wasted on when and what to prefetch. Network bandwidth and server capacity is wasted when prefetching decisions prove less than perfect. Timeliness is another cruical factor in determining how effective prefetching turns out. The time interval between, the system being about to perform an I/O operation and actually doing so in most cases is very small. Prefetching must be done within this time.

**Periodic Removal** Another issue that needs consideration is periodic removal [29] versus on demand replacement. If a cache is nearly full replacement on demand would involve replacement on nearly every request there after meaning higher overhead costs. Periodic removal of files to ensure availability cache space and reduced overhead might be a solution. But periodic removal would

decrease number of files in the cache and hence hit rates.

**Complexity** A good argument supporting why caching must not rely on simplistic approaches such as LRU, LFU and size is presented by Dix [30]. It is argued that caching should be a cooperative activity between the system and the user. Dix illustrates this point with an interesting scenario.

> "Imagine a field engineer is accessing layout diagrams for a faulty electricity sub-station, half way through the repair the communication goes down and relavant part of the plans aren't on the local machine. A cache miss might cause several minutes delay. In the mean time which was the 10,000 volt cable?"

Because cost of failure is higher and time scales are longer, caching algorithms can afford to be *more complex*. Coda [5, 18] allows the user to set up a preferences on which files should be permenantly cached. Lotus notes gives the user control over replication policy. Ebling [24] implements *translucent* cache management for Coda. Critical aspects of caching are exposed to the user while noncritical aspects are hidden to maintain usability. This scheme makes use of a user patience model to evaluate if a user should be burdened with caching decisions or not.

# Chapter 3

# File System Workload and Access Patterns

User behavior is dramatically different from what it was ten years ago. This is due to a variety of factors, such as faster processors, larger storage capacity, wide range of applications, high speed network connectivity, to name a few. Periodic analysis of user behavior and file access patterns is therefore *essential*, in making design choices and decisions [31] while designing the file systems of tomorrow. Characteristics of current user workload may be determined by tracing file system activity and analysing the collected traces. The traces also serve as a workload over which to test experimental systems.

We describe the DFS and MIST traces in this section that have been used as the workload in our caching experiments. The older DFS Traces are compared with a set of newly collected traces, the MIST traces. It helps in understanding how user behavior has changed over the years and also gives us clues on what

to expect in future. The studies in this section lead to the design of new caching algorithms described in Chapter 4.

## 3.1 DFS Traces

The DFS Traces [11], collected from Carnegie Mellon University, were used as the workload to run simulations, testing the proposed semantic caching algorithms.

During the period from February'91 to March'93 the Coda project collected traces of all system call activity on 33 different machines. The publicly available DFSTrace reading library was made use of to handle the trace files. Seven different traces files (two from Mozart, Ives and Barber each and one from Dvorak [1] ) were analysed. The machine Barber was a server with the highest rate of system calls per second. Ives had the largest number of users, and Mozart was selected as a typical desktop workstation. Statistical data characterizing each of the seven traces used are presented in the Table 3.1.

[1]Mozart, Ives, Barber and Dvorak are machine names

| Trace | Opens | Closes | Duration(hrs) | Files |
|---|---|---|---|---|
| Mozart1 | 25890 | 33953 | 49.43 | 709 |
| Mozart2 | 93575 | 126756 | 162.83 | 1644 |
| Dvorak1 | 122814 | 196039 | 121.75 | 4302 |
| Ives1 | 41245 | 55862 | 75.70 | 247 |
| Ives2 | 26911 | 36614 | 48.81 | 686 |
| Barber1 | 30876 | 42155 | 52.12 | 725 |
| Barber2 | 14734 | 20005 | 23.99 | 592 |

Table 3.1: Statistics of workload used.

## 3.2 MIST Traces

Studies have been conducted on existing file systems [13, 14] but not periodically enough to reflect constantly changing workloads. This is probably due to the difficulty of obtaining trace data, and also to the large volume of data that is likely to result. As the DFS traces were a decade old it was assumed, and later proven correct, that access patterns have since changed. A new set of traces needed to be colled to study these changes. A kernel level tracing system was developed to collect more current data.

The study presented here is the detailed characterization of file access at the kernel level generated by six different machines over a period of a month. Subsets of the traces used for analysis covered periods ranging from a few hours to few days. All the traced machines had the same configuration Pentium 4, 2.2GHz with 512MB of RAM running patched Redhat Linux 8.0(kernel 2.4.18-14) to support system call interception.

### 3.2.1 Trace Collection

To monitor user activity at the filesystem level system calls such as `open()`, `close()`, `read()`, `write()` etc., need to be logged. This can be done by system call interception. Logging is accomplished with the use of a kernel module. With this module access is gained to the kernel space of the machine, whose activity is to be logged. Using this access, all `open()` and `close()` system call activity and related data is captured. The System Call Table accessible only in kernel space provides a list of function pointers to all the system calls. System call intercep-

tion involves replacing the default `open()` and `close()` function in the kernel's System Call Table with new ones.

When a process calls a standard file open function in user space, a system call is made. This call maps to an index offset in the System Call Table array. As the kernel module modifies the function pointer at the open index to point to its own implementation, the execution switches into kernel context and begins executing the new open call. This is shown in Figure 3.1.

The module has been designed so that information generated can be either logged to a file or by the syslogd daemon. The structure of the information logged is shown in Table 3.2 and a sample of the trace data is shown in Figure 3.2. Though system call interception has its advantages, it can be used to seriously compromise system security. Reprogramming system calls without a complete understanding of all aspects of system functioning can easily produce race conditions affecting system stability.

Due to these reasons most linux kernels starting from 2.4.10 onwards prevent modules from using the system call table. To overcome this all the kernels being used had to be modified and recompiled to allow system call interception. These modified kernels allowed the tracing module to be loaded. Care was taken while implementing the new open and close system call, to keep the implementation simple while extracting as much information as possible.

We considered logging other potentially interesting calls such as `read()`, `write()`, `execve()`, `seek()` etc, but did not implement it for two reasons. Firstly, large amounts of data were generated very fast and secondly tracing had to be done without the user experiencing any sort of performance degradation. Therefore the

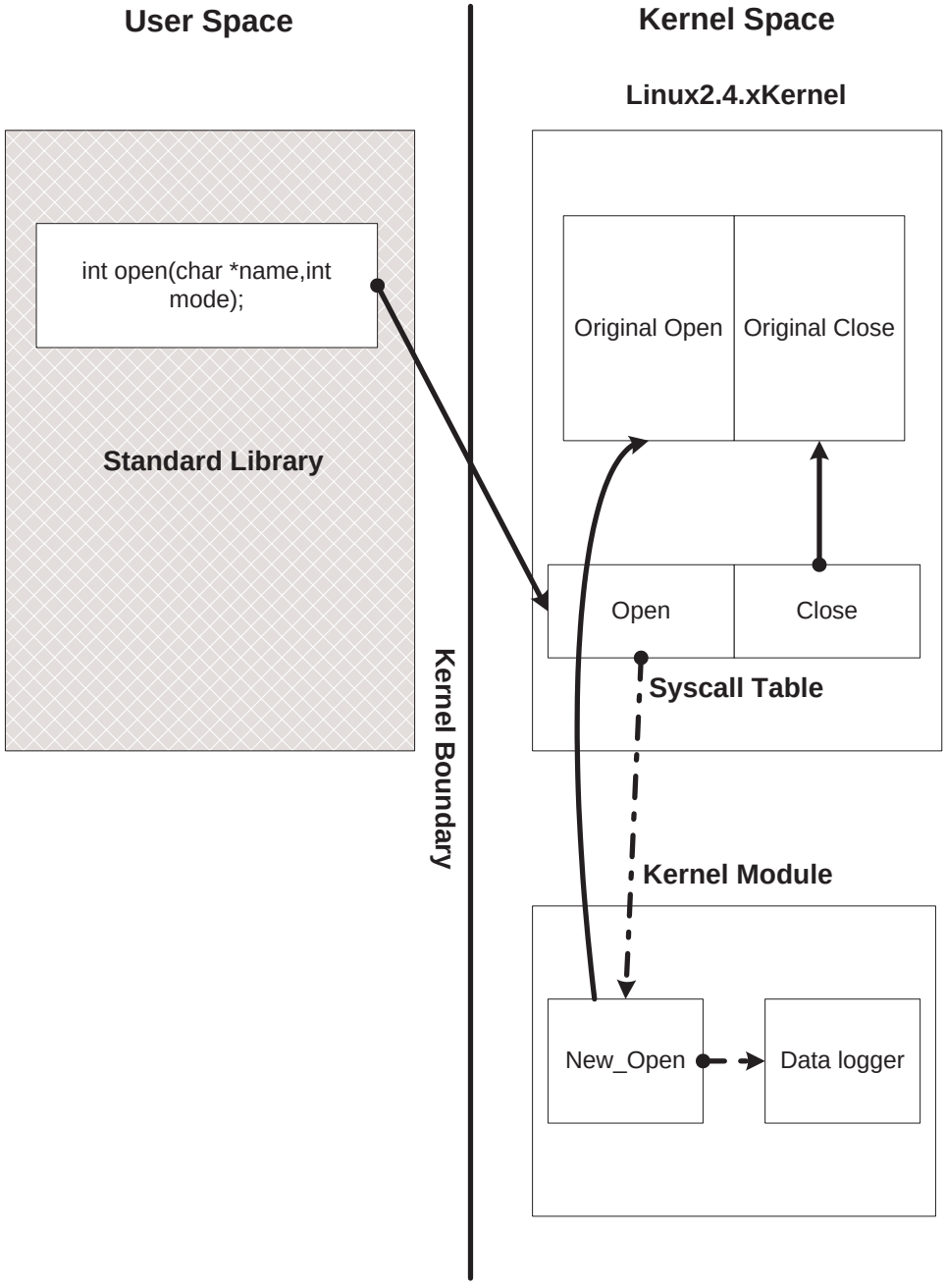



Figure 3.1: The conceptual representation of open system call redirection.

```
1081951980 440964 close 5 /usr/lib/perl5/5.8.0/File/Spec.pm 8716 chbg.pl 100444
1081951980 441058 open 5 /usr/lib/perl5/5.8.0/File/Spec/Unix.pm 12053 chbg.pl 100444 100000
1081951980 441274 open 6 /usr/lib/perl5/5.8.0/i386-linux-thread-multi/Cwd.pm 14246 chbg.pl 100444 100000
1081951980 441741 open 7 /usr/lib/perl5/5.8.0/base.pm 2437 chbg.pl 100444 100000
1081951980 442579 close 7 /usr/lib/perl5/5.8.0/base.pm 2437 chbg.pl 100444
1081951980 448562 close 6 /usr/lib/perl5/5.8.0/i386-linux-thread-multi/Cwd.pm 14246 chbg.pl 100444
1081951980 448701 open 6 /usr/lib/perl5/5.8.0/i386-linux-thread-multi/XSLoader.pm 3836 chbg.pl 100444 100000
1081951980 449912 close 6 /usr/lib/perl5/5.8.0/i386-linux-thread-multi/XSLoader.pm 3836 chbg.pl 100444
1081951980 450390 open 6 /usr/lib/perl5/5.8.0/i386-linux-thread-multi/auto/Cwd/Cwd.so 84576 chbg.pl 100555 0
1081951980 450451 close 6 /usr/lib/perl5/5.8.0/i386-linux-thread-multi/auto/Cwd/Cwd.so 84576 chbg.pl 100555
1081951980 454499 close 5 /usr/lib/perl5/5.8.0/File/Spec/Unix.pm 12053 chbg.pl 100444
```

Figure 3.2: Sample from kernel trace collected

system had to be designed keeping these constraints in mind. A shell script was used to collect the traces from the various machines being monitored using ssh and scp. A small suite of programs and scripts were developed to analyse the collected data. All access to the *proc* file system was filtered out as it is a virtual file system and its files aren't being stored on disk.

### 3.2.2 Observations

We have based most of this study on the eight traces sumarized in Table 3.3. An attempt was made to characterize the traces based on the following paramters; File size, Time to next access, Open time, Access Count and Activity (Refer Appendix A for all graphs).

The BSD studyin 1985 [14] showed 75% of files were open less than one-half second. Baker in 1991 [13] says 75% of files are open less than one-quater

| Field Name | Data Type | Description |
|---|---|---|
| Seconds | Long | Time in seconds when system call was logged |
| Microseconds | Long | Time in microseconds when system call was logged |
| Operation | char | System call (either `open` or `close`) |
| Descriptor | int | File descriptor of file being used |
| Name | char | File name |
| Size | long | Size of file being used in Bytes |
| Process | char | Name of the process that made the call |
| Flag | octal | Describes file attributes, applies only to `open` |
| Mode | octal | Describes mode of the file (read-only, read-write, etc.) |

Table 3.2: Structure of a Mist Trace record.

second. The MIST traces show 75% of accesses are open less than a hundredth of a second as shown in Figure 3.3. Similar results are seen on the other MIST traces, refer Appendix A. Baker concludes that this is due to increasing processor speeds. But another conclusion that can be drawn is, that the majority of files being accessed are being used by the system rather than the user, due to the imperceptible time periods involved. An example of this is when a user wants to edit a document with a word processor or listen to music with a media player. Even though only a single file, the document or the song is being opened, a large number of system files are being accessed in the background by the application. These files maybe shared libraries, codecs, font files, styles sheets, drivers etc. These files are open for very short periods and form the majority of accesses being made. This could be a very useful parameter in distinguishing files opened

| Trace | Opens | Closes | Duration(hrs) | Files |
|---|---|---|---|---|
| Kewei (Apr14) | 81492 | 83517 | 15.73 | 2230 |
| Sharun (Apr14) | 81670 | 86716 | 16.10 | 2251 |
| Siva (Apr14) | 59229 | 67179 | 50.06 | 2593 |
| Zhu (Apr14) | 48107 | 52585 | 49.85 | 3605 |
| Sharun (Apr16) | 398542 | 421723 | 75.57 | 4019 |
| Zhu (Apr16) | 85996 | 94889 | 75.52 | 7232 |
| Kewei (Apr16) | 67764 | 76420 | 76.08 | 4683 |
| Siva (Apr16) | 65577 | 70848 | 76.23 | 2542 |

Table 3.3: Traces used in this study

by a user and those opened due to system activity. In certain scenarios every single system file need not be cached and in such cases being able to distinguish between user and system files could be very useful.

The majority of files being accessed are read-only, application specific having very short open times. This trend will continue as applications grow more complex and processors more fast. Another important aspect that was looked into was file popularity. Access count tells us how many times a file was accessed over the length of the tracing period. Figure 3.4 shows almost 80% of the files were accessed less than 5 times. Similar results were obtained from all the other MIST

| Trace | Read | Read Write | Write | Large Files | | |
|---|---|---|---|---|---|---|
| | | | | Read-Only | Write-Only | Read-Write |
| Kewei (Apr16) | 44469 | 231 | 83 | 8945 | 3394 | 10022 |
| Kewei (Apr14) | 9537 | 106 | 64 | 71216 | 351 | 199 |
| Siva (Apr14) | 39328 | 329 | 9270 | 9445 | 441 | 366 |
| Siva (Apr16) | 50355 | 373 | 6854 | 6132 | 1187 | 657 |
| Sharun (Apr14) | 47583 | 1479 | 84 | 29100 | 685 | 1767 |
| Sharun (Apr16) | 231145 | 6909 | 159 | 147205 | 2648 | 5198 |

Table 3.4: Mode in which files were accessed.

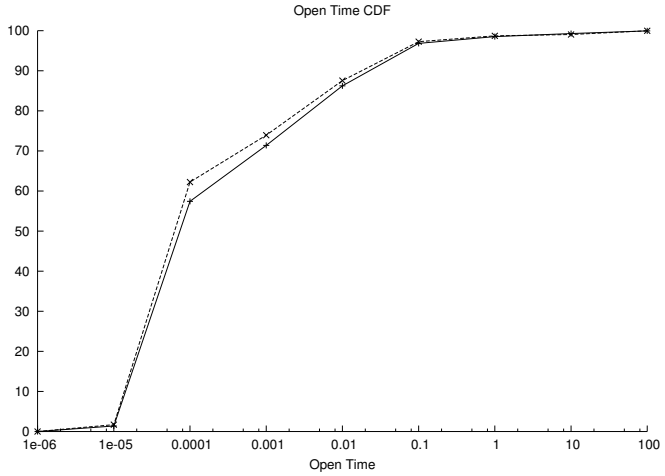



Figure 3.3: File open times. Cumulative distribution of the time files were open traces, refer Appendix A. The long tails of these graphs indicate there is small percentage of very popular files. Most of the popular file are system files such as configuration files or shared libraries.

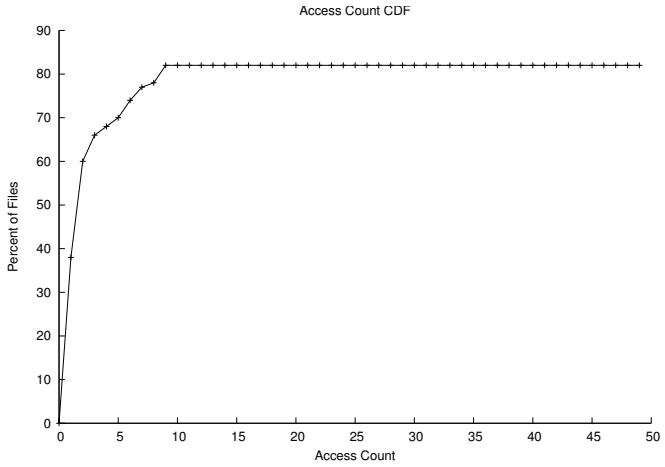



Figure 3.4: File access count. Cumulative distribution of number of times files were accessed over entire tracing period

It can be observered that most of the accesses are coming from the /usr /lib and /etc directories as shown in Figure 3.5 indicating how system files dominate user files. Of course this is not the best way of differentiating system and user files

but it gives a good indication of what is being accessed. Other studies [14, 32] also indicate majority of file accesses are not caused by user data but by program execution, swapping, metadata etc.

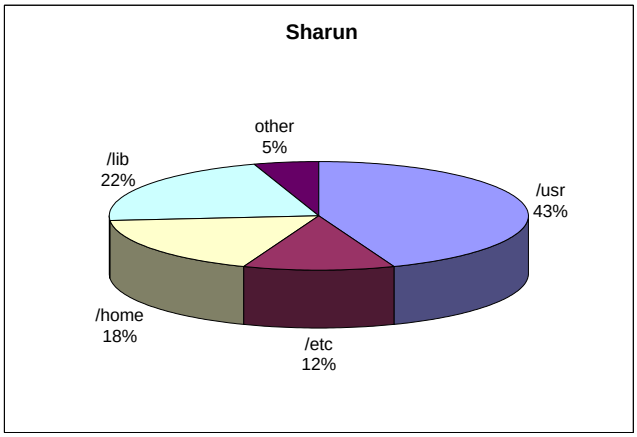



Figure 3.5: Distribution of access among standard filesystem directories

Figure 3.6 represents the cumulative distribution of file sizes when files were closed. It shows that 75% of all accesses were to files less than 100KB in size. Again this graph has a long tail indicating few accesses are to very large files. This is the general trend, 60-75% of all access were to files less than 100KB in size(refer Appendix A).

Looking at Figure 3.7 three levels of activity may be defined. There are periods of inactivity or less than 1000 files opened per hour, normal activity where 1000 to 5000 files are opened an hour and periods of high activity where greater than 5000 files are opened per hour. Changes from one level of activity to another can be seen in the graphs as user behaviour changes through the course of the day.

Some processes generate many more file opens than others. Processes such as update, which updates the search database and nautilus for file managment

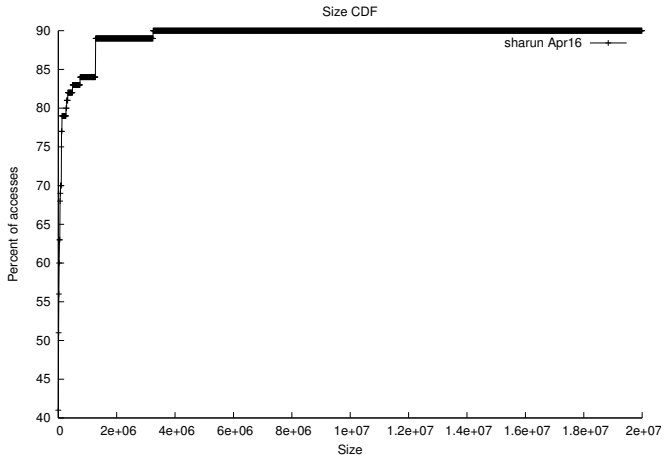
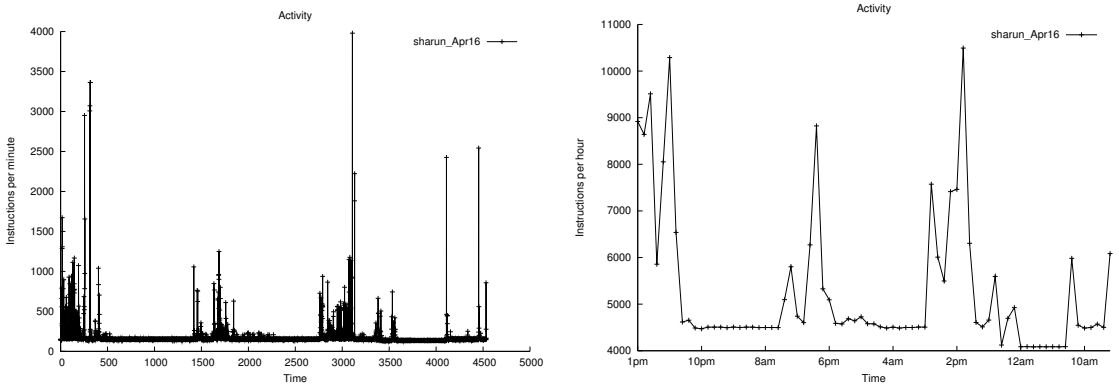



Figure 3.6: File size. Distribution of file sizes measured when files were closed




a b

Figure 3.7: User Activity.(a)Number of files accessed per minute over entire trace-length(b)Number of files accessed per hour over entire tracelength

open almost all files on the partition each time they are run. Word processing applications such as pdflatex and openoffice also generate a lot of opens while loading font files, shared libraries and style sheets. When the user isn't active, files continue to be opened by background processes and deamons. Another observation that can be made is that different users produce different amounts of activity depending on the applications used. Development and program testing can generate high activity while web browsing, multimedia and text editing produce normal activity. Designing caches to adapt to such changes in activity level is challenging. Two ways that this can be done is, through user participation in the caching processes or maximizing files available and allowing the user to work with what is available.

Significant differences in access patterns and user behavior were observed in comparison with the DFS traces. What interestingly hasn't changed is the precursor counts of files, we still see a majority of files have at most one or two very 'popular' precursors among their individual set of precursors. This is again because file accesses aren't random. Therefore it seems only logical to exploit this relationship between file accesses that conventional caches totally ignore.

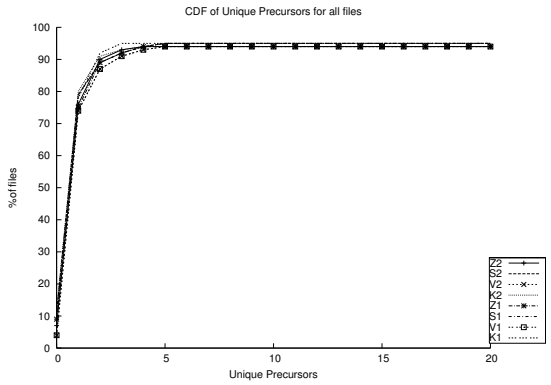



Figure 3.8: Cumulative distribution of the number of unique predessors

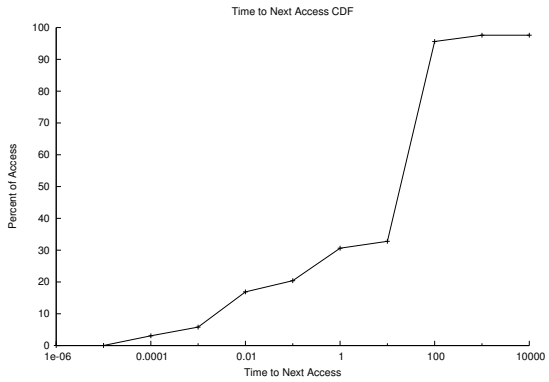



Figure 3.9: Time to next access

Knowledge on *when* a file is to be accessed next can be useful in determining how long a file should be allowed to stay in the cache. The *Time to Next Access* is defined here as the time between, when a file is closed and when it is next opened. In some studies it has been refered to as the inter access time. We measure this parameter for every access. Figure 3.9 shows that around 40% of the accesses resulted in an access to the same file within 10 seconds. In general half the accesses result in an access to the same file within a short time span. This means if a file stays in the cache for about 10 seconds they have a 50% chance of being accessed again. This is interesting as it means half the files, likely to be in the cache are to be accessed ‘soon’ while the remaining may not.

File throughput has definately increased over the last decade mainly due to much larger, large files. Having recorded only size at open and close actual throughput cannot be measured but an upper bound on throughput can be determined. This varies from trace to trace depending on what a user is doing and what time of the day it is as can be seen in Table 3.6.

| Traces | | | |
|---|---|---|---|
| V-Apr14 | V-Apr16-19 | Z-Apr14 | Z-Apr16-19 |
| evolution-mail | evolution-mail | mozilla-bin | mozilla-bin |
| nautilus | pdflatex | xscreensaver-ge | xscreensaver-ge |
| gs | sh | gs | sh |
| pdflatex | xscreensaver-ge | sh | gs |
| mozilla-bin | vim | fortune | pdflatex |
| xscreensaver-ge | fortune | nautilus | gvim |
| sh | sftp-server | pdflatex | nautilus |
| netscape-bin | netscape-bin | sftp-server | fortune |
| nautilus-adapte | mozilla-bin | xscreensaver | xscreensaver |
| eog-image-viewe | java | gweather-applet | scp |

Table 3.5: Top 10 Active Processes for two users on different days

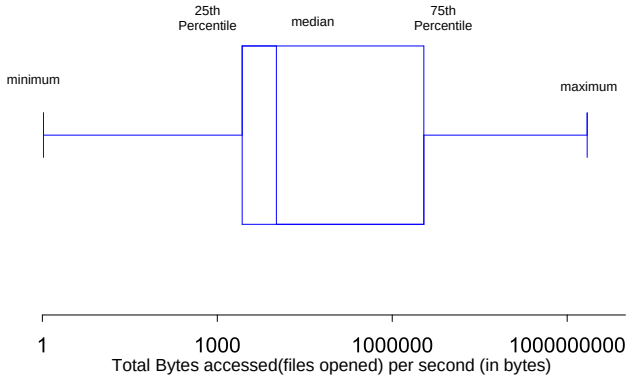



Figure 3.10: Throughput, in terms of size of files opened per second

| Percentile | Traces | | | |
|---|---|---|---|---|
| | S1 | M | Z | S2 |
| 10th | 0 | 0 | 0 | 2.8 |
| 25th | 1.3 | 2.6 | 2.5 | 52.5 |
| Median | 147.3 | 3.3 | 9.8 | 7356 |
| 75th | 12424 | 142.2 | 3336 | 7384 |
| 90th | 12605 | 4932 | 4935.3 | 7937.9 |

Table 3.6: Total Kilobytes accessed(from file open calls) per second.

# Chapter 4

# Semantic Caching

The basic idea of the proposed approach is motivated by the observation that file access isn't random. It is driven by user behavior and application programs. There exists a semantic relationship between two files in a file access sequence. We classify this relationship into two categories, *inter-file relations* and *intra-file relations*. An inter-file relationship exits between two files $A$ and $B$, if $B$ is the next file opened following $A$ being closed. $A$ is called $B$'s precursor. An intra-file relationship is said to exist between two files $A$ and $B$ if they are both open before they are closed. Our aim is to translate this relationship information into an eviction index based on which caching can be performed. The relationship may be 'strong' or 'weak'. Our caching replacement algorithm preserves the 'strong' while replacing the 'weak'.

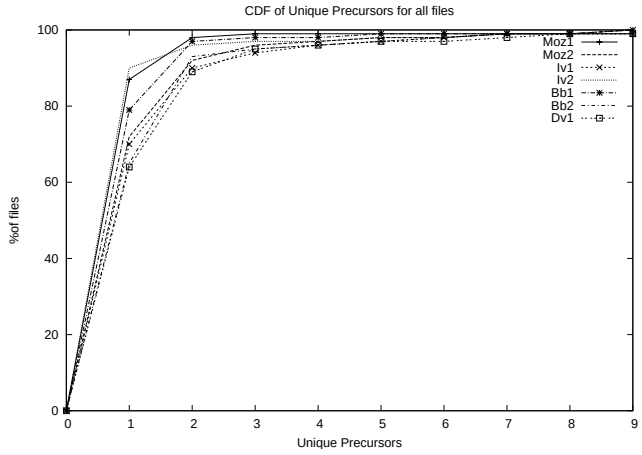



Figure 4.1: The cumulative distribution function of precursor count of file access.

## 4.1 Inter-file relations

To define an inter-file relationship, the information obtained by studying DFStraces [11] and our own file activity traces captured by system call interception is used [33]. We found something common in both traces, a large portion of the files examined have a small number of unique precursors (precursors rather than successors are considered as this information is easy to obtain and manage). In some cases 80% of the files have only one unique precursor as can be seen in Fig. 4.1. Similar results have been observed in other studies[34].

A heuristic parameter $INTER_i$ is defined to represent the importance of a file $i$ with respect to *inter-file* relations with it's precursors. The greater the importance the less likely it will be replaced and therefore it is used as an eviction index by our caching algorithm. The importance of the file is determined by the following factors.

$X_i$ - represents the number of times file $i$ is accessed.

$T_i$ - represents the time since the last access to file $i$.

$T_j$ - represents the time since the last access to file $j$ where $j$ is a precursor of $i$.

$Y_j$ - represents the number of times file $j$ precedes file $i$.

$$INTER_i = \frac{X_i}{T_i + \sum_{j=1}^{n}(T_j - T_i)\frac{Y_j}{X_i}} \tag{4.1}$$

The importance of file $i$ as shown in equation 4.1 is directly proportional to its access count and inversely to the time since its last access. The summation represents the strength of the inter-file relationship $i$ has with its precursors.

The strength of the inter-file relationship between $i$ and $j$ is not dependent only on the recentness of access of $i$ or $j$ represented by $T_j - T_i$ but also on the number of times $j$ precedes $i$ represented by $Y_j$. Therefore the greater the weight $\frac{Y_j}{X_i}$ more importance is given to the recentness. Consider the case where file $j$ (a popular precursor to file $i$ meaning $\frac{Y_j}{X_i}$ is relatively large ) has occurred more recently than file $i$, that is recentness $T_j - T_i$ is negative. This would reduce the summation value increasing the importance of file $i$. This is what is required, if $j$ has been accessed recently, $i$ is highly likely to be accessed next and must stay in the cache. Therefore files with stronger relationships are given more importance than files with weaker relations.

## 4.2 Intra-file relations

Intra-file relationships are those where both files are opened before they are closed. An intra-file relationship is based on the concept of *shared time*. Consider two files $i$ and $j$ that are both opened before they are closed and $i$ is closed before $j$ is

closed, then we define *shared time of* $i$ *with respect to* $j$, $S_{i,j}$, as the time between $i$'s close and the later of the two opens as shown in Equation 4.2. Shared time is calculated when file $i$ is closed. The intuition used here is, files that have relatively large shared time are likely to share time in the future.

$$S_{i,j} = C(i) - MAX(O(i), O(j)) \tag{4.2}$$

$where\ O(i)\ and\ C(i)\ are\ the\ open\ and\ close\ times\ of\ file$

$i\ respectively\ and\ C(i) < C(j)$.

A heuristic parameter $INTRA_i$ is defined to represent the *un*importance of a file $i$ based on it's *intra-file* relations, shown in Equation 4.3. This is opposite to the definition of importance based on inter-file relations explained above, in the sense that the higher its value the less important is the file. Here rather than depending on precursors, we depend on the files that shared time.

$$INTRA_i = T_i + \sum_{j=1}^{n} (T_j - T_i) \frac{S_{i,j}}{S_{total}} \tag{4.3}$$

$T_i$ - represents the time since the last access to file $i$.

$T_j$ - represents the time since the last access to file $j$ where $j$ is open before $i$ is closed.

$S_{i,j}$ - represents the shared time of file $i$ with respect to file $j$ where $i$ is closed before $j$.

$S_{total}$ - represents the total shared time with all files that are open before $i$ is closed

($\sum_{j=1}^{n} S_{i,j}$).

The strength of the intra-file relationship between $i$ and $j$ is dependent on the recentness of access of $i$ or $j$ represented by $T_j - T_i$ which is weighted by the relative shared time $\frac{S_{i,j}}{S_{total}}$. If $j$ has occurred more recently than $i$ and $S_{i,j}$ is relatively large, $INTRA_i$ is reduced or file $i$ is less *un*important. On the other hand if $j$ has occurred much before $i$ and $S_{i,j}$ is relatively large the unimportance of $i$ increases. In this way intra-file relations are preserved in the cache using $INTRA_i$ as an eviction index.

Both Equation 4.1 and Equation 4.3 are combined to produce a general measure of importance of the file based on inter-file as well as intra-file relations. It is defined as follows:

$$BOTH_i = \frac{X_i}{T_i + \sum_{j=1}^{n}(T_j - T_i)\frac{Y_j}{X_i} + \sum_{j=1}^{n}(T_j - T_i)\frac{S_{i,j}}{S_{total}}} \tag{4.4}$$

# Chapter 5

# Evaluation

This chapter describes the evaluation of the semantic caching algorithms proposed in the previous chapter. The design and working of the simulator is explained. To evaluate our approach, four other replacement policies were implemented for comparision purposes. Three different metrics of performance were chosen namely Hit rate, Byte hit rate and Files replaced. The complete simulation results for both DFS traces and MIST traces are presented in Appendix C and B respectively.

A simple file system client cache simulator was implemented to operate on both the DFS traces and the MIST traces which were described in Chapter 3. The cache itself is simulated using a hash table. Two additional lists are maintained, one keeps track of the currently open files and the other keeps track of closed files. The closed file list had a secondary purpose of maintaining relavant statistical information such as access count, number of unique predecessors, shared time, inter access time etc. The basic structure can be seen in Figure 5.1.

The simulator goes through the trace looking for the `open()` and `close()`

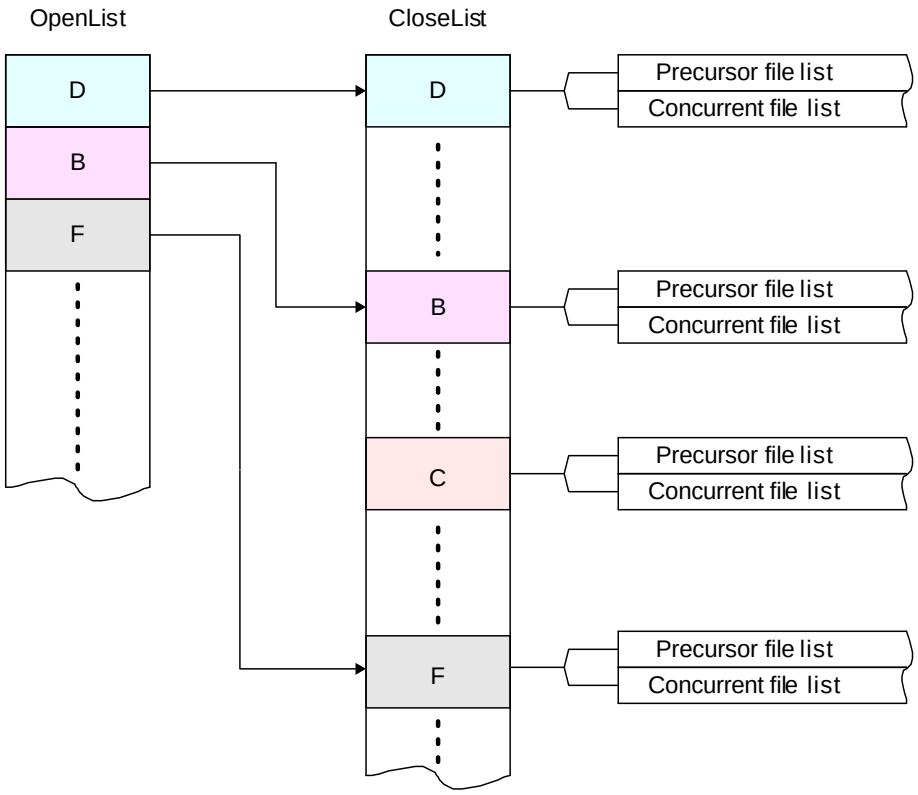



Figure 5.1: Basic data structure used by the replacement algorithm.

system calls. On encountering an `open()` the following operations are performed: First a check is performed to see if space is available in the cache. If it is, an entry corresponding to the file that has been opened, is added to the Open list, the Close list and the file itself is added to the Cache. The entry added into each list is different but share common information such as the filename, its size and the time it was opened. Every closed file also maintains a precursor list and access counts for each precursor in the list. Every new file opened is added to the end of the Close list. If it already exists in the Close List it's entries open time is updated with the new open time.

When out of cache space, files need to be replaced to accommodate the new file. To prevent large files from pushing many small files out of the cache we define a $MaxFileSize$ threshold. Any file bigger than this threshold value isn't brought into the cache. We set this threshold as 30% of the total cache size. After this check is performed the main replacement policy is enforced. An eviction index is calculated for each file in the cache. Depending on the policy the file with the biggest or smallest index is removed from the cache. If this doesn't create enough space to accommodate the new incoming file the file with the next biggest/smallest index is removed. This process repeats until enough space has been freed to accommodate the new file. Calculation of the index is dependent on the cache replacement policy and is explained in the following section.

## 5.1 Policies

This implementation looks at seven different policies, the first four are popular approaches against we which we evaluate our algorithm.

1. Round Robin (RR) - The eviction index is set to a constant value for all files. It provides a lower bound on performance. There would seem no reason to use any policy which performs worse than this one when considering computional or spatial overhead.

2. Least Recently Used (LRU) - This the most commonly used algorithm in almost all existing filesystem caches. The eviction index represents the time since the last access to a file. The file with the highest index is replaced.

3. Least Frequently Used (LFU) - This is based on the access counts of each file. The most popular files stay in the cache and the least popular files are replaced.

4. Greedy Dual-size (GDS) - This index is calculated based on the file size. Larger the file the smaller the index. File with the smallest index is replaced. We use the inflation value defined in [19] to keep frequently accessed large files in the cache.

5. INTRA - The eviction index is calculated based on intra-file relations as defined in Equation 4.3.

6. INTER - The eviction index is calculated based on inter-file relations as defined in Equation 4.1.

```
Start:
Read Trace Record
If ( operation is OPEN )
        If ( file not in cache )
                If ( file_size < MAX_FILE_SIZE)
                        If ( file_size < free_cache_space  )
                                Add_To_Cache();
                        Else
                                Calculate_Index();
                                Replace();
                        Endif
                Endif
        Endif
        Add file to Open_List;
        Add file to Close_List;
Endif

If ( operation is CLOSE )
        Remove from Open_List;
        Update file information in Close_List;
Endif
Goto Start if not end of file;
```

Figure 5.2: Pseudocode used to implement the caching simulator

7. BOTH - The eviction index is calculated taking both inter-file and intra-file relations into consideration as defined in Equation 4.4.

Once the index for all the files are calculated, replacement is performed based on the eviction index until enough space is available. The new file is then added to the cache. The performance of the cache is studied by varying its size.

If Replacement policy is,

LRU)
Index = Time of last CLOSE
File with largest index evicted

RR)
Index = 1

LFU)
Index = Access Count
File with smallest index evicted

GDS)
Index = Inflation value + 1/Size
File with smallest index evicted

INTRA)
Index = based on shared time (refer equation)
File with largest index evicted

INTER)
Index = based on predecessor count (refer equation)
File with smallest index evicted

BOTH)
Index = INTER + INTRA (refer equation)
File with smallest index evicted

Figure 5.3: Replacement policies

## 5.2 Metrics

We evaluate our semantic-based approach ($INTRA$, $INTER$, $BOTH$) by comparison against four conventional caching algorithms ($RR$, $LRU$, $LFU$ and $GDS$). We used two parameters, *cache hit rate*, *byte hit rate* to measure the effectiveness of using file relationship information in making caching decisions against conventional approaches. It indicates the percentage of files or bytes accessed that can be handled by the cache. A third parameter *Replace attempts* was used as an indicator of communication overhead in a low-bandwidth distributed environment. This is because each replacement in a client, necessitates synchronization with a remote file server in a distributed file system. Another way we measure this overhead is looking at the *total bytes missed* over the tracing period. If the total bytes missed is low then the communication with the server is also low.

Due to space constraints only a small part of the evaluation is present here, more details can be found in the Appendix B.

## 5.3 Results

### 5.3.1 DFSTraces

The first part of our evaluation is done using the DFS Traces. The simulation results for traces ives1, ives2 and mozart2 have been presented here. The simulation results of the reminaing traces can be found in Appendix C. Simulation has been performed with three different cache sizes 10K, 25K and 100K. The reason for such small cache sizes is the age of the DFS Trace. It represents an older workload with

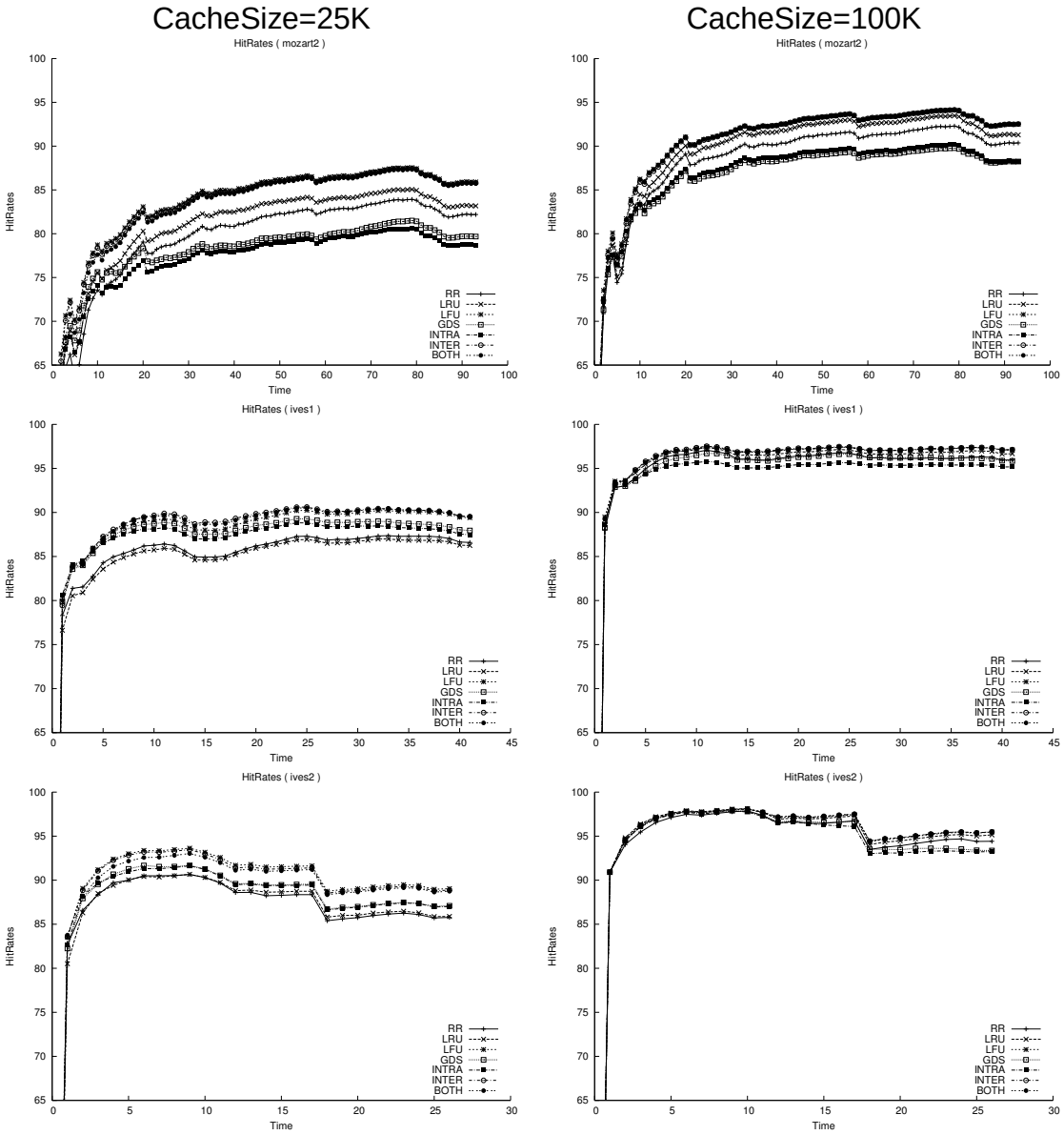



Figure 5.4: Hit rates recorded during the simulation using trace files mozart2, ives1 and ives2 with caches of size 25K and 100K resp.

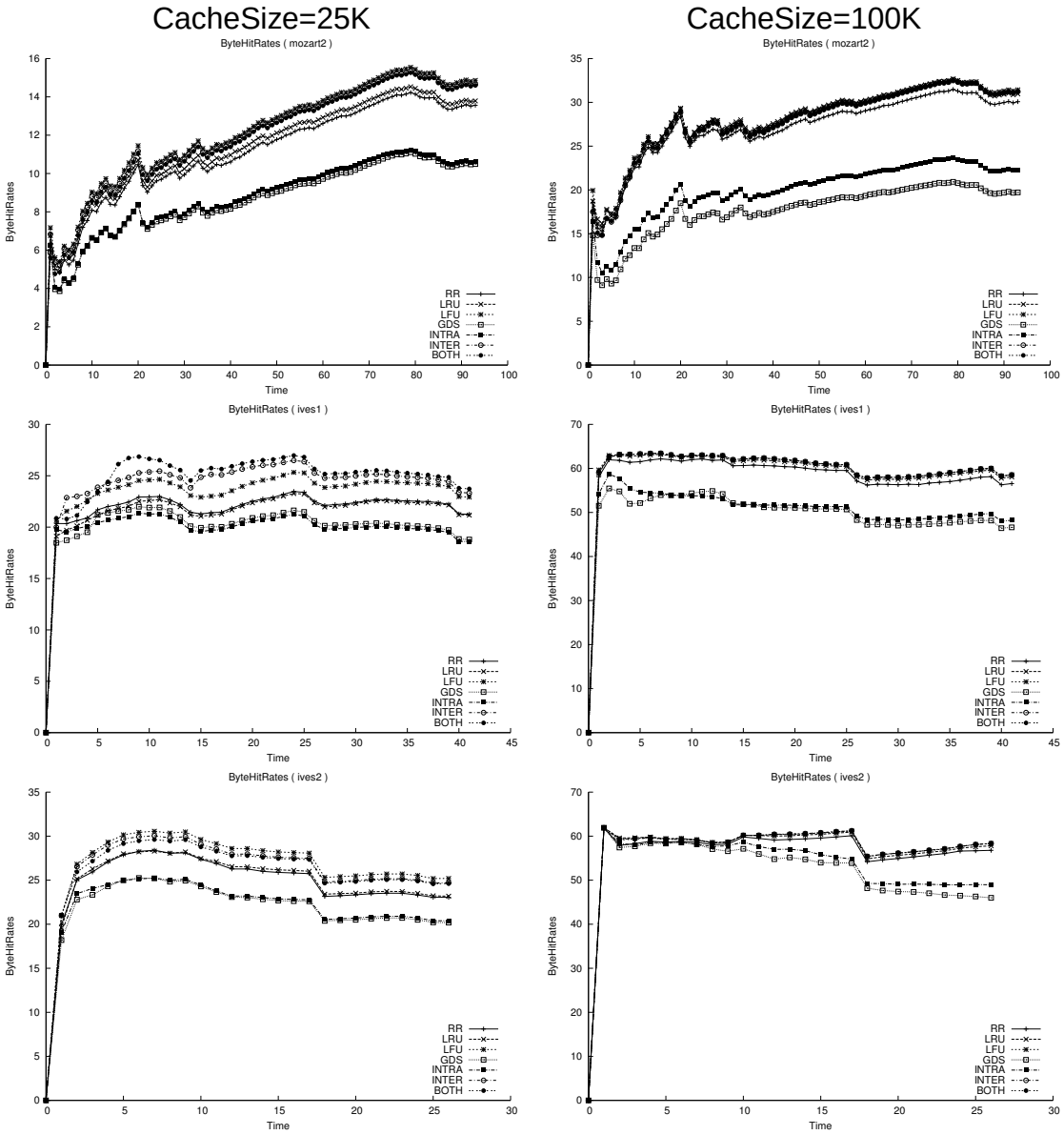



Figure 5.5: Byte Hit rates recorded during the simulation using trace files mozart2, ives1 and ives2 with caches of size 25K and 100K resp.

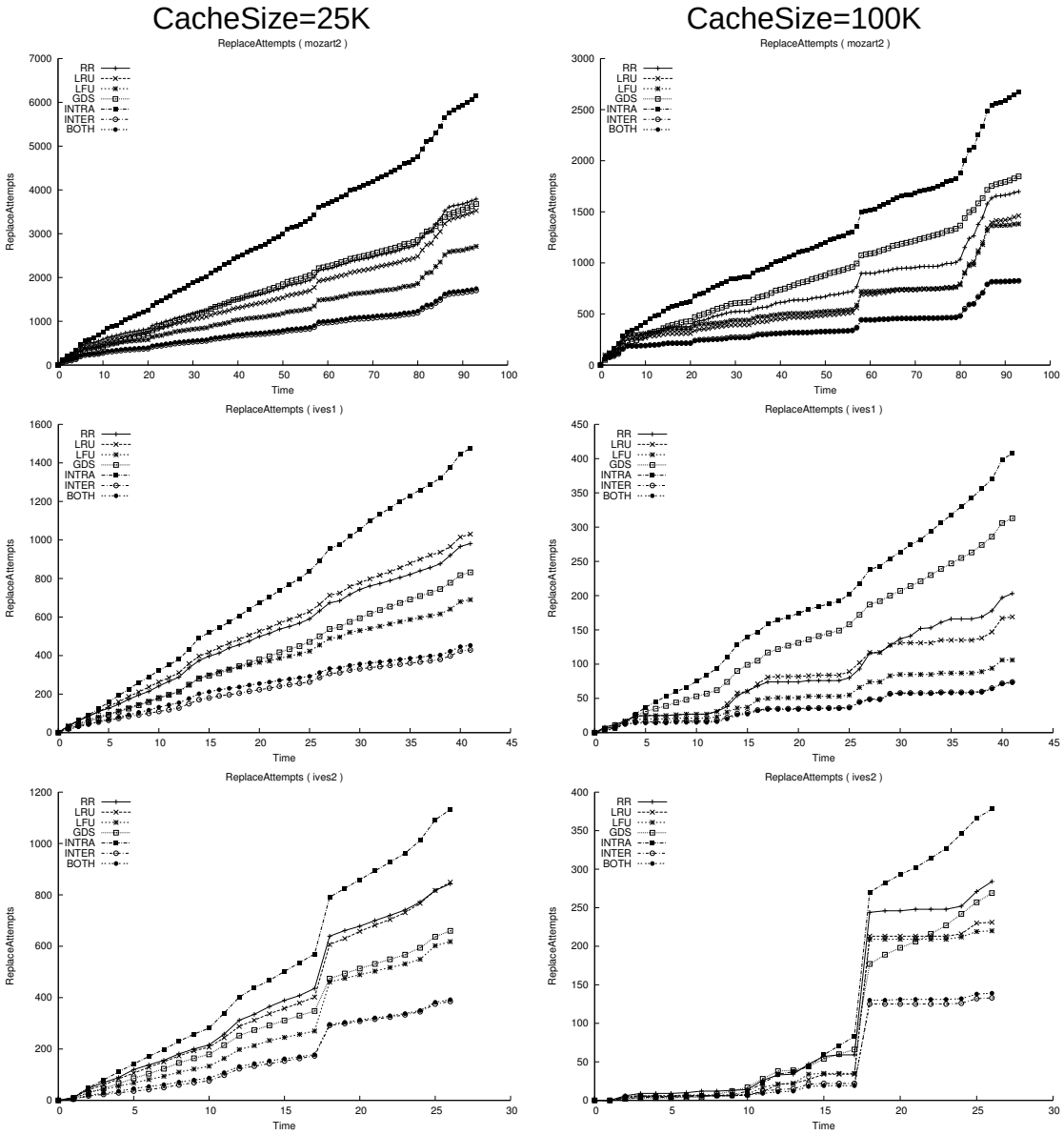



Figure 5.6: Replace attempts recorded during the simulation using trace files mozart2, ives1 and ives2 with caches of size 25K and 100K resp.

much smaller file sizes than seen today. It has been used, however, as it remains one of the most popular traces being used by the research community. The x-axis of these figures represent the number of file requests that have been processed, and the y-axes report hit ratio, byte hit ratio, and file replaced respectively. Each parameter is measured after every thousand requests handled. As can be seen $INTER$ and $BOTH$ exhibit the highest hit ratios and byte hit ratios, and also the lowest number of replace attempts.

In general, in terms of hit ratios, $INTRA$ performs very badly and this we feel is because of the very low number of intra-file relations present in the DFS traces, which means file access is generally sequential in nature. $INTER$ or $BOTH$ in almost all cases [33] have the best hit ratios which validates our belief that file relations should be taken into consideration while making caching decisions.

### 5.3.2 MIST Traces

In this section the simulation results using the MIST Traces have been presented. The hit rates, byte hit rates, files replaced and total bytes missed have been examined in the following tables. The remaining simulation results have been presented in Appendix B. In general INTER show better hit rates than all conventional algorithms. As of now most files accessed are small and the few large files accessed are very large. GDS also performs well as it takes advantage of this fact. It always replaces the largest file in the cache creating a lot of space each time. When there are more large files in the cache INTER has the advantage as it considers relationships. This could be the reason in almost all cases INTER performs better than

GDS in terms of hit rate.

Bytehitrate in general is good only for caches above 5-10Mb in size as this allow larger files to be cached longer. A 50 Mb cache seems to be sufficient for most of todays workload though in some cases of high activity involving large files (installing packages) bigger caches may be required. The reason why file relations should be considered an important factor are the results from the replace attempts metrics. For example in Table 5.3 in comparision with a conventional algorithm like LRU for a 10MB cache which replaces 450 files, INTER replaces 16 files. Though GDS replaces only 9 its hitrates are still less than INTER.

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 71.31 | 69.30 | 69.15 | 68.05 | ***72.77*** | 69.67 | 69.50 |
| 500K | 74.85 | 76.32 | 75.87 | 76.36 | ***78.46*** | 76.77 | 75.90 |
| 1Mb | 79.17 | 78.55 | 79.03 | 77.02 | ***81.71*** | 78.23 | 78.73 |
| 5Mb | 85.81 | 85.95 | 85.71 | 85.83 | ***86.48*** | 86.06 | 85.92 |
| 10Mb | 92.89 | 92.76 | 92.72 | 92.96 | ***93.57*** | 92.77 | 92.60 |
| 50Mb | 94.96 | 94.96 | 94.95 | 94.92 | ***95.52*** | 94.96 | 94.96 |

Table 5.1: Hit Rates of trace Siva (Apr16). It can be seen that INTER performs better than the other replacement policies for all tested cache sizes.

As the goal is to minimize communication, a simple test was performed to check how much communication is generated due to replacements as shown in Figure 5.7. If we assume that replacements result in write backs of the files replaced that have since changed, by examining the total “excess” bytes replaced

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 1.30 | 1.13 | 1.11 | 1.03 | ***1.38*** | 1.17 | 1.14 |
| 500K | 1.74 | 2.20 | 2.10 | 2.17 | ***2.61*** | 2.33 | 2.08 |
| 1Mb | 4.06 | 3.78 | 4.01 | 2.48 | ***6.25*** | 3.30 | 3.91 |
| 5Mb | 13.19 | 13.68 | 13.06 | 12.97 | 13.03 | 13.67 | ***13.70*** |
| 10Mb | ***40.77*** | 40.35 | 40.31 | 40.67 | 40.65 | 39.92 | 39.86 |
| 50Mb | 98.38 | 98.38 | 98.37 | 97.31 | ***98.41*** | 98.30 | 98.36 |

Table 5.2: Byte Hit Rates of trace Siva (Apr16). It can be seen that a 10Mb cache produces above 90% hit rate regardless of policy which corresponds to only 40% of the bytes being accessed. While a 50Mb cache is able to hold most of the bytes being accessed.

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 62 | 141 | 117 | 27 | 25 | 31 | 142 |
| 500K | 168 | 244 | 227 | 33 | 43 | 39 | 148 |
| 1Mb | 376 | 399 | 282 | 32 | 44 | 39 | 238 |
| 5Mb | 342 | 334 | 324 | 17 | 17 | 20 | 198 |
| 10Mb | 450 | 333 | 308 | 9 | 16 | 12 | 194 |
| 50Mb | 275 | 91 | 32 | 1 | 0 | 1 | 13 |

Table 5.3: Files replaced over the length of the trace period of trace Siva (Apr16). It is quite clear that the number of files replaced are much less for INTER,INTRA and GDS in comparision to conventional caching algorithms like LRU and LFU.

we can get an upperbound on the communication due to replacement. As the minimum bytes required to be replaced is equal to the file coming into the cache, the "excess" here refers to the bytes replaced beyond this value. Files whose size has been changed have been considered. It was observed that files needed to be written back to the server due to replacement wasn't significantly large. Type specific communication optimization techniques [12, 35] may be used to further minimize the communication overhead.

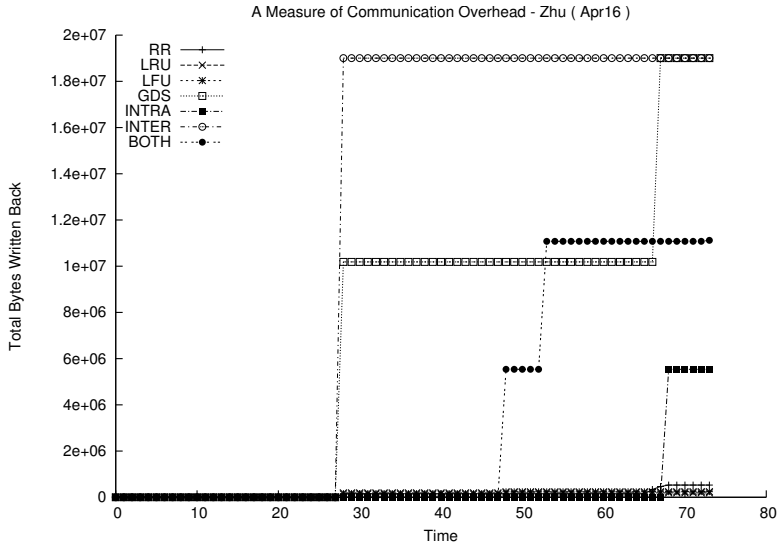



Figure 5.7: This figure shows the total number of bytes that may be written back due to replacement over the trace duration for trace Zhu (Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 559 | 675 | 717 | 379 | 504 | 418 | 596 |
| 500K | 3160 | 3123 | 2969 | 2354 | 2608 | 2259 | 2201 |
| 1Mb | 6996 | 6603 | 5898 | 4475 | 4105 | 5343 | 5262 |
| 5Mb | 10755 | 16856 | 15665 | 4991 | 8468 | 10651 | 9345 |
| 10Mb | 21164 | 16655 | 14277 | 6335 | 13185 | 11443 | 8609 |
| 50Mb | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 5.4: Total Kilobytes missed over trace duration for trace Siva (Apr16). It can be seen that INTER performs better than LRU in all cases. GDS performs better in some cases for this trace as activity is low and number of unique files being dealt with is much less than in other cases. The last line implies bytes missed is less than 1KB

# Chapter 6

# Conclusion

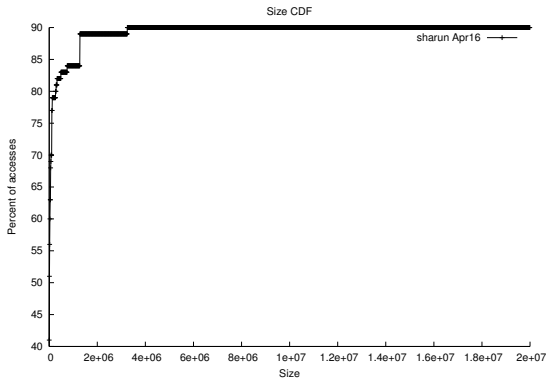
A trace collection module has been developed. The tracing is completely transparent to the users and adds no noticeale load to the system It has been used to collect workload that reflects present day user behavior and access patterns. The traces provide more current information to people investigating file system design. This workload allows us to practically test new ideas and concepts for future file systems. An analysis of this workload provided us the information to define inter and intra file semantic relationships. We have presented a semantic-based caching algorithm and shown that it performs better than conventional caching approaches in terms of hit ratio and byte hit ratio. We have also shown that it does this performing far fewer replacements. Compared to prevalent replacement strategies that ignore file relations and communication overhead, this approach would seem to better suit distributed file systems that operate across heterogeneous environments, especially with low-bandwidth connections such as cegor.

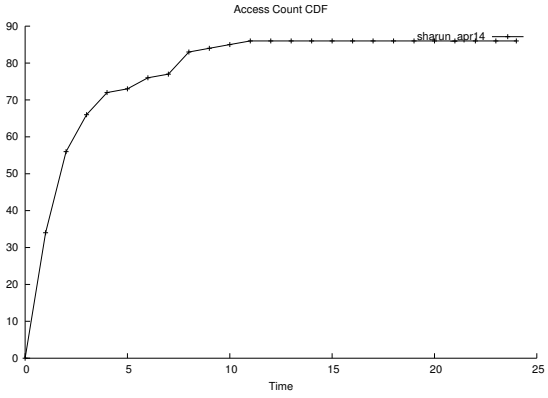
# Appendix A

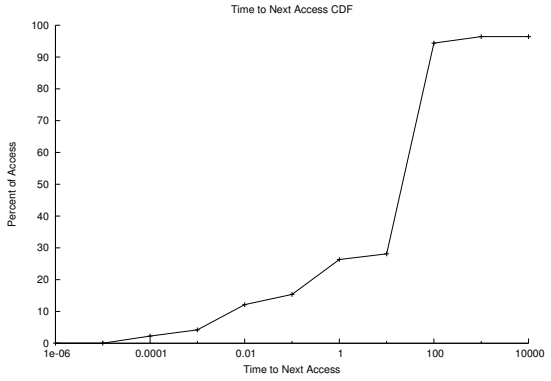
# Analysis of Mist Traces

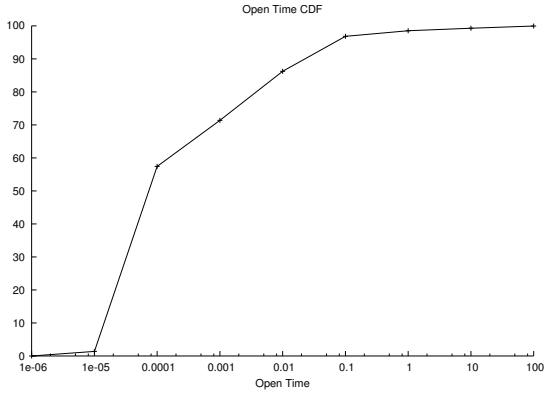
| Traces | | | |
|---|---|---|---|
| K-Apr14 | K-Apr16-19 | S-Apr14 | S-Apr16-19 |
| nautilus | mozilla-bin | chbg.pl | chbg.pl |
| mozilla-bin | wish | gconftool-2 | gconftool-2 |
| sh | xscreensaver-ge | sh | sh |
| gnome-panel | nautilus | nautilus | nautilus |
| gs | sh | netscape-bin | netscape-bin |
| fortune | gs | gconfd-2 | gconfd-2 |
| ggv-postscript- | ggv-postscript- | wish | java |
| nautilus-adapte | fortune | sox | realpath |
| gnome-terminal | nautilus-adapte | play | ls |
| rhn-applet-gui | xpdf | xscreensaver | wish |

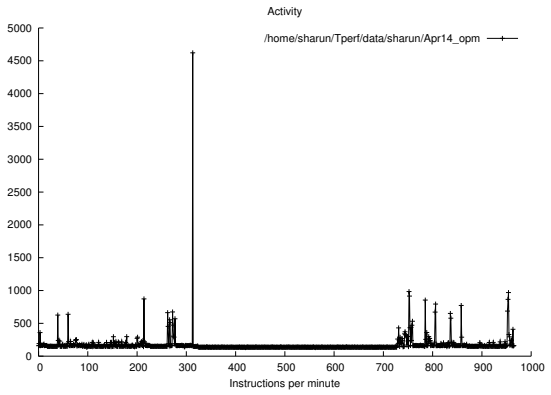
Table A.1: Top 10 Active Processes for some more users

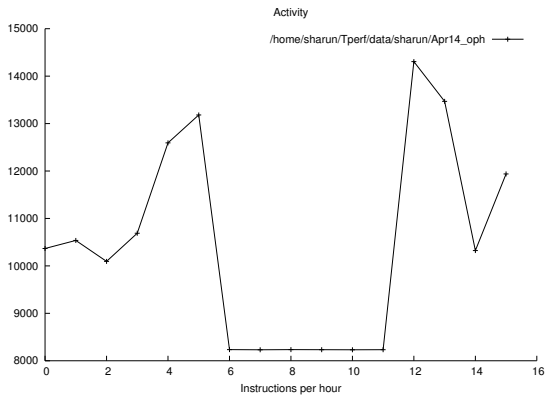

Size CDF
sharun Apr16
Percent of accesses
Size
0
2e+06
4e+06
6e+06
8e+06
1e+07
1.2e+07
1.4e+07
1.6e+07
1.8e+07
2e+07
40
45
50
55
60
65
70
75
80
85
90

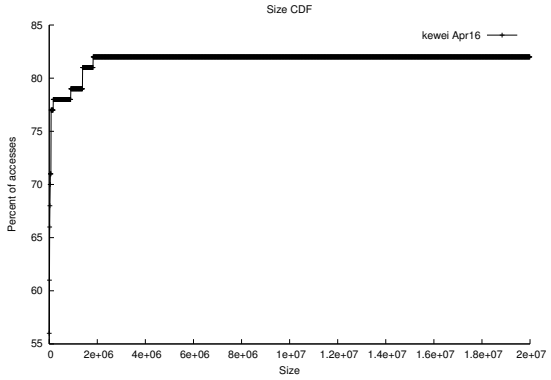

Access Count CDF
sharun_apr14
Time
0
5
10
15
20
25
0
10
20
30
40
50
60
70
80
90

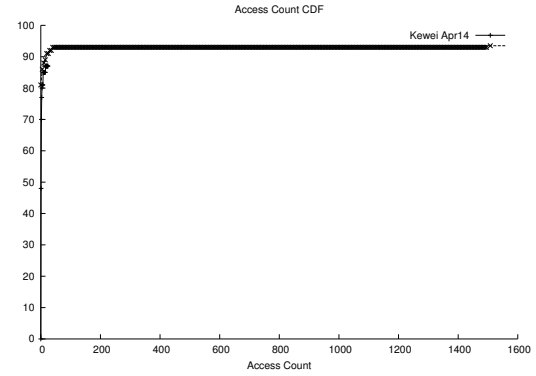

Time to Next Access CDF
Percent of Access
Time to Next Access
1e-06
0.0001
0.01
1
100
10000
0
10
20
30
40
50
60
70
80
90
100

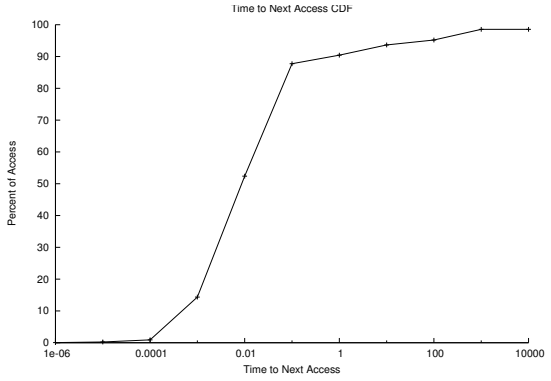

Open Time CDF
Open Time
1e-06
1e-05
0.0001
0.001
0.01
0.1
1
10
100
0
10
20
30
40
50
60
70
80
90
100

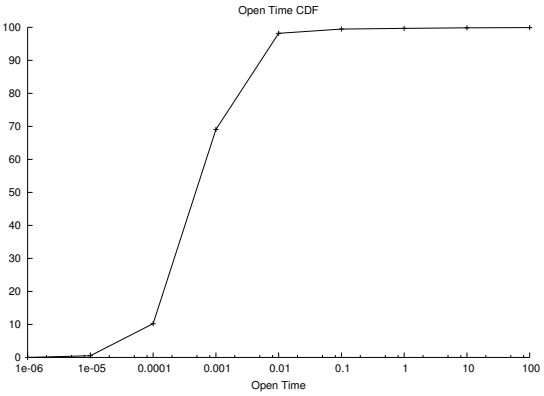

Activity
/home/sharun/Tperf/data/sharun/Apr14_opm
Instructions per minute
0
100
200
300
400
500
600
700
800
900
1000
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

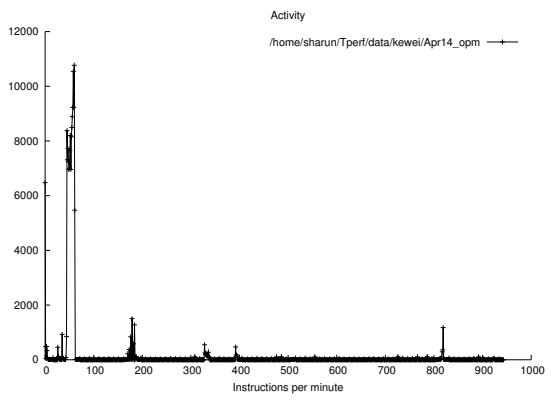

Activity
/home/sharun/Tperf/data/sharun/Apr14_oph
Instructions per hour
0
2
4
6
8
10
12
14
16
8000
9000
10000
11000
12000
13000
14000
15000

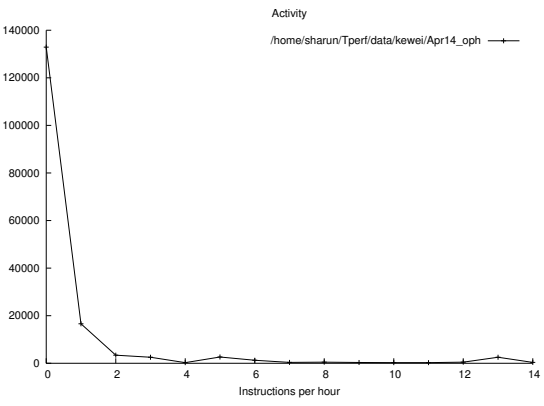

Size CDF
kewei Apr16
Percent of accesses
Size
85
80
75
70
65
60
55
0
2e+06
4e+06
6e+06
8e+06
1e+07
1.2e+07
1.4e+07
1.6e+07
1.8e+07
2e+07

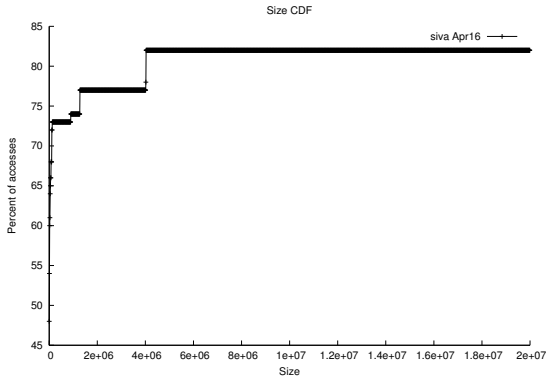

Access Count CDF
Kewei Apr14
Access Count
100
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000
1200
1400
1600

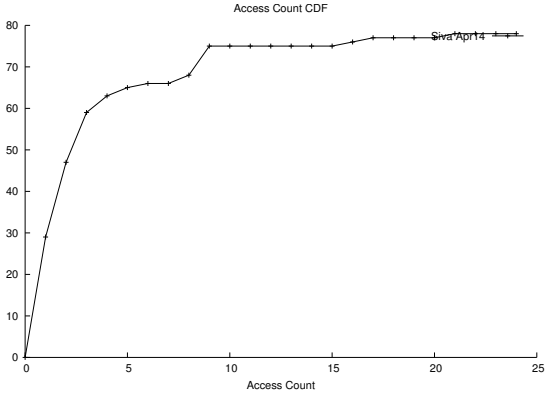

Time to Next Access CDF
Percent of Access
Time to Next Access
100
90
80
70
60
50
40
30
20
10
0
1e-06
0.0001
0.01
1
100
10000

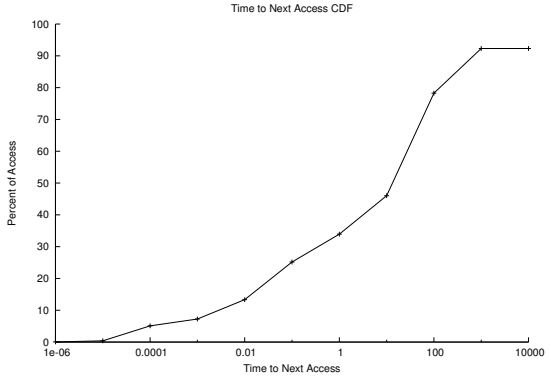

Open Time CDF
Open Time
100
90
80
70
60
50
40
30
20
10
0
1e-06
1e-05
0.0001
0.001
0.01
0.1
1
10
100

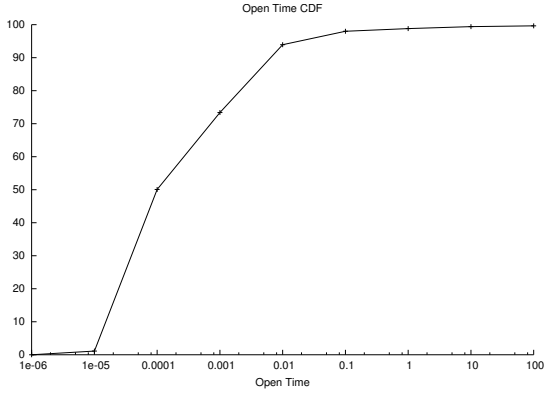

Activity
/home/sharun/Tperf/data/kewei/Apr14_opm
Instructions per minute
12000
10000
8000
6000
4000
2000
0
0
100
200
300
400
500
600
700
800
900
1000

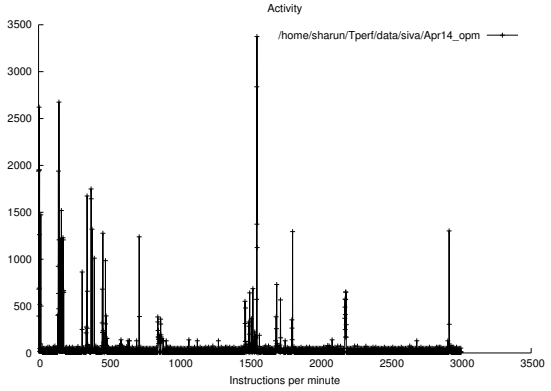

Activity
/home/sharun/Tperf/data/kewei/Apr14_oph
Instructions per hour
140000
120000
100000
80000
60000
40000
20000
0
0
2
4
6
8
10
12
14

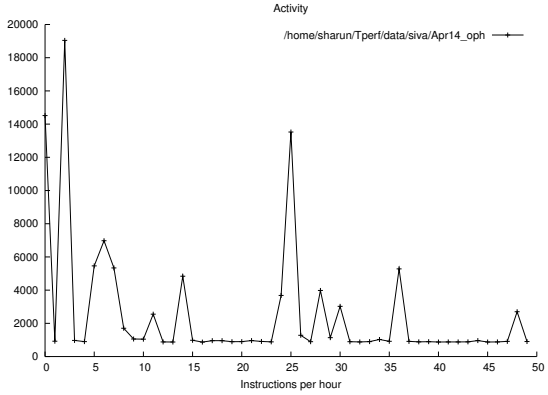

Size CDF
siva Apr16
Percent of acesses
Size
0
2e+06
4e+06
6e+06
8e+06
1e+07
1.2e+07
1.4e+07
1.6e+07
1.8e+07
2e+07
45
50
55
60
65
70
75
80
85


Access Count CDF
Siva Apr14
Access Count
0
5
10
15
20
25
0
10
20
30
40
50
60
70
80


Time to Next Access CDF
Percent of Access
Time to Next Access
1e-06
0.0001
0.01
1
100
10000
0
10
20
30
40
50
60
70
80
90
100


Open Time CDF
Open Time
1e-06
1e-05
0.0001
0.001
0.01
0.1
1
10
100
0
10
20
30
40
50
60
70
80
90
100


Activity
/home/sharun/Tperf/data/siva/Apr14_opm
Instructions per minute
0
500
1000
1500
2000
2500
3000
3500


Activity
/home/sharun/Tperf/data/siva/Apr14_oph
Instructions per hour
0
5
10
15
20
25
30
35
40
45
50
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000

# Appendix B

# Simulation results using MIST Traces

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 65.47 | 61.57 | 61.82 | 62.00 | ***65.92*** | 64.41 | 62.27 |
| 500K | 73.29 | 73.34 | 72.79 | 73.38 | ***73.75*** | 73.38 | 72.79 |
| 1Mb | 75.81 | 75.33 | 74.75 | 75.25 | ***76.35*** | 76.31 | 75.46 |
| 5Mb | 80.87 | 80.85 | 81.26 | 81.64 | ***82.30*** | 81.93 | 80.75 |
| 10Mb | 82.36 | 83.58 | 85.78 | ***91.48*** | 85.81 | 86.82 | 81.54 |
| 50Mb | 93.06 | 93.07 | 93.08 | 93.08 | 93.07 | 93.08 | 93.07 |

Table B.1: Hit Rates of trace Siva (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 1.26 | .84 | .84 | .85 | 1.24 | 1.12 | .87 |
| 500K | 2.73 | 2.75 | 2.55 | 2.61 | 2.81 | 2.63 | 2.53 |
| 1Mb | 4.57 | 4.09 | 3.61 | 3.72 | 4.70 | 4.65 | 4.14 |
| 5Mb | 12.13 | 12.29 | 13.55 | 13.37 | ***16.65*** | 14.89 | 11.64 |
| 10Mb | 17.55 | 21.64 | 30.96 | ***51.21*** | 29.37 | 33.98 | 14.14 |
| 50Mb | 89.03 | 89.03 | 89.03 | 89.01 | 89.01 | 89.03 | 89.01 |

Table B.2: Byte Hit Rates of trace Siva (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 256 | 330 | 234 | 48 | 43 | 54 | 206 |
| 500K | 279 | 393 | 263 | 28 | 27 | 42 | 237 |
| 1Mb | 299 | 344 | 258 | 22 | 23 | 30 | 221 |
| 5Mb | 323 | 537 | 362 | 10 | 11 | 19 | 276 |
| 10Mb | 303 | 371 | 336 | 7 | 7 | 10 | 415 |
| 50Mb | 45 | 221 | 11 | 1 | 1 | 1 | 64 |

Table B.3: Files replaced, trace Siva (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 80.56 | 80.54 | 80.11 | 82.81 | 81.78 | 74.77 | 80.53 |
| 500K | 86.82 | 86.82 | 86.85 | 86.37 | 87.52 | 86.05 | 86.77 |
| 1Mb | 88.13 | 87.80 | 88.10 | 88.77 | 88.68 | 87.76 | 88.12 |
| 5Mb | 95.70 | 95.66 | 95.71 | 95.75 | 95.77 | 95.53 | 95.73 |
| 10Mb | 95.79 | 95.62 | 95.79 | 95.86 | 95.83 | 95.74 | 95.82 |
| 50Mb | 97.32 | 97.31 | 97.31 | 97.33 | 97.33 | 97.33 | 97.32 |

Table B.4: Hit Rates of trace Sharun(Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 4.50 | 4.49 | 4.39 | 6.18 | 5.48 | 3.55 | 4.49 |
| 500K | 7.41 | 7.42 | 7.42 | 6.49 | 8.15 | 6.92 | 7.38 |
| 1Mb | 8.87 | 7.64 | 8.84 | 9.96 | 9.47 | 8.40 | 8.84 |
| 5Mb | 27.14 | 27.11 | 27.11 | 26.43 | 27.15 | 25.97 | 27.14 |
| 10Mb | 27.60 | 25.71 | 27.59 | 27.07 | 27.48 | 27.07 | 27.60 |
| 50Mb | 93.24 | 93.23 | 93.24 | 92.44 | 93.24 | 93.24 | 93.23 |

Table B.5: Byte Hit Rates of trace Sharun (Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 691 | 697 | 545 | 36 | 96 | 100 | 431 |
| 500K | 758 | 657 | 417 | 24 | 31 | 47 | 328 |
| 1Mb | 909 | 642 | 493 | 12 | 19 | 42 | 374 |
| 5Mb | 1109 | 745 | 504 | 17 | 32 | 28 | 395 |
| 10Mb | 1196 | 512 | 631 | 4 | 17 | 19 | 597 |
| 50Mb | 889 | 224 | 352 | 2 | 7 | 9 | 286 |

Table B.6: Files replaced, trace Sharun (Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 77.91 | 77.88 | 77.94 | 89.19 | 89.01 | 89.28 | 77.82 |
| 500K | 87.28 | 87.17 | 87.25 | 92.55 | 92.60 | 92.66 | 87.25 |
| 1Mb | 88.65 | 88.55 | 88.63 | 93.13 | 93.13 | 93.15 | 88.65 |
| 5Mb | 95.30 | 95.25 | 95.30 | 95.44 | 95.46 | 95.44 | 95.30 |
| 10Mb | 95.38 | 95.38 | 95.38 | 95.38 | 95.39 | 95.38 | 95.38 |
| 50Mb | 95.67 | 95.67 | 95.67 | 95.67 | 95.69 | 95.67 | 95.67 |

Table B.7: Hit Rates of trace Sharun (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 5.47 | 12.61 | 12.62 | 39.20 | 38.99 | 39.67 | 12.60 |
| 500K | 18.99 | 19.00 | 18.97 | 41.39 | 41.30 | 41.88 | 18.97 |
| 1Mb | 21.72 | 21.70 | 21.70 | 42.72 | 42.58 | 42.85 | 21.69 |
| 5Mb | 51.13 | 51.13 | 51.13 | 59.29 | 59.19 | 59.29 | 51.13 |
| 10Mb | 52.29 | 52.29 | 52.29 | 52.29 | 52.18 | 52.29 | 52.29 |
| 50Mb | 75.73 | 75.73 | 75.73 | 75.73 | 75.67 | 75.73 | 75.73 |

Table B.8: Byte Hit Rates of trace Sharun (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 109 | 134 | 101 | 6 | 6 | 5 | 126 |
| 500K | 37 | 207 | 40 | 2 | 2 | 3 | 28 |
| 1Mb | 96 | 158 | 69 | 2 | 2 | 2 | 29 |
| 5Mb | 33 | 153 | 90 | 1 | 1 | 1 | 22 |
| 10Mb | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 50Mb | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table B.9: Files replaced, trace Sharun (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 66.76 | 67.24 | 66.08 | 64.80 | 70.98 | 64.02 | 68.67 |
| 500K | 77.34 | 77.42 | 77.29 | 76.22 | 77.60 | 76.01 | 77.51 |
| 1Mb | 79.44 | 79.65 | 78.48 | 78.97 | 79.62 | 78.60 | 79.56 |
| 5Mb | 85.46 | 85.59 | 85.47 | 85.42 | 85.59 | 84.47 | 85.50 |
| 10Mb | 85.48 | 85.66 | 85.50 | 85.64 | 85.65 | 85.55 | 85.44 |
| 50Mb | 85.70 | 85.73 | 85.70 | 85.81 | 85.78 | 85.80 | 85.73 |

Table B.10: Hit Rates of trace Zhu (Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 4.23 | 4.43 | 4.04 | 3.69 | 6.46 | 3.21 | 5.25 |
| 500K | 12.81 | 12.80 | 12.46 | 10.15 | 12.78 | 10.03 | 12.87 |
| 1Mb | 20.53 | 20.79 | 15.82 | 16.78 | 20.22 | 16.42 | 20.56 |
| 5Mb | 66.96 | 67.14 | 66.92 | 62.89 | 67.09 | 54.95 | 66.85 |
| 10Mb | 69.20 | 69.03 | 67.47 | 66.84 | 69.19 | 65.63 | 69.04 |
| 50Mb | 70.20 | 70.17 | 70.22 | 70.22 | 70.17 | 70.18 | 70.05 |

Table B.11: Byte Hit Rates of trace Zhu (Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 1178 | 1400 | 1187 | 152 | 314 | 477 | 878 |
| 500K | 862 | 811 | 720 | 46 | 136 | 235 | 727 |
| 1Mb | 824 | 548 | 579 | 25 | 54 | 89 | 767 |
| 5Mb | 214 | 79 | 296 | 15 | 19 | 51 | 296 |
| 10Mb | 338 | 199 | 370 | 24 | 136 | 69 | 697 |
| 50Mb | 181 | 218 | 621 | 6 | 6 | 22 | 334 |

Table B.12: Files replaced, trace Zhu (Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 67.96 | 67.93 | 67.59 | 73.21 | 69.51 | 71.89 | 67.41 |
| 500K | 78.08 | 78.16 | 78.03 | 77.56 | 78.28 | 77.43 | 78.04 |
| 1Mb | 80.55 | 80.91 | 80.92 | 80.25 | 81.18 | 79.91 | 80.95 |
| 5Mb | 86.75 | 86.79 | 86.76 | 86.83 | 86.96 | 86.84 | 86.77 |
| 10Mb | 86.74 | 86.85 | 86.70 | 87.01 | 86.98 | 86.85 | 86.77 |
| 50Mb | 87.02 | 87.02 | 87.02 | 87.02 | 87.00 | 87.02 | 87.02 |

Table B.13: Hit Rates of trace Zhu (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 3.51 | 3.54 | 3.40 | 5.50 | 4.06 | 4.94 | 3.38 |
| 500K | 9.63 | 9.63 | 9.58 | 8.63 | 9.61 | 8.52 | 9.62 |
| 1Mb | 16.85 | 17.32 | 17.26 | 14.29 | 17.36 | 13.66 | 17.57 |
| 5Mb | 51.24 | 51.26 | 48.88 | 48.65 | 51.11 | 48.91 | 51.24 |
| 10Mb | 51.34 | 51.36 | 49.03 | 51.40 | 51.27 | 49.02 | 51.31 |
| 50Mb | 51.61 | 51.61 | 51.61 | 51.61 | 51.54 | 51.61 | 51.61 |

Table B.14: Byte Hit Rates of trace Zhu (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 501 | 463 | 435 | 30 | 71 | 65 | 333 |
| 500K | 465 | 316 | 319 | 19 | 60 | 38 | 321 |
| 1Mb | 500 | 353 | 299 | 14 | 34 | 26 | 317 |
| 5Mb | 293 | 273 | 117 | 4 | 18 | 7 | 219 |
| 10Mb | 350 | 217 | 152 | 2 | 6 | 5 | 206 |
| 50Mb | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table B.15: Files replaced, trace Zhu Apr14

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 2699 | 2891 | 2311 | 1351 | 1811 | 2005 | 2059 |
| 500K | 3795 | 4325 | 3904 | 1900 | 2274 | 3151 | 3822 |
| 1Mb | 6667 | 6489 | 6894 | 3095 | 2721 | 4402 | 6970 |
| 5Mb | 26405 | 31004 | 28001 | 15033 | 16178 | 20443 | 20592 |
| 10Mb | 38538 | 41034 | 37840 | 22058 | 27487 | 31165 | 32787 |
| 50Mb | 44690 | 54032 | 43429 | 46229 | *24703* | 41843 | 35314 |

Table B.16: Total bytes missed of trace Kewei (Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 4194 | 4366 | 4271 | 1979 | 3015 | 4621 | 4262 |
| 500K | 5766 | 5652 | 5962 | 2387 | 5095 | 7345 | 5791 |
| 1Mb | 8521 | 7219 | 8054 | 3045 | 5235 | 6074 | 6796 |
| 5Mb | 8652 | 11316 | 13405 | 8564 | 8768 | 26962 | 41564 |
| 10Mb | 47547 | 48502 | 52445 | 47475 | 36435 | 66646 | 71154 |
| 50Mb | 52387 | 48786 | 50764 | 28110 | *27623* | 44954 | 47419 |

Table B.17: Total bytes missed of trace Zhu (Apr16)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 1845 | 1617 | 1627 | 469 | 795 | 813 | 857 |
| 500K | 3212 | 2823 | 2742 | 1324 | 1988 | 1855 | 2045 |
| 1Mb | 5292 | 5476 | 4909 | 2587 | 3363 | 3590 | 2979 |
| 5Mb | 5593 | 8593 | 5711 | 4283 | 8146 | 5066 | 5085 |
| 10Mb | 9437 | 7468 | 6322 | 2787 | 5855 | 5735 | 6450 |
| 50Mb | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table B.18: Total bytes missed of trace Zhu (Apr14)

| Size | LRU | LFU | RR | GDS | INTER | INTRA | BOTH |
|---|---|---|---|---|---|---|---|
| 100K | 1154 | 1317 | 1338 | 789 | 790 | 876 | 826 |
| 500K | 3783 | 4502 | 3899 | 2345 | 2278 | 3171 | 2505 |
| 1Mb | 7895 | 6353 | 6518 | 2521 | 4026 | 4340 | 4003 |
| 5Mb | 22040 | 24381 | 19602 | 5538 | 8357 | 10840 | 32402 |
| 10Mb | 22494 | 18160 | 22245 | 7112 | 12053 | 11027 | 12172 |
| 50Mb | 11522 | 11522 | 11522 | 11522 | 11522 | 11522 | 11522 |

Table B.19: Total bytes missed of trace Siva (Apr14)

# Appendix C

# Simulation results using DFS Traces

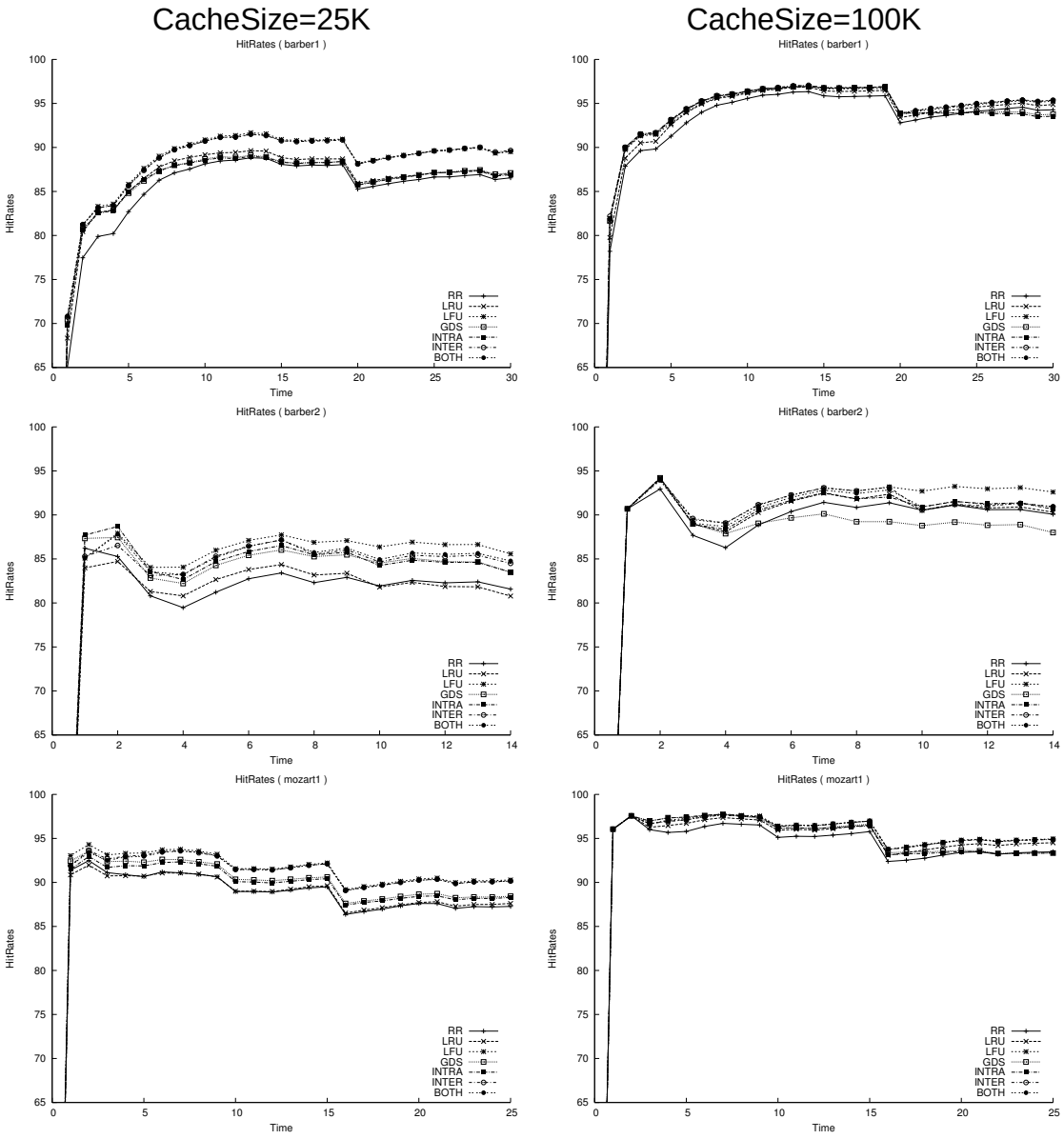



Figure C.1: Hit Rates recorded during the simulation using trace files barber1, barber2 and mozart1 with caches of size 25K and 100K resp.

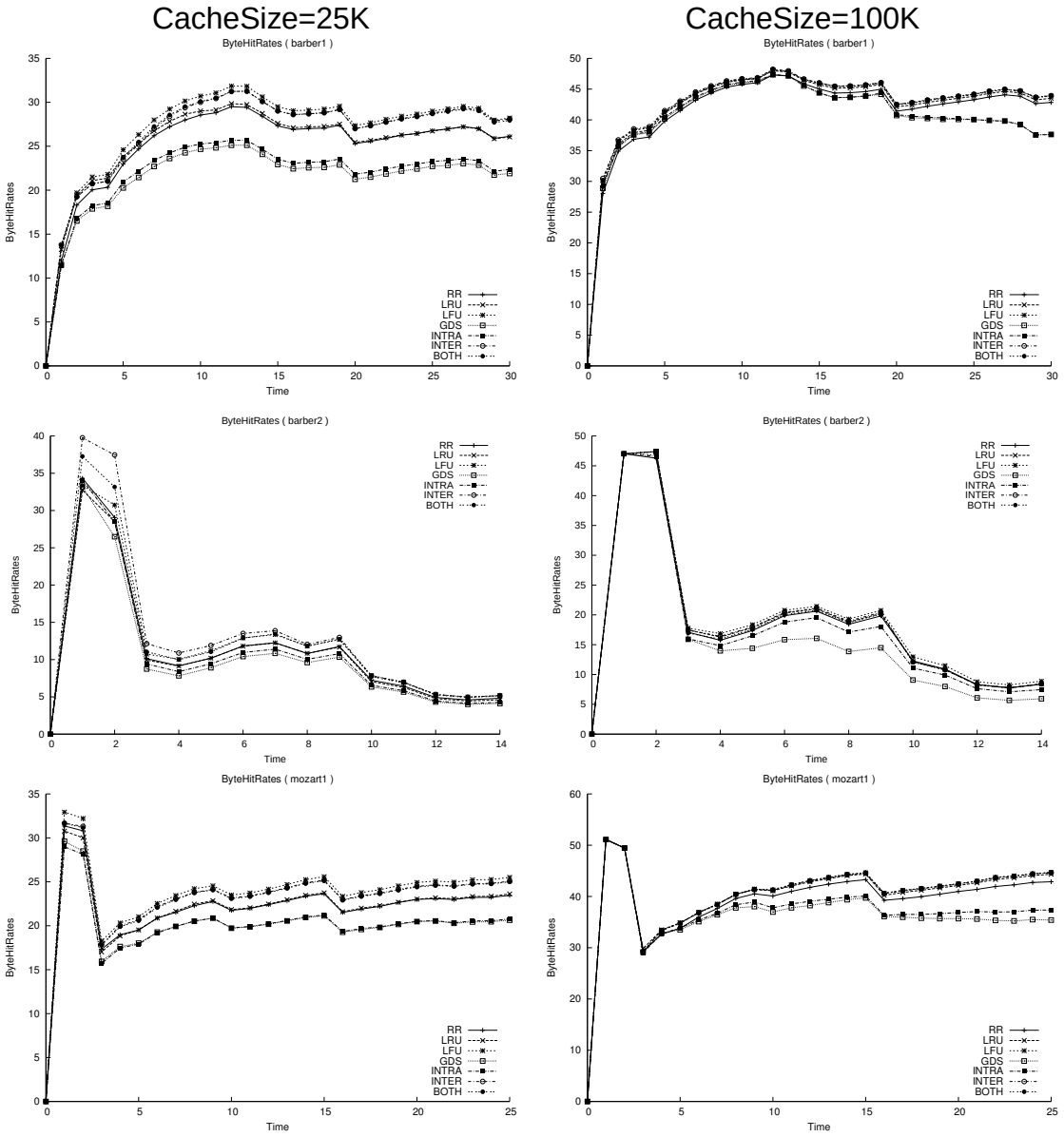



Figure C.2: Byte Hit Rates recorded during the simulation using trace files barber1, barber2 and mozart1 with caches of size 25K and 100K resp.

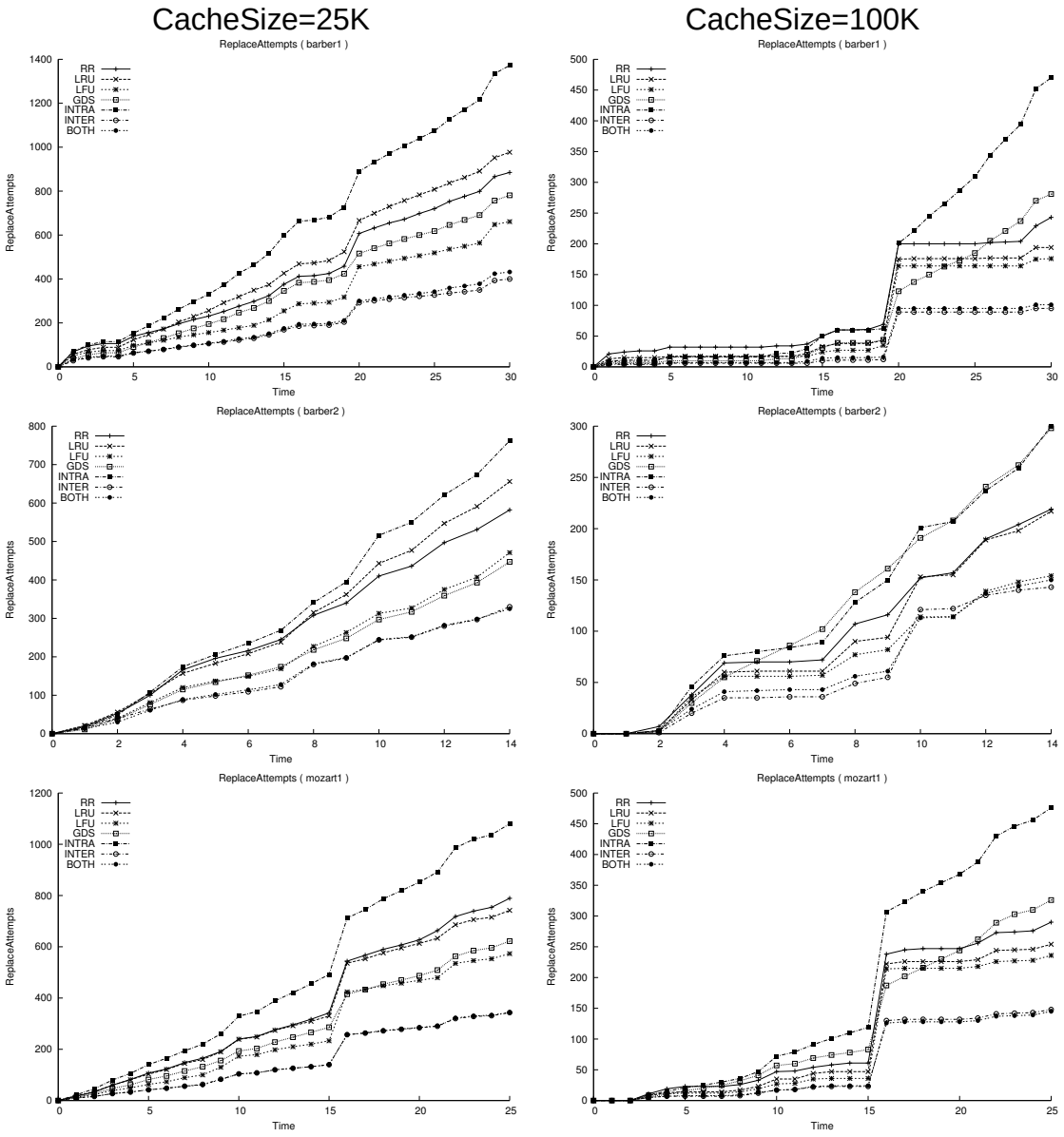



Figure C.3: Replace attempts recorded during the simulation using trace files barber1, barber2 and mozart1 with caches of size 25K and 100K resp.

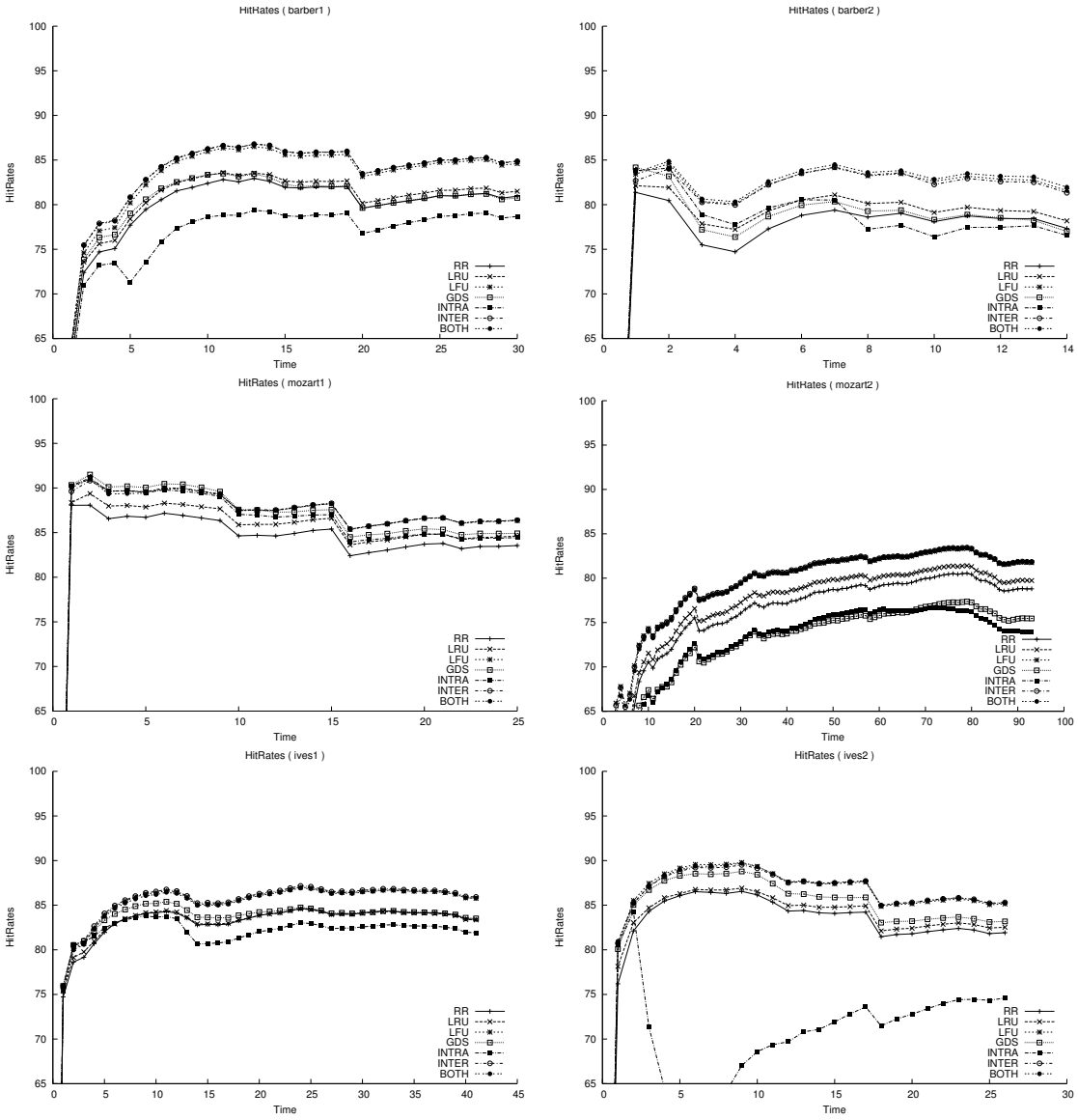



Figure C.4: Hit Rates recorded during the simulation using trace files barber1, barber2, mozart1, mozart2, ives1 and ives2 with caches of size 10K

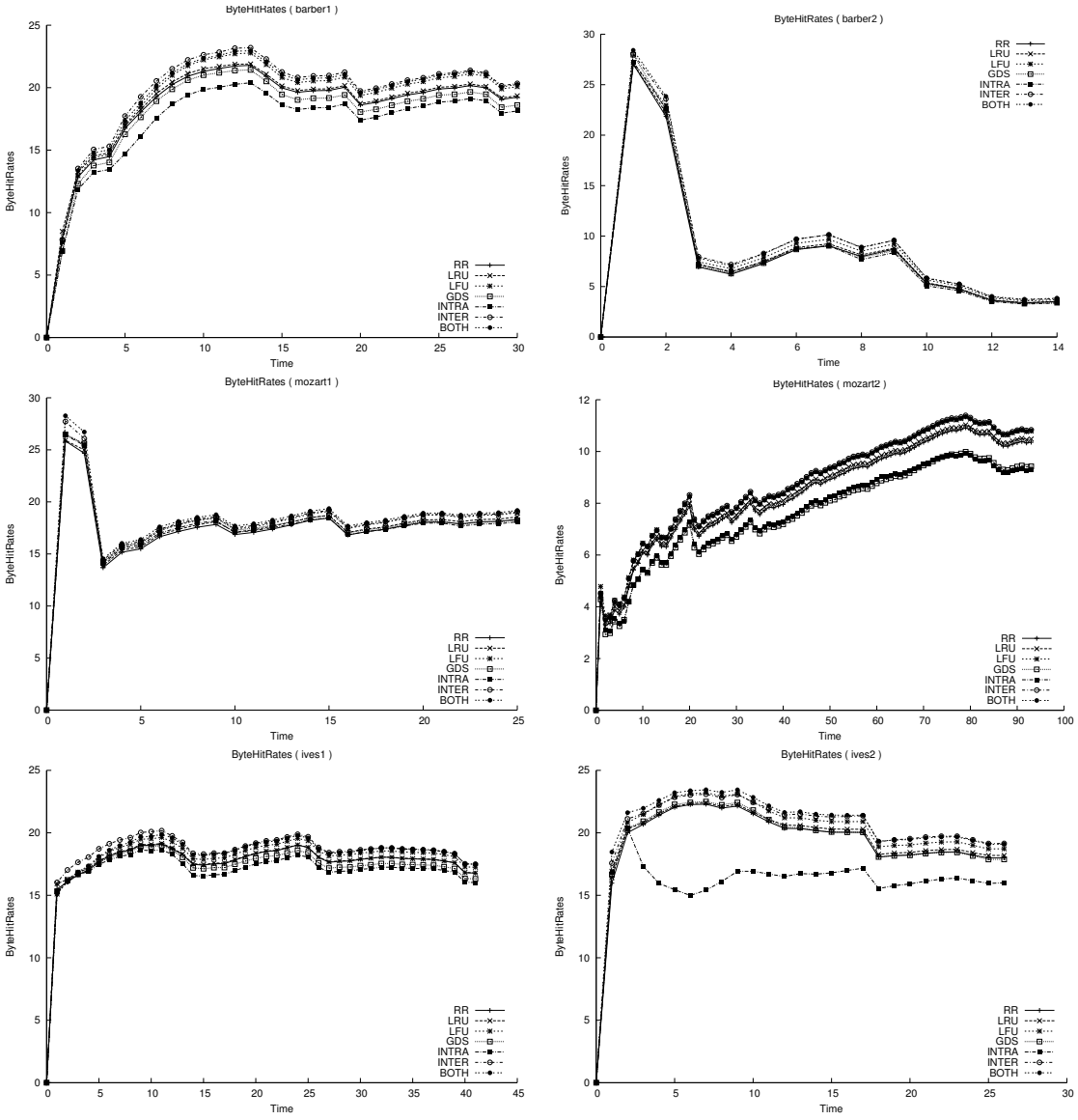



Figure C.5: Byte Hit Rates recorded during the simulation using trace files barber1, barber2, mozart1, mozart2, ives1 and ives2 with caches of size 10K

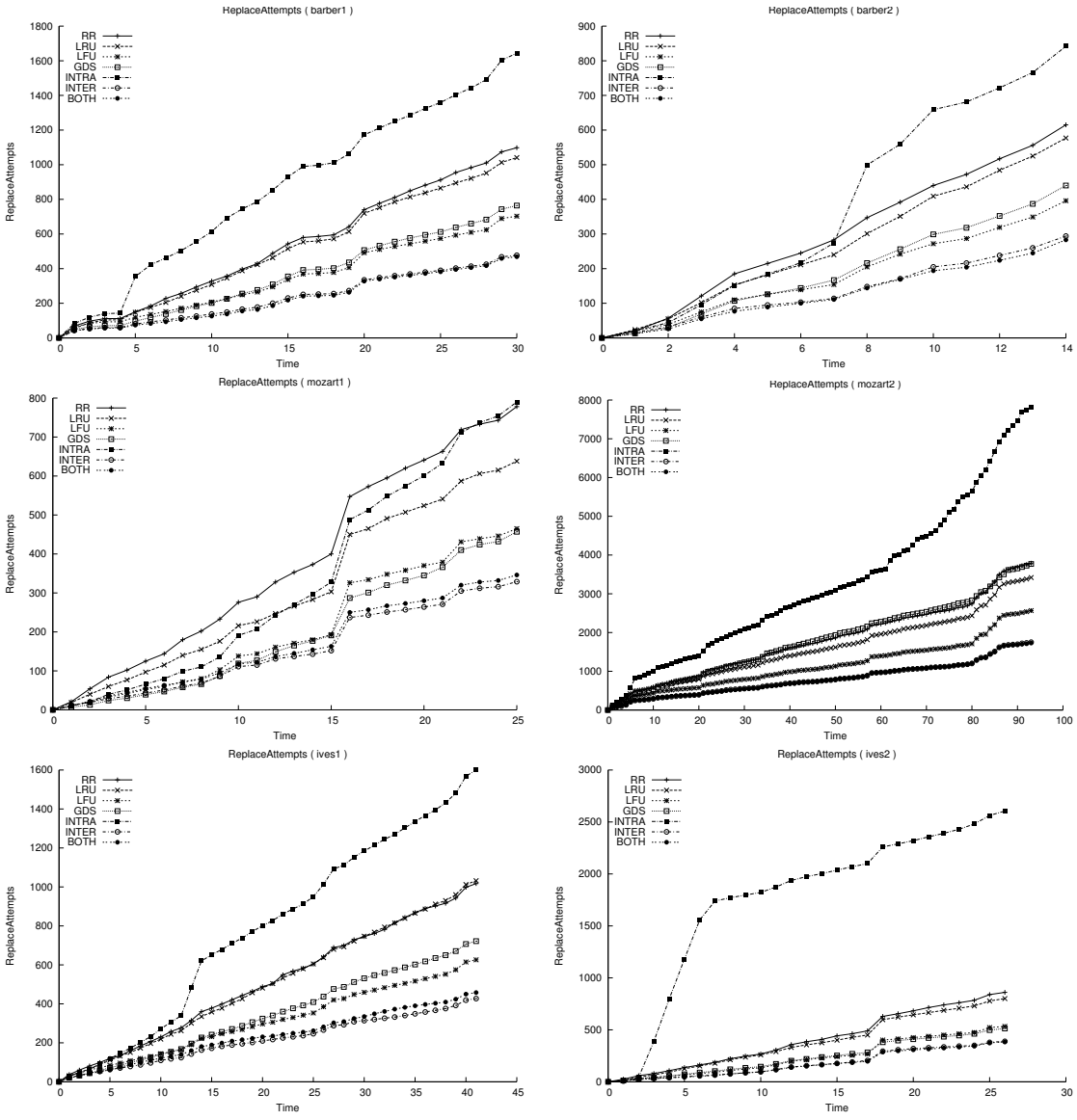



Figure C.6: Replace attempts recorded during the simulation using trace files barber1, barber2, mozart1, mozart2, ives1 and ives2 with caches of size 10K

# Bibliography

[1] B. Callaghan and P. Staubach. NFS Version 3 Protocol Specification, rfc 1813, June 2000.

[2] T. Anderson, M. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stocia. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP-18)*, October 2001.

[4] J. H. Howard, M. Kazar, S. Menees, d. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[5] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. of the 13nd ACM Symp. on Operating Systems Principles*, October 1991.

[6] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of the Fifth USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[7] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[8] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. nelson, and B. B. Welch. The sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[9] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating systems for 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[10] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[11] L. B. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software-Practice and Experience*, 26(6):705–736, 1996.

[12] W. Shi, H. Lufei, and S. Santhosh. Cegor: An adaptive, distributed file system for heterogeneous network environments. Technical Report MIST-TR-2004-003, Department of Computer Science, Wayne State University, January 2004.

[13] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, 1991.

[14] J. K. Ousterhout, Hervé Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, pages 15–24. ACM Press, 1985.

[15] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proceedings of the 1993 USENIX Winter Technical Conference*, pages 405–420, January 1993.

[16] D. A. Pease. Unix disk access patterns revisited. Technical report, Deptartment of Computer Science, University of Calafornia Santa Cruz, 1999.

[17] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

[18] M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, 2002.

[19] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems (USITS'97)*, December 1997.

[20] A. Amer and D. D. E. Long. Aggregating caches: A mechanism for implicit file prefetching. In *Proceedings of the ninth International Symposium on Model-*

*ing, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2001)*, pages 293–301, August 2001.

[21] A. Amer, D. D. E. Long, J. F. Paris, and R. C. Burns. File access prediction with adjustable accuracy. In *Proceedings of the International Performance Conference on Computers and Communication (IPCCC 2002)*, April 2002.

[22] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Measurement and Modeling of Computer Systems*, pages 188–197, 1995.

[23] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, 1997.

[24] M. R. Ebling, B. E. John, and M. Satyanarayanan. The importance of translucence in mobile computing systems. *ACM Trans. Comput.-Hum. Interact.*, 9(1):42–67, 2002.

[25] S. Shepler and B. Callaghan. NFS Version 4 Protocol Specification, rfc 3530, April 2003.

[26] T. W. Page, R. G. Guy, J. s. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software-Practice and Experience*, 28(2):123–133, February 1998.

[27] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network file-servers. In *Proc. of the 13th International Conference on Distributed Computing Systems*, 1996.

[28] B. Walker, G. Popek, C. Kline R. English, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.

[29] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams. Removal policies in network caches for world-wide web documents. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 293–305. ACM Press, 1996.

[30] A. Dix. Cooperation without reliable communication: Interfaces for mobile applications. *Distributed Systems Engineering*, 2(3):171–181, 1995.

[31] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, May 1996.

[32] A. Peacock. Dynamic detection of deterministic disk access patterns. Master's thesis, Department of Computer Science, Brigham Young University, 1994.

[33] S. Santhosh and W. Shi. Semantic-distance based cache replacement for wide-area file systems. Technical Report MIST-TR-2004-002, Department of Computer Science, Wayne State University, January 2004.

[34] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. In *Proc. of the Hot Topics in Operating Systems (HotOS) VII*, March 1999.

[35] H. Lufei and W. Shi. Performance evaluation of communication optimization techniques for distributed file systems. Technical Report MIST-TR-2003-006, Department of Computer Science, Wayne State University, July 2003.

## Abstract

FACTORING FILE ACCESS PATTERNS AND USER BEHAVIOR INTO CACHING DESIGN FOR DISTRIBUTED FILE SYSTEMS

by

SHARUN SANTHOSH

August 2004

Advisor: Dr. Weisong Shi

Major: Computer Science

Degree: Master of Science

The ability to stay continually connected, due to ubiquitous availabilty of a wide variety of networks, is a persent day reality. Yet the convenience and transparency of use, offered by present distributed file systems across heterogeneous networks have a long way to go. This thesis, examines present day user behavior and file access patterns for clues to imporve distributed file system design. Motivated by the results, a semantic based caching algorithms have been proposed. Not only does this approach produces better hit rates than conventional algorithms but it does so reducing the number of replacements by almost half. This gives a user more options while in a weakly connected or disconnected state and also results in savings on synchronization-related communication overhead, essential to the effectiveness of a heterogenous distributed system.

# Autobiographical Statement

EDUCATION

- Master of Science (Computer Science) Sept 2004
  Wayne State University, Detroit, MI, USA
- Bachelors of Engineering (Computer Science) May 2002
  University of Madras, Chennai, India.

PUBLICATIONS

- Cegor: An Adaptive Distributed File System for Heterogeneous Network Environments. W.Shi, S.Santhosh, H.Lufei.,10th IEEE Intl. Conference on Parallel and Distributed Systems 2004.
- Application-Aware Service Differentiation in PAWNs, H.Lufei, S.Sellamuthu, S.Santhosh, and W.Shi, International Conference on Parallel Processing 2004.
- A Semantic-Based Cache Replacement Algorithm for Distributed File Systems. S.Santhosh and W.Shi, Michigan Academy of Science Annual Meeting 2004
- HAP: A Home-based Authentication Protocol for Nomadic Computing, S.Santhosh and W.Shi, International Conference on Internet Computing 2003.

- Secure Application-Aware Service Differentiation in Public Area Wireless Networks, S.Santhosh, H.Lufei, S.Sellamuthu, and W.Shi, Technical Report MIST-TR-03-010, Nov. 2003, submitted for journal publication