FILE CACHE MANAGEMENT FOR MOBILE COMPUTING

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

For the Degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Kevin W. Froese

August 1997

Permission To Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

Abstract

The emerging demand for mobile computing has created a need for improved file system support for mobile clients. File access is a fundamental requirement of any computing environment, and mobile users expect their file service needs to be met just like any other user. Current file systems with support for mobility provide availability through file replicas cached at the client. These systems manage client file caches in the same manner as file caches in traditional LAN-based distributed systems. However, mobile clients may experience a wide range of network connections which vary greatly in terms of bandwidth, error rates, and expense of use, and these diverse environments add to the complexity of the cache management task. The impact of mobility on cache management has yet to be thoroughly investigated.

This thesis presents the results of a study of techniques for managing a mobile client's file cache. Trace-driven simulation is used to investigate performance issues relating to three aspects of mobile client file cache management – hoarding, write backs, and caching unit. Special attention is paid to the development of metrics for evaluating file cache performance in a mobile environment. Results show that through the use of well-designed cache management strategies and careful network management, it is possible to provide acceptable file system performance in a mobile environment, even with a low-bandwidth connection.

Acknowledgements

This thesis would not have been possible without the support of many people. I would like to thank:

Rick Bunt, my supervisor, for his guidance and encouragement, and for helping me keep this work in perspective when it seemed overwhelming. Thanks also to my thesis Advisory Committee members, Derek Eager and Jim Greer, and to Janelle Harms, the External Examiner, for their valuable comments and suggestions.

Lily Mummert (Carnegie Mellon University) for providing me with valuable file reference traces.

The Natural Sciences and Engineering Research Council of Canada (NSERC), Telecommunications Research Laboratories (TRLabs), and IBM Canada for funding my post-graduate education.

Sonya, for being my biggest supporter and best friend.

All the people who made work more fun. Lee Kennedy, Brad Mazurek, Martin Arlitt, Leanne Breker, and Fabian Searwar for making lunches (and goofing off) such a good time. Greg Oster, for countless hours spent fixing my bugs, and for teaching me most of what I know about C and Unix. John Chayka for making not working more fun.

Finally, my parents, for always being there when I needed them.

Table Of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The continual improvement in the capabilities and portability of notebook computers and hand-held personal digital assistants (PDA's) has given rise to a new style of computer use: that of *mobile computing*. Whereas traditional "stationary" computing consists of users performing work at fixed-location workstations with abundant access to "home resources" such as a file system, mobile computing sees the computer accompanying the user, for use in many different locations.

A mobility-aware network file system is a critical component of any mobile computing environment [9]. To be effective, the file system must provide mobile users with functionality comparable to that of a LAN environment. Users require not only continued read access while weakly connected, but also the ability to update files and have these updates be reintegrated with the home file system in a timely manner. Accomplishing this requires a cache manager able to successfully hoard files, replace files within the cache, maintain a log of mutating file operations, reintegrate modified files when possible, and arbitrate network access between reads and writes – all across a wide range of network connections.

Mobile computing allows for *location-independent* work. This method of operation is typically characterised by three actions on the part of the user: disconnection from a network, movement to a new location (along with the mobile client), followed by reconnection to the network and renewed access to data. (It is also possible for a user to resume working at a new location without first establishing a network connection – this is known as "disconnected operation" [6], and can be thought of as a special case of the more general problem.) The goal of system support for location-independent computing is to attempt to make the changes in the computing environment transparent to the user (or at least to the application program), thus allowing the user to work in the same manner and with the same productivity from any location.

Location-independent computing is possible without support from networks, file systems, or even mobile clients. In the simplest case, a user can simply copy and carry any files they choose with them to a new working location. It may even be possible that the user does not have to manually transport files if they are remotely accessible from the new location (say via telnet or FTP). Recent developments such as Mobile IP [39] make this form of work easier.

There are several motivations for providing more explicit file system support for location-independent computing. First, there may not always be a network connection available at the new working location, and this means it is the user's responsibility to transport files. Additionally, the user is also responsible for updating copies of files at various locations and ensuring that needed files are present [27]. This provides ample opportunity for mistakes to be made, and for files to become lost or forgotten. Perhaps most importantly, the efforts required by the user to operate effectively in a location-independent manner are inconvenient. Too much time is spent performing activities which are not directly related to the work a user wishes to perform.

Given that it is desirable to provide support at the system level for location-independent computing, it is necessary for the file system to ensure the availability of a user's files at any location. This is done by employing automatic file replication management techniques [21]. Central to these techniques is the caching of files at a mobile client. There are three phases to this form of client file caching: the hoarding of files to prepare for mobility, the servicing of file system requests while mobile, and reintegration of the file system when the user returns to a LAN environment.

It has been noted that:

> Caching issues are beginning to predominate the open research topics in this area [mobile computing]. In between connected and disconnected states, there are many states of expensive, intermittent and unreliable connections. Adapting caching to these varying situations is a necessity [2].

Since the mobile environment differs significantly from stationary environments (in terms of available bandwidth, work performed by users, etc.), it is unclear what type of file cache management techniques are needed to provide acceptable performance for mobile clients. While the file system performance of a mobile client in a *weakly-connected* environment (where only a low-bandwidth network connection is possible) need not match that in a *strongly-connected* (typically a LAN) environment (since it assumed that users are willing to sacrifice some performance in exchange for mobility), every effort should be made to keep the performance impact of mobility to a minimum.

Previous work in this area has focussed on providing some level of functionality to the users of mobile clients in location-independent environments. The performance of the cache management techniques used to date has not been thoroughly investigated, nor has the performance of alternative cache management policies. This work attempts to address this lack of performance information through an investigation of several techniques for managing a mobile client's file cache.

The remainder of this thesis is organised as follows. Chapter 2 provides background information on mobile computing and cache management issues, and provides an overview of existing file systems which support mobility. Chapter 3 discusses factors affecting file cache performance and presents the metrics used in this study to evaluate cache management policies in mobile environments. The experimental workload and simulation model used are described in Chapter 4. The design and results of the performance study undertaken are presented in Chapters 5, 6, and 7. Finally, Chapter 8 summarises the work done, presents the contributions of this thesis, and discusses future research opportunities.

Chapter 2

Background

This chapter describes the issues related to the management of a mobile client's file cache, and discusses the importance of developing efficient file cache management techniques for mobile clients. The definition of mobile computing used in this work is presented, as is a description of existing mobile file systems. Finally, issues arising in file cache management in both stationary and mobile environments are discussed.

2.1 Motivation

A significant amount of work has been done in developing new file system strategies to support mobile computing. Systems such as AFS [16, 17, 19], Ficus [14], and Coda [21] have all been adapted to provide file access for mobile clients. All of these systems rely on a file cache at the client for providing file availability. The performance of this cache has a major impact on the ability of a user to perform useful work. For example, if a client attempts to access a file which is not present in its cache, the user may be stalled until the file can be retrieved – which could take longer than a user is willing to wait (depending on the nature of the connection). While each file system has developed policies for managing certain aspects of the file cache (such as write backs in AFS [17, 19] and Coda [10]), cache management policies have not been addressed in detail.

This research examines the performance implications of various approaches to managing a mobile client's file cache. Two facets of cache management, the choices of hoarding and write-back policies, are investigated in detail. The suitability of block-based caching in a weakly-connected environment is also examined.

2.2 Mobile Computing

"Mobile computing" is a broad term with many possible interpretations [22]. Definitions range from "computing while moving" (in either a wide or a local area) [11, 20, 30, 31, 36, 40, 50] and "location-independent computing" [10, 14, 16, 43] to "ubiquitous computing" [53]. For this research, mobile computing is defined to be a style of computing where an individual user may perform work on a client machine in several different locations (at different times), each of which may have a significantly different type and quality of connection to the "home" file server. While the user does not move *during* a session, he/she often changes location *between* sessions. Ideally, the user should experience the same working environment regardless of his/her location [3, 42]. That is, the user seeks uniformity in:

- functionality – defined by the work which can be performed at a given location
- performance – defined by how effectively work can be performed at a location
- interface – defined by the actions a user must take to perform a task at a location

To date, existing mobile file systems have focussed on providing functionality. This work focuses on performance. Interface issues are only beginning to be examined by the mobile computing community.

It is important to note that while mobility has the potential to be extremely beneficial for the user, it creates many difficulties for the operating system in general, and the file system in particular [2]. Mobile clients are resource-poor compared to stationary workstations, and will likely continue to be so in the foreseeable future. The resources of a mobile client are much more likely to fluctuate over time than are those of a stationary client. Users will likely perform different tasks on a mobile client than on a stationary workstation. Finally, security becomes a much larger problem in the mobile environment, both in terms of risks to mobile clients themselves, and to their data (especially when wireless communications are used). These constraints must be kept in mind at all times when designing mechanisms to support mobility.

2.2.1 Environment Definition

A style of computing based upon current LAN-based client-server distributed systems is assumed for this work. Each user has a "home" location, which is the LAN where the home file server resides. While there may be several file servers at the home location, for this work such a configuration will be thought of as a single logical file server. Apart from the home location (say, the workplace), users may work from several other locations from time to time (such as the user's place of residence or a hotel room). Note that the "home" location is not necessarily the location where a user performs most of his/her work. Rather, it should be thought of as the (physical) permanent home for a user's files.

A key assumption about the environment is that while a user may experience a wide range in available bandwidth between various working locations, the bandwidth at a given location is fairly constant. For example, a user may have access to a 10 Mb/s Ethernet at the office, but only a 28.8 kb/s modem when working from home or on a business trip, and at times may have no network connection at all.

The type of work performed is assumed to be that found in a typical computer science academic/research environment: such things as text editing, compiling programs from source code, document preparation and viewing, and information browsing (such as using a World Wide Web client). This type of workload is the basis for this study despite the arguments that this is *not* the type of work that users will likely perform in a mobile environment [27]. While this argument has merit, the fact that it is impossible to accurately predict what future workloads from mobile environments will look like makes the use of current workloads a reasonable choice. It is felt that stronger results can be obtained by using a workload model based on current data rather than a speculative model of future user behaviour.

Finally, when mobile clients are discussed, they are assumed to be *at least* as capable as current notebook computers (in terms of CPU speed, memory size, and storage capacity).

2.2.2 Types of Connectivity

As mentioned above, a mobile client may experience a wide range of network connections over time. A client's connection with a file server can be placed in one of three broad categories:

- *strongly connected* – the client is connected to a file server over a fast and reliable link, such as an Ethernet

- *weakly connected* – the client is connected to a file server, but only over a slow and possibly error-prone or expensive link [32], such as a modem connection (either wired or cellular), or a radio link

- *disconnected* – the client has no network connection to a file server [21]

For this work, it is assumed that a mobile client will experience each of these types of connection over some period of time. However, only one type of connection will be experienced during any given session. It should be noted that while a weakly-connected client may temporarily become disconnected due to a network failure, the client is still considered to be operating in a weakly-connected environment. A truly disconnected client has no network connection to a file server at all during a session.

2.3 File Systems for Mobile Computing

Traditionally, file system performance is measured in terms of factors such as latency (how quickly the file system responds to client requests), scalability (how many clients the file system can serve), and availability (what measures are taken to ensure that clients can access files at all times). These factors are important for both mobile and LAN-based file systems. However, in the mobile environment the issue of availability becomes predominant. Since it is unlikely that a client will have a strong connection to a file server when away from its home location, the network can not be relied on exclusively as a means to make files available to a client. Thus, the approach taken by most systems is to cache files locally at the client. While this ensures that the files in the client's cache are available for use, it also creates several problems. A detailed description of file caching in both disconnected and weakly-connected environments follows.

2.3.1 Disconnected Operation

File system support for disconnected operation centres around the idea of maintaining a file cache at the client, so that when disconnection occurs the client can continue to operate on these files. (It is important to note that whole-file caching, as opposed to file block caching, is used when supporting disconnected operation.) For this approach to be useful, the cache must contain (prior to disconnection) the files which a user will want to work on in the near future. This is accomplished through the caching of recently used files, by attempting to keep a user-defined set of files in the cache [46], and/or through long-term predictive caching techniques [10, 23]. While operating in disconnected mode, a client can perform any operation on these files that could be performed while connected. However, if a client tries to access any file not in its cache (i.e. a cache miss), an error occurs. The user either must choose to work on something else, or must wait until reconnected to a file server in order to access the file.

While disconnected, all file system operations which a client performs are recorded in a *log file*. Upon reconnection with the file server, the log file is replayed, which has the effect of propagating to the file server all file system changes that were made. In this way, the final state of the file system is (ideally) the same as if the changes were made while connected. Log files are usually optimised in order to save disk space at the client and to decrease the time needed for reintegration [18, 46].

This optimisation is accomplished through recognition of which operations actually affect the final state of the file system. Operations that do not change the state of the file system (such as reads) are called *non-mutating*, and are not recorded in the log file. *Mutating* operations are logged, but only if no subsequent mutating action "cancels" the operation. For example, if a file is created and then removed, the final state of the file system has not changed, and therefore neither the creation nor the removal needs to be logged.

A file system's *replication strategy* determines what type of operations are allowed on replicas of files. A *pessimistic* strategy allows only one copy of a file to be updated at a time, and ensures that all copies are then modified to reflect these changes. An *optimistic* strategy allows replicas to be modified independently, but still attempts to detect and resolve conflicts when they occur [1]. An optimistic replication strategy must be used by a file system that allows disconnected clients to modify files. Such a strategy allows for the possibility that a cached copy of a file may be modified at the same time that a copy residing on a file server is changed. This is known as a *write-write conflict*. Studies have shown that such conflicts occur only rarely [21, 24, 46], and that optimistic replication is a good strategy for a disconnected environment. However, some write-write conflicts do occur, and these must be resolved. This is an issue which has been studied extensively [8, 16, 26, 29, 33, 40, 41, 43, 49, 50, 51], with current work focusing on the development of *application-specific resolvers* [25].

2.3.2 Weakly-Connected Operation

Mobile clients differ from desktop workstations in two main ways. First, mobile clients tend to be resource-poor compared to desktop workstations. Processing power, memory and disk capacity, and network bandwidth of mobile clients may all be several orders of magnitude less than those of a stationary client [30]. As well, mobile clients may often possess only a weak connection to a file server, or possibly no connection at all. Common types of weak connection include modem links (either wired or cellular, with limited bandwidths, probably no greater than

57.6 kb/second), radio links (with bandwidths of up to 19.2 kb/second), and wireless local area networks (with bandwidths of up to several Mb/second). The key feature of these types of connections for this research is that they offer significantly lower bandwidth than a wired LAN. Thus, it would not be practical to support the file system demands of weakly-connected clients in the same manner as fully-connected clients. While the techniques used for supporting disconnected operation form a useful base towards supporting weakly-connected operation, there are significant differences between the two environments which must be addressed by a file system.

Figure 2.1 shows how reads and writes are handled in a weakly-connected environment. Whenever the client attempts to read from a file, the cache is searched for the file. If the file is present in the cache no network access is needed. If the file is not in the cache, then the read request is placed in the read queue where it waits for access to the network. When the network becomes available, the request is sent to the file server over the weak connection, and the file is then transfered to the client. When a client issues a write, the operation is saved in the log file, and is placed in the write queue. The write may later be sent to the file server over the weak connection for file system reintegration. All network traffic passes through a network interface which arbitrates access to the client's network connection.

The most important advantage of having a weak connection (as opposed to being disconnected) is that cache misses are no longer critical. If a client needs to access a file which it does not have in its cache, that file can be retrieved from the file server, even if the latency is rather large. In this respect, a weakly-connected client operates much like it is fully-connected. However, the more files a client needs to fetch from the file server, the worse the perceived performance will be for the user.

Another aspect of weakly-connected operation is that file system mutating operations can be written back to the file server. There are both advantages and disadvantages to performing write backs, and there are many issues involved in determining the performance impact of using write backs. The use of write backs in weakly-connected environments has not been extensively studied in the literature. The issues surrounding the implementation of a write-back mechanism are discussed

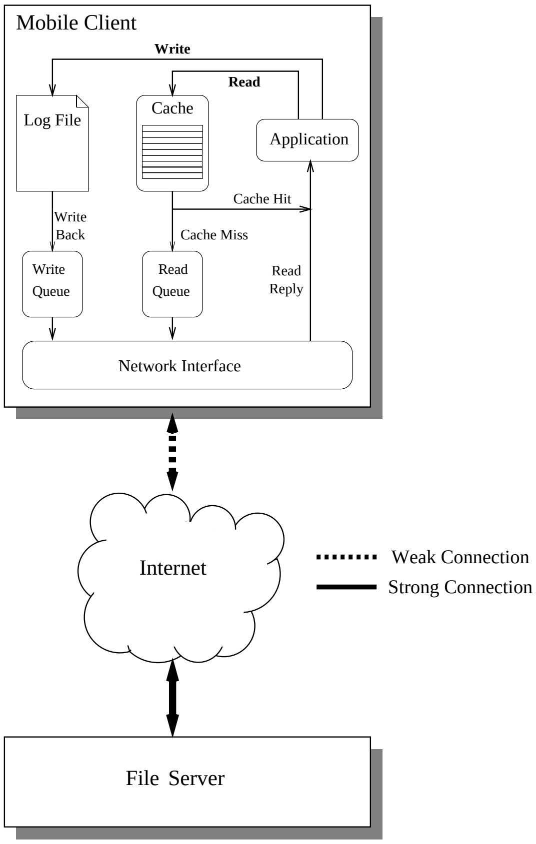

Figure 2.1: Reads and writes in a weakly-connected environment

in Section 2.4.2.

An unexplored situation created by the presence of a weak connection is that whole-file caching is not mandatory. While current systems do take this approach, caching individual disk blocks is an alternative whose performance implications have yet to be investigated (see Section 7.2).

2.3.3 Bandwidth Conservation

The overriding principle which must be followed when providing file system support for weakly-connected operation is that of bandwidth conservation, since this is likely the most precious resource at a mobile client [10]. It may often be necessary to use more of other resources, such as disk space and CPU time, in order to keep bandwidth requirements to a minimum [17]. The servicing of cache misses should be given the highest priority on bandwidth, since a user may be effectively blocked from working until a file is received from the file server. All other functions requiring bandwidth (such as write backs) should be deferred until there are no cache misses to be serviced.

2.4 File Cache Management Issues

The management of a client's file cache in a distributed system is a fairly well understood process. However, 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. As well, 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 behaviour is strongly affected by perceived performance, it might be expected that the workload (i.e. file reference patterns) of a weakly-connected or disconnected user will be different from that of a strongly-connected user. Since any good cache management policy relies on a knowledge of the patterns of requests which occur, it may be advantageous to reflect this change in file request traffic in cache management policies. A discussion of client file cache management policies in a mobile environment follows.

2.4.1 Replacement Policy

In existing distributed file system environments, client file caches typically utilise a Least Recently Used (LRU) replacement policy. That is, when a block in the cache needs to be removed to make room for an incoming block, the block which was referenced the furthest in the past is chosen for replacement. This policy is known to perform well in this type of environment [1, 54]. Most current research into mobile file systems has assumed that the LRU replacement policy will continue to be suitable for a mobile client. While there are exceptions to this prevailing opinion, such as [16], even these researchers have not yet investigated alternatives. Since the pattern of requests seen by the file cache may well be different from that seen in current distributed systems, it seems reasonable to investigate whether LRU continues to be the best choice for a replacement policy.

For performance reasons it may be necessary for new replacement policies (or variations on existing policies) to be investigated. An example of this is the possibility of adding the capability to "pin" certain files in the cache to the replacement policy [16]. That is, certain files may be flagged as being ineligible for replacement, ensuring that they will always be in the client's cache. This may be necessary because a mobile client may not be able to be regain possession of a critical file once it has been removed from its cache, owing to the uncertain nature of the client's network connectivity.

2.4.2 Write-Back Policy

In the context of weakly-connected operation, a write back is the propagation of file system mutating operations back to a file server. In this way, the results of a write at a client are reflected in the file system rather than just stored in a log file. A cache's write-back policy determines when and how write backs occur. There are performance advantages to delaying write backs [1, 5, 12]. For example, multiple mutating operations can be accommodated in a single write back. Examples of write-back policies include *write back on close* and *write back on reconnect*. Write backs can be initiated either by the client [10, 17, 19], or by the server [48]. In a weakly-connected environment, write backs may be performed as soon as there is available bandwidth, or they may be delayed for some amount of time [12].

There are several reasons why performing write backs is desirable, even in a bandwidth-limited environment. First, once a write has been sent to the file server, it no longer needs to be stored in the client's log file. This has the advantage of keeping the log file small, which can be important if disk space is limited at the client (which it usually is). As well, if writes are propagated to the file server while weakly connected, there are fewer events that need to be replayed upon reintegration with the network. Thus, the time needed for reintegration will be shorter. Finally, the quicker the file server is able to commit to disk the changes a weakly-connected client makes to files, the less chance there is of a write-write conflict occurring in that file. Since resolving such conflicts can be difficult, reducing the probability of them occurring is beneficial.

2.4.3 Prefetching

In a traditional cache management scheme, the first reference to a file will always result in a miss, since the file can be in the cache only if it was previously referenced (fetch on demand). In a traditional distributed file system setting, this is of little concern. However, in a mobile environment this can have a large impact on performance. For example, should the user of a weakly-connected client wish to run an application which reads from many separate data files, none of which are cached, each file would have to be sent to the client from the file server as it was needed by the application. If the data files were large enough that the cache could not hold them all, each invocation of the application would take a significant amount of time. However, if the file system is able to estimate future references (based, perhaps, on past references), it could begin to send those files to the client *before* they are requested. In this way, the high latency of file accesses could be hidden. The caching of files before they are requested is called *prefetching*. Prefetching is appealing in a weakly-connected environment, not because there is insufficient bandwidth to meet demand, but because there is insufficient bandwidth *at any instant*.

Prefetching algorithms are significantly more complex than other cache management algorithms since they attempt to predict future requests before they occur. Previous work [10, 23, 28] has relied on keeping extended histories of a user's file access patterns, and then recognising these patterns when they occur again. While some success has been achieved with these algorithms, it may also be possible to extend an algorithm so that it is able to successfully predict file accesses even when requests do not fit into a previously observed pattern.

The ability of prefetching to hide the latency of a slow network connection makes it an extremely attractive avenue of performance research. However, the ability of any prefetching mechanism to successfully anticipate future needs is limited, and this creates several problems. Not only is there the potential for bandwidth to be wasted due to poor prefetching decisions (which occurs any time a prefetched file is replaced in the cache before it is referenced), but there is also the problem of how prefetched files should be accommodated by the replacement policy [4]. If prefetched files are managed in the same way as demand-fetched files, the problem of choosing which resident file to replace for a prefetched file arises. If prefetched files are treated specially, the cache's replacement policy must become much more complex. While these issues have prevented prefetching from gaining much success in traditional distributed environments, prefetching may be found to be more successful in mobile environments.

2.4.4 Hoarding

Hoarding is somewhat similar in concept to prefetching, but it is concerned with the loading of the file cache while a user is strongly connected, in anticipation of future disconnection from the network. The term originated with the Coda file system [46]. In Coda, the file system keeps track of a user's reference history, and uses this along with a user-provided hint file (called a *hoard database*) to load a client's cache with files that the user will (hopefully) want to use while disconnected. Each file has some priority associated with it (provided by either the user or the system), and these values are used to help determine which files should be cached if not all can be accommodated. Thus, the goal of hoarding is to provide file availability over a longer period than prefetching is concerned with. For this research, the term hoarding will be used to refer not only to the specific techniques used in Coda, but more generally to any method of "heating" a mobile client's file cache while strongly connected, possibly (but not necessarily) with the help of user advice.

2.4.5 Adaptivity

The ability of cache management policies to adapt to changes in the operating environment will likely be critical to achieving good file cache performance for mobile clients. This is not only because the available bandwidth will vary with location, but also because the work that a user performs will likely change as well, and thus the request stream seen by the file cache may have different characteristics. Since it may be difficult to know how best to adapt to various working environments, it may be necessary to solicit advice from the user on how various situations should be handled. This could be especially important in a weakly-connected environment, where a poor cache management decision could result in a much greater drop in performance than if the client was strongly connected [32]. It is important to note that in order for cache management policies to be adaptive, there must be some mechanism in place for the operating system to monitor and detect location changes and inform the cache of these changes.

2.5 Cache Management in Existing Distributed Environments

In traditional distributed file systems, client file caches are used both to improve the response time of client file accesses, and to reduce network and server load (which in turn improves scalability) [15, 54]. While cache management in current, non-mobile, distributed file systems is significantly simpler than in a mobile file system, the strategies employed are the starting points from which cache management mechanisms will be built. Therefore, a brief overview of cache management in several existing distributed file systems is given.

2.5.1 NFS

Sun's Network File System (NFS) [44] is one of the most widely used distributed file systems. File caching in NFS is relatively simple. File block (as opposed to whole file) caching is used, with an 8 kB block size. File caches are maintained in client memory. A delayed write-back mechanism is used, where writes are flushed from the cache after 30 seconds, or when the file is closed. The only cache consistency mechanism is the periodic (every few seconds) invalidation of file attributes (one of which is the time of last modification). While this means that a client will quickly detect if a cached file has been modified, it also creates a lot of network traffic and increases demands on the server (reducing scalability). As well, this approach requires that a client be in regular (and frequent) contact with the file server, and so this approach would not be well suited to a mobile environment.

2.5.2 AFS

The Andrew File System (AFS) [15], developed at Carnegie Mellon University, was designed to improve the scalability of network file systems. In order to reduce server load, clients have a two-level file cache. A smaller in-memory cache is used along with a larger, on-disk cache. Originally, whole-file caching was used, but current versions break large files into 64 kB "chunks". A write back on close mechanism is used to reduce write traffic to the file server. AFS's cache consistency mechanism is also designed to reduce file system traffic. Whenever a client caches a file, the file server issues a *callback* to the client, which is a promise to inform the client if the file is modified. If such a modification does occur, the server informs the client by issuing a *callback break*, which results in the invalidation of the copy of the file cached by the client. The use of callbacks results in a file system which could more easily be adapted for use in a mobile environment.

2.5.3 Sprite

The Sprite network file system [35, 38], from Berkeley, attempts to allow for improved file sharing between users. This is accomplished by having the file server detect when two users are concurrently writing to the same file, and by then disabling caching of that file on both clients. Thus, each write is reflected in the file almost immediately, and each read should return the most up-to-date copy. When a file is not being shared, writes are delayed 30 seconds before being written back to the file server. To improve overall cache performance, large in-memory caches are used, which are allowed to vary in size (based on the amount of memory not in use by the virtual memory manager). The variable cache size mechanism could also be useful for mobile clients, although in the form of an on-disk cache.

2.5.4 xFS

Berkeley's Experimental File System (xFS) [7] represents an attempt to further improve the scalability of distributed file systems. It is basically a set of optimisations to AFS. File caching is performed somewhat similarly to AFS, except that when a cached file has been modified, it is not written back to the file server. Instead, the server is notified of the modification, and the client becomes the owner of the file. If another client attempts to read the file in the future, it will be directed by the server to contact the new owner to obtain the most recent version. While such a scheme may not be practical in a mobile environment (because it may not be possible to contact the client with the most recent version of a file), the idea of simply informing the file server of modifications (rather than writing them back) has the potential to be useful in a mobile file system.

2.6 Challenges of the Mobile Environment

File caching in existing distributed file systems, as described above, has become a fairly well understood process. However, the mobile environment is significantly different from a LAN environment, and because of this, attempts to use the same cache management techniques in both environments may not be effective.

Mobility implies change in location, and this in turn implies a potential change both in the resources available to the cache manager and in the workload presented. Thus, the largest impact of the mobile environment is that it forces the cache manager to be *adaptive* to changing circumstances. The cache manager must be able to react to a new environment by altering management policies or by adjusting policy parameters. For a cache manager to be adaptive it must:

- *Make no assumptions about resource availability.* Since the manager likely can not anticipate what resources will be available to it in the near future (because connection to resource suppliers is uncertain), it can not rely on certain resources being available at a specific moment.

- *Be able to cope with resource shortages.* The cache manager must have a reasonable course of action to take when demand for a resource is greater than its supply.

- *Be prepared to handle different types of workloads.* The cache manager should be prepared to respond to workloads with varying degrees of locality, read-to-write ratios, inter-access times, etc.

Operating in a mobile environment also means that the performance impact of a bad decision by the cache manger can be much greater than in a LAN environment. In general, the weaker the connection a client operates with, the larger the performance penalty when a bad caching decision is made [32]. For example, if a read request is made to a file which has just been replaced in the cache, the need to re-fetch the file over a slow link will cause a much greater delay to the user than if the client was connected with a fast link. A cache manager must make fewer "bad" decisions in a mobile environment than in a strongly-connected environment.

Finally, the mobile environment forces the cache manger to be ready to handle failures. For example, a weakly-connected client may experience a loss of communication while a file transfer is taking place on behalf of the cache manager (such as during a write back). The cache manager must be prepared to handle this type of situation. If the manager simply aborted the transfer and restarted it at a later time, a significant amount of bandwidth (and the user's time) may have been wasted. However, if the manager was able simply to wait out the disconnection and resume the transfer in progress at a later time, the user may not even notice the failure. Work needed to implement "persistent connections" of this type is described in [55].

2.6.1 Principles of Effective Cache Management in Mobile Environments

Given the challenges that the mobile environment presents to cache management, there are several principles which should be followed in order to achieve effective cache management:

- *Know the current environment.* The cache manager can not be expected to adapt to a changing environment if it can not know what resources are available to it. The operating system must provide the cache manager with the current status of resources such as available bandwidth, disk space, and CPU load so that the manager can track resource availability and adjust its operation accordingly.

- *Know the current workload.* Caching can be successful only when there are known properties in the reference stream which can be exploited (such as temporal locality) to predict future user behaviour with a reasonable degree of accuracy. If the cache manager is to be expected to perform well in an environment where there may be significant changes in the characteristics of the workloads presented to it over time, there must be a mechanism to recognise the salient properties of the current reference stream. Such a "workload tracking" mechanism would be of even greater benefit if it could be used to anticipate future trends in the workload.

- *When in doubt, ask the user.* As noted in [32], user advice increases in value in an environment where poor cache management decisions may incur a large performance penalty. Therefore, while the overall goal of mobility support is to *hide* the fact that the user is not working in a strongly connected LAN environment, it may sometimes be better to impose on the user than to make a bad management decision. If advice is sought infrequently, and if the performance benefits of such advice are clear, requesting advice should not disturb the user's work to any great extent.

While these principles (especially the first two) certainly apply to caching in LAN-based file systems as well, they are even more critical when users are mobile. This is because of the larger degree of uncertainty which surrounds all file system activities when mobility is involved. Essentially, a traditional file cache manager needs to be aware of its relatively *constant* environment and workload, whereas the cache manager of a mobile client must cope with several different environments and the corresponding changes in workload associated with each. Additionally, mobile clients have a much reduced ability to recover from poor caching decisions. If bad replacement decisions are made by the cache manager, the user of a mobile client will be much more aware of the performance cost of the bad decisions than would a user working in a LAN environment. This makes awareness of the environment and workload even more critical for mobile client file cache management.

2.7 Cache Management in Existing Mobile File Systems

Several research projects have developed mobile file systems. Each of these has taken a different approach to the problem, and all have made different assumptions about the type of mobile environment which needs to be supported. Four of these file systems – AFS, Ficus, Columbia University's mobile file system, and Coda – are discussed in this section.

2.7.1 AFS

AFS did not originally offer support for mobile computing, but it has been enhanced at the University of Michigan to provide support for both disconnected and weakly-connected operation [16, 17, 19]. One of the design restrictions adopted by the designers was that, for compatibility reasons, only client software should be modified. The type of mobility supported is location independence, as is the goal of this work. Both disconnected and weakly-connected operation rely on a file cache which is "heated" through normal user activity while strongly connected – there are no mechanisms for a user to request explicitly that a specific file be cached. The cache manager uses a simple LRU replacement algorithm, and no prefetching is performed.

A mobile client's cache manager operates in one of four user-specified modes: connected, disconnected, fetch-only, or partially-connected. In connected mode, file operations are performed as in traditional AFS, with callbacks used to ensure cache coherence. While in disconnected mode, the cache manager assumes that the network is unavailable, and all cached files are assumed to be current (i.e. optimistic replication is used). All file system mutating operations are logged, and replay is performed when a user issues reverts to connected mode. Fetch-only mode is the same as connected mode, except that log replay is not performed. This mode is generally used to fetch a file not in the client's cache without having to wait for the entire to be replayed. Partially-connected mode is intended for use when the client has a weak network connection. Mutating operations are logged as in disconnected mode, but callbacks are used as in connected mode. The log file is optimised only when changing from disconnected mode to either connected or partially-connected mode. Additionally, "background replay" of the the log file is performed. All network traffic is placed into one of three queues: one for interactive, one for other network traffic, and one for log replay traffic. Lottery scheduling [52], a proportional-share randomised resource allocation algorithm, is used to arbitrate network access between the queues. Each queue receives some number of tickets (with interactive traffic receiving the most tickets and replay traffic receiving the least), and a lottery is held whenever the network is ready to transmit a packet. The "winning" queue then transmits a packet, and the process is repeated. Adaptive management of client file caches could be effected to a degree by varying the parameters of the scheduling algorithm.

2.7.2 Ficus

The Ficus file system [14], developed at the University of California at Los Angeles, is a peer-to-peer distributed file system, with "natural" support for disconnected operation, due to its optimistic replication policy. The peer-to-peer nature of Ficus makes it different from other mobile file systems, in that clients do not technically cache replicas of files stored on a server. However, many of the same issues arise since a client machine could be disconnected from the rest of the network at any time, and thus would have access to only those files stored locally at the time of the disconnection. A sophisticated predictive caching mechanism (called Seer) has been developed [23]. Seer observes a user's file access patterns and uses this knowledge to ensure that files which will likely be accessed in the near future are stored locally. Much of the work on Seer is relevant to the development of a prefetching algorithm for a mobile client's file cache.

2.7.3 Columbia University's Mobile File System

Columbia University has also developed a distributed file system with support for mobile computing. Much of the work centres on efficient file replica management [51, 49, 50]. A unique feature of this file system is the write-back mechanism, which is server-based rather than client-based; that is, the server notifies the client when a modified file should be written back (based on when it is convenient for the server). The server also informs the client when a modified file may be removed from its cache. Work has also been done on a prefetching mechanism for the client cache [28]. The approach taken is to store file reference patterns in tree structures, and then attempt to match current references with a stored pattern. As soon as a match is made, the remaining files in the pattern are prefetched. Once files have been prefetched, they are treated the same as demand-fetched files, and are subject to LRU replacement.

2.7.4 Coda

The Coda file system is a descendent of AFS, designed to ensure the "high availability" of files even in the event of network or server failures [37, 46]. From this design goal came support for disconnected, and later weakly-connected operation. Having been in use for several years, Coda has evolved to become a very sophisticated mobile file system.

A client's cache manager, called *Venus*, is the primary component responsible for supporting mobility. As in AFS, Venus operates in distinct modes. The *hoarding* mode is similar to the connected mode in AFS, relying on callbacks to ensure cache coherence. Additionally, Venus attempts to ensure that "critical" files are present in the cache during this time, through the use of LRU caching, user-specified hoard databases, and predictions based on long-term file reference histories (see Section 2.4.4). A disconnected client operates in *emulation* mode, similar to disconnected mode in AFS, with mutating operations being recorded in a log file for future replay. When a weak network connection is available to a client, it operates in *write disconnected* mode, a mix of the hoarding and emulation modes. File updates are logged, as when disconnected, but cache misses and coherence are handled in the same manner as when strongly connected. The network is also used for two other activities is this mode: *trickle charging* and *trickle reintegration* [10]. Trickle charging is a form of prefetching, while trickle reintegration is the process of propagating log file entries to a file server. Both of these activities are given lower priority for network access than demand fetches (i.e. cache misses).

Venus performs optimisations on the log file while the client operates in write disconnected mode. In order to ensure that optimisations can be performed, only log entries which have been in the log for some user-defined *aging period* are eligible for reintegration. (The default aging period is 10 minutes.)

Work has also been done to make Venus adaptive when operating in a weakly-connected environment [32]. Besides taking simple actions such as deferring reintegration and prefetching traffic until there is bandwidth available, more complex measures are taken. For example, a user patience model has been developed, which is used by Venus when a cache miss occurs. If the estimated time to retrieve the file exceeds a user's "patience threshold", Venus simply issues a cache miss message instead of downloading the file. As well, Venus solicits user advice to help decide which files are worth retrieving from a file server. Through the use of measures such as these, the developers claim that Coda has become adept at providing location-independent file service.

Chapter 3

Evaluating Mobile Client Cache Performance

Since client file caching is central to overcoming some of the key problems presented by the mobile environment (such as file availability and low-bandwidth network connections), it is desirable to manage the cache as efficiently as possible. In order to accomplish this, it is necessary to be able to evaluate various approaches to managing a mobile client file cache. This chapter discusses the cache management decisions that must be made, and how these decisions can impact cache performance. From this examination a set of metrics that can be used to evaluate cache performance in a mobile environment is presented.

3.1 Factors Affecting Cache Performance

While some of the factors that affect the performance of a mobile client's file cache are the same as those affecting a traditional file cache, the limitations of the mobile environment can increase the degree to which performance is affected. Additionally, there are factors affecting cache performance that are unique to the mobile environment.

For this study, the performance impact of seven factors was examined:

- **Preparation for Disconnection** – Unlike the file cache of a LAN workstation, a mobile client's file cache must operate in distinct "modes"; that is, the short-term objective of a mobile client's file cache manager changes over time. While the client is strongly connected (i.e. in *hoarding mode*), the goal of the cache manager is to prepare the cache for disconnection by hoarding files that it anticipates will be needed by the user in the future. The extent to which the cache manager is successful in performing this task has a significant impact on the performance of the cache while disconnected/weakly connected. Without a large fraction of the files referenced by a mobile user present in the cache at the start of a session, performance will be unacceptably poor.

- **Replacement Policy** – The choice of replacement policy is the factor most commonly associated with cache performance. Replacement policies have been extensively studied in the context of traditional file caches [1, 4, 54]. The impact of replacement policy on cache performance in the mobile environment is especially significant in the case of whole-file caching, where a single replacement decision can subsequently result in multiple cache misses.

- **Workload** – The performance of any cache is significantly impacted by the workload presented to it. Characteristics such as locality, the distribution of read and write requests, and the distribution of the size of files referenced can all have a significant impact on cache performance. As much as it is possible, it is desirable to achieve consistent cache performance across a wide of range of workloads. Achieving this is complicated by the fact that there are an unlimited number of possible workloads a cache may presented with, and this makes it impossible to account for all contingencies when designing or testing a cache management strategy.

- **Cache Size** – The size of cache is an important performance factor in any environment. In most cases, a larger cache will perform better than a smaller cache. However, this property is less likely to hold true in a mobile environment where file-based caching is employed. Nevertheless, the larger the difference in sizes between two caches, the more likely it is that the larger cache will perform better than the smaller cache.

- **Caching Unit** – There are two main choices of caching unit – file blocks, or whole files. Most LAN file caches use file blocks as the caching unit. When block caching is employed, there is the additional consideration of the size of blocks to cache. Alternative caching units have also been proposed, such as clusters of files [23].

- **Available Bandwidth** – In a strongly-connected environment, bandwidth is generally high (and constant) enough to keep transfer times small, and therefore bandwidth is not usually considered when examining cache performance. In a weakly-connected environment, however, the speed of the network connection is an important factor in the performance of write backs. The slower the network link, the more likely it is that write backs may interfere with read traffic, decreasing performance.

- **Write-Back Policy** – The choice of write-back policy has a significant performance impact in both strongly- and weakly-connected environments. In a mobile environment, however, the low bandwidth network connection means that any time a write back is performed there is a significant chance of a read request occurring, which would not be able to be processed until the write back is completed. This interference degrades read performance, which should be avoided whenever possible.

3.2 Cache Performance Metrics

To investigate the performance of cache management techniques it is necessary to have metrics by which performance can be quantified. Traditionally, miss rate (or its dual hit rate) has been the metric used almost exclusively to measure performance. Unfortunately, miss rate is not a good measure of cache performance in a mobile environment since not all misses are equal. When whole-file caching is in use, this assumption is not true – a miss on a small file will take less time to service than a miss on a large file. Evaluating a mobile client's cache performance by miss rate can therefore be misleading.

The inadequacy of miss rate as a performance metric means new metrics need to be developed to evaluate caching in a mobile environment. Such metrics must be able to reflect the qualitative perceptions of performance experience by the user of a mobile client [45]. A set of metrics has therefore been developed for this research to evaluate the performance implications of various cache management strategies. These metrics are based on the set of performance-influencing factors described in Section 3.1.

3.2.1 Bytes Missed

As mentioned previously, one of the problems with using miss rate as a metric is that it fails to account for the varying size of files referenced. The use of bytes missed as a metric presents a more accurate measure of how successful a cache is at reducing network file transfers. Measuring bytes missed also more closely reflects how a user would perceive performance. If two caches have the same miss rate, a user would be more satisfied with the cache which spent less time waiting for files to arrive from a server. It is the fact that bytes missed relates directly to the amount of time a user is stalled waiting for read operations to complete which makes it a useful metric of cache performance.

3.2.2 Time Until First Miss

If a user is working in disconnected mode, the only miss rate which is acceptable is zero – all cache misses are fatal (at least with respect to the current task). The user would like to be able to perform his/her work without needing to establish a network connection. The length of time from when disconnection begins until the first cache miss occurs is one measure of the cache's ability to provide the user with this functionality. The longer this period of time is, the more work a user is able to complete while disconnected. This metric is also useful in the context of weakly connected operation. A large time until first miss is an indication that few read misses will occur, which means that little bandwidth will be required to successfully operate while weakly connected.

3.2.3 Time Expansion

From a user's perspective, likely the most important measure of performance is the amount of time required to complete his/her work. Ideally, a mobile user would like to be able to perform his/her work in the same amount of time that it would take in a strongly-connected environment. Realistically, it is acceptable for a user's activities to take slightly longer in a mobile environment.

A useful measure of cache performance therefore is the ratio of time required to process all of user's file system requests while weakly connected to the time required to perform the same activity while strongly connected. This ratio is called the *time expansion factor*. This provides a measure of the "slowdown" incurred when operating in a mobile environment.

3.2.4 Write Interference Time

As described in Section 3.2.3, the total amount of time required to service all requests is a useful metric. This quantity consists of two components:

- **computation time** – time spent by applications executing instructions
- **I/O time** – time during which an application must wait for I/O operations to complete

In turn, I/O time comprises:

- **read service time** – time to perform file transfers resulting from cache misses
- **write interference time** – time during which a write back is in progress when the read queue is not empty

Write interference time is of interest since it directly measures the amount by which the decisions of a given write-back policy extend the total time required to service all requests in a trace.

3.2.5 Bytes Written Back

The amount of write-back activity which occurs is of interest since write backs help reduce the possibility of update conflicts occurring (see Section 2.4.2). Write-back activity can be measured in two ways: as the number of write backs performed, or as the total number of bytes written back. For this thesis, the number of bytes written back was chosen since there can be a wide range in the size of individual write backs. An ideal write-back policy would write back a large number of bytes while causing minimal interference with read traffic.

3.2.6 Log Size

The size of the mobile client's log file is also a metric of interest. The maximum size of the log indicates a minimum level of available disk capacity needed at the client (in addition to that required for the file cache and other local files) to operate in weakly-connected mode for the duration of the trace. The average size of the log is also of interest since it provides an indication of long-term trends in log file size, rather than just measuring peak size.

3.2.7 Time In Log

The period of time from when a file is modified until it is written back is of interest since it is during this period that update conflicts can occur. This can be measured by recording the total time each update spends in the log prior to write back. The shorter the time in the log, the less chance there is of an update conflict occurring.

3.2.8 Log Overwrites

The number of log entries overwritten by subsequent overlapping write activity is examined. This is an important metric since the reason for delaying write backs is to reduce the number that need to be performed, but if the frequency of log overwrites is low, the disadvantages of delaying writes outweigh the advantages.

3.2.9 Miss Rate

Although miss rate is not a good performance metric for a whole-file cache, it is somewhat more useful in evaluating the performance of a file-block cache, even in a mobile environment. Since each miss results in the same number of bytes transfered from the file server to the client, miss rate provides an accurate measure of how successful such a cache is at reducing network transfers.

Chapter 4

Simulation Environment

In order to examine the performance of various cache management policies a trace-driven simulation study was performed. The first step in this study was to gather a set of file system activity traces to serve as a workload for the simulator. A simulation model was then developed which describes the components and processes considered in this study. Finally, a simulator was implemented based on the simulation model, and experiments were conducted using the previously gathered workload traces as input. Each of the steps is described in this chapter. Further details on the trace collection process and the implementation of the simulator can be found in Appendices A and B.

4.1 Experimental Workload

The workload for the simulation experiments was generated from a set of file system activity traces of individual users performing day-to-day work in a research environment. These traces came from two locations: the University of Saskatchewan, and Carnegie Mellon University. All traces were gathered on strongly connected workstations in a Unix environment. The traces from each of these environments is described in detail below.

4.1.1 University of Saskatachewan Traces

The file system activity traces from the University of Saskatchewan were gathered using a customised version of a kernel-level tracing tool called *KItrace*. KItrace can record information about system calls, such as `open()` and `read()`. Special kernel code is required to allow KItrace to record system call information. Normally, information such as the device and inode numbers of files, and the type of system call made, are gathered by the kernel. For this work, modifications were made to the kernel (and to KItrace) to gather additional information, such as user ID's, process ID's, and file sizes. The details of the system calls recorded, and the corresponding parameters of each call, are given in Table 4.1.

It should be noted that `unlink()` system calls are not included in the traces used in the simulation experiments. Since inodes are reused by the file system, it is possible for a file to be deleted and a new file created with the same device and inode number as the deleted file. The simulator would not recognise the new file as being distinct from the original file. However, the error introduced by this fact is assumed to be small, since examination of traces containing `unlink()` calls showed that fewer than 1% of events were `unlink()` events.

Traces of two durations were collected: 24 hours, and seven days. The 24-hour traces were collected from a set of four workstations, from midnight to midnight. Several different users were active on each workstation during the course of the trace. Because of this, post-processing was necessary to extract the activity of individual users. This resulted in traces which contained several hours of a single user's activity over a 24-hour period. The five traces with the greatest amount of activity were selected for use in the simulation experiments.

The seven-day traces were collected simultaneously on the same four workstations used for the 24-hour traces. Due to storage limitations, the traces were broken into seven day-long segments, each of which was moved to off-line storage at the end of each day. After all traces were collected, the day-long trace segments were combined into a set of four week-long, continuous traces, each from a single workstation. From

Table 4.1: System calls and parameters recorded by *KItrace*

System Call	Parameters Recorded
`open()` `closef()`	time, process ID, user ID, error status, vnode, file type, device inode, file size, mode, file name
`read()` `write()`	time, process ID, user ID, error status, vnode, file type, device inode, file size, file offset, transfer size, file name

these traces, the activity of single users was filtered out and stored in separate trace files. These individual-user traces from each workstation were then combined (maintaining the time stamps on each event), resulting in traces which recorded the file activity of a single user (on all four workstations) over the period of one week. The four traces with the greatest amount of activity were then selected for use in the simulation experiments.

More details on the collection and processing of these traces are provided in Appendix A.

4.1.2 Carnegie Mellon University Traces

The traces obtained from Carnegie Mellon University (CMU) were also of 24-hour and seven-day durations. The traces were gathered using *DFSTrace*, a tool for recording file system activity in a distributed Unix environment [34], and had been used previously to study issues relating to disconnected operation [46]. While these traces contain a wide range of file system information, neither `read()` nor `write()` records were present, so the only trace records used in simulation experiments were `open()` and `close()`. The parameters recorded for each of these systems calls are given in Table 4.2.

The CMU traces were gathered on several general-use workstations. Each trace contains activity from several users. When used as input for simulation experiments, the CMU traces are first passed through a filter, which removes all records other than the `open()` and `close()` calls generated by a specified user. Thus, the activity of a single user is presented to the simulator.

Table 4.2: System calls and parameters recorded by *DFSTrace*

System Call	Parameters Recorded
`open()` `close()`	time, flags, mode, file descriptor, index, user ID, old size, size, file type, fid, directory fid, path

4.1.3 Trace Characteristics

Summary statistics of the traces used are given in Tables 4.3, 4.4, 4.5, and 4.6. The number of active hours in each trace was computed by subtracting the amount of "idle" time in the trace from time between the first and last references in the trace. "Idle" time was defined to be all time between events, if that time was equal to or greater than 15 minutes. The threshold of 15 minutes was chosen because, as shown in Figure 4.1, almost all inter-event times are less than 15 minutes, and almost all inter-event times greater than 15 minutes are *much* greater. Total Requests is a count of the number of events in the trace. Unique Files is the number of unique files referenced, and Unique Bytes is the total size of those unique files, measured when each file was first opened. Read and Writes Transfers are the sum of the sizes of all `read()` and `write()` calls (respectively) in the trace. Finally, Write Percentage is the percentage of `read()` and `write()` calls that are writes.

While most traces contain roughly the same amount of number of requests, Trace 5 in the set of University of Saskatchewan 24-hour traces is an exception, containing significantly more requests than the other traces of the same duration. This trace has likely recorded the file system activity of a series trace-driven simulations, activity which is common in the laboratory where the traces were captured. While this type of activity might not be performed at a mobile client, it has been included in this study to show the performance impact of intensive file system activity on a mobile client's file cache.

Table 4.3: University of Saskatchewan 24-hour trace characteristics

Trace	Active Hours	Total Requests	Unique Files	Unique Bytes (MB)	Read Transfers (MB)	Writes Transfers (MB)	Write Requests (%)
1	5.1	6389	59	4.4	498	< 1	81.4
2	6.8	1692	36	4.2	297	42	50.4
3	6.2	18909	277	15.8	5105	6	15.8
4	4.5	4647	342	11.0	1829	< 1	5.2
5	9.2	1386038	151	34.7	3526726	2730	46.3

Table 4.4: Carnegie Mellon University 24-hour trace characteristics

Trace	Active Hours	Total Requests	Unique Files	Unique Bytes (MB)	Total Transfers (MB)
1	6.0	2028	831	17	47
2	5.6	3684	212	2	55
3	6.9	2761	653	15	40
4	8.2	4142	925	10	73
5	5.4	3692	271	14	164

Table 4.5: University of Saskatchewan seven-day trace characteristics

Trace	Active Hours	Total Requests	Unique Files	Unique Bytes (MB)	Read Transfers (MB)	Writes Transfers (MB)	Write Requests (%)
1	12.9	21013	383	17	4596	4	37.8
2	7.0	53860	1000	18	4718	35	67.9
3	37.5	484746	883	100	323663	1165	39.4
4	8.5	17690	443	22	3723	2	25.2

Table 4.6: Carnegie Mellon University seven-day trace characteristics

Trace	Active Hours	Total Requests	Unique Files	Unique Bytes (MB)	Total Transfers (MB)
1	12.1	63390	545	31	1512
2	49.2	37805	836	33	619
3	113.5	13695	302	11	1279
4	25.6	6766	923	57	190

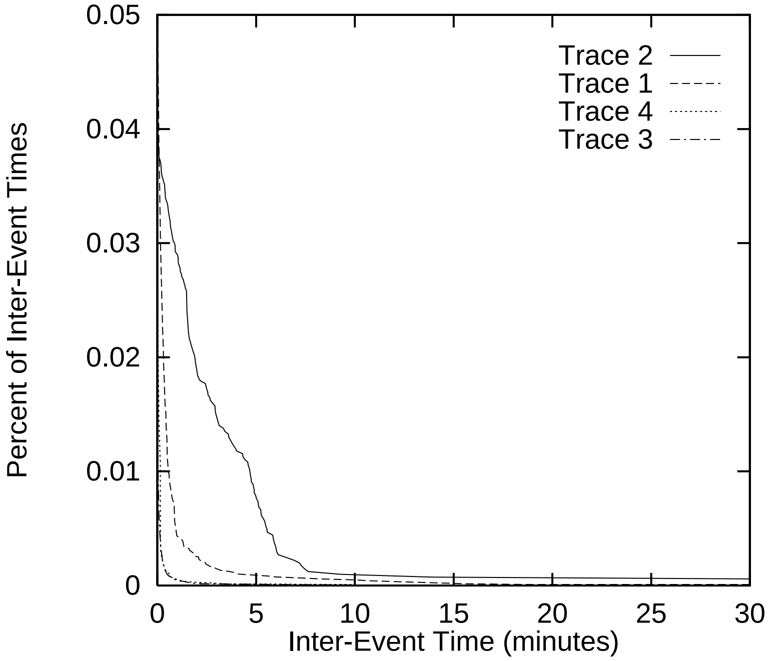

Figure 4.1: Inter-event time histogram, University of Saskatchewan 24-hour traces

4.2 Simulation Model

The simulation model describes the systems and processes of a mobile file system accounted for in this study. A description of the mobile environment is given in Section 2.2.1. Four principal components are modelled: the network, the file cache, the log file, and the network interface. Seven processes are modelled which define how these components interact with each other. Each of these components and processes is described in detail in this section.

4.2.1 Components Modelled

The four components modelled are the network link between the mobile client and the file server, the file cache maintained by the mobile client, the mobile client's log file, and the client's network interface. Each of these is described below.

- **Network** – The network connection between the mobile client and the home location where the file server resides is modelled by three parameters: bandwidth, propagation delay, and error rate. Bandwidth describes the capacity of the network link in bits per second. A bandwidth of zero means the client has no network connection (i.e. it is disconnected). Propagation delay is the time for a zero-length transfer between the client and the server to complete. Error rate is expressed as the average number of bits successfully transfered between errors. If an error occurs, the entire transmission in which the error occurs is restarted. Only a single transfer can use the network at any given time. For the experiments performed in this work, both propagation delay and error rate have been set to zero. The bandwidth parameter can be viewed as the *effective* bandwidth available after accounting for propagation delay and error rate, since as propagation delay and error rate increase, the effective bandwidth of the network decreases – the time to perform any given transfer will increase.

- **File Cache** – The mobile client's file cache is modelled as a collection of entries, sorted by arrival order. Cache entries can be either atomic or compound. In turn, atomic entries can be either variable-sized files or fixed-size blocks. Compound entries can consist of any number of atomic entries. Regardless of the number of items in a compound entry, it is treated as a single unit for the purposes of replacement. Cache entries may be sorted by one of several criteria, including recency of reference, frequency of reference, entry size, or some combination of these. The file cache is of a fixed size.

- **Log File** – The mobile client's log file is modelled as a sorted collection of records. Each record contains information regarding a write operation performed at the client. Specifically, the identity of the file written, the size of the write, and the offset into the file at which the write occurred are stored. The log file is maintained in a first-in first-out manner.

- **Network Interface** – The client's network interface is modelled as an arbitrator for access to the network. There are two sources of traffic which must share access to the network – read requests and write backs (no other network traffic is modelled). Each of these sources of traffic has its own queue. The manner in which traffic from these queues is given access to the network is determined by the write-back policy in use (see Section 4.2.2). Only a single transfer can occur on the network at any given time.

4.2.2 Processes Modelled

The seven processes that describe the interaction between the modelled components are presented below.

- **Generation of File I/O Requests** – All action in the model is driven by file I/O requests generated by applications. These application requests are modelled using the contents of a trace file (see Section 4.1). Each event recorded in the trace is time stamped, and describes an operation on a specific file. Depending on the contents of the request, the contents of the cache or log may be modified, and a network transfer may occur.

- **Change in Network Characteristics** – The characteristics of the network connection between the mobile client and the home file server (bandwidth, propagation delay, and error rate) may change at any predetermined time during a simulation. Such changes represent a change in the user's working location. This change in network characteristics occurs only in between other activities – the network will never change during a file transfer, for example.

The change is instantaneous, and no simulation time passes during the change. The bandwidth of the network is used to place the client into one of three modes. The client is considered to be strongly connected if the bandwidth is at least 10 Mb/sec, and is considered to be operating in *hoarding mode*, preparing for subsequent disconnection as the user becomes mobile. If the client is not strongly connected but the network bandwidth is non-zero, the client is considered to be weakly connected. Otherwise, the client is considered to be disconnected.

- **File Transfers** – File transfers occur whenever a cache miss occurs, or when a write back is performed. File transfers are modelled as a change in the simulation time during which no other activity occurs. The amount of simulation time that passes depends on two factors: the size of transfer and the current network characteristics. Specifically, the transfer time is given by:

$$t(s,b) = \frac{s}{b} \tag{4.1}$$

  where:

  $t(s,b)$ is the time required to perform the transfer (sec)
  s is the transfer size (bits)
  b is the network bandwidth (bits/sec)

  Only a single file transfer can occur at any given time.

- **Cache Replacement** – Cache replacement occurs whenever a new item needs to be cached (due to a miss), but there is not enough free space in the cache to accommodate the new item. Items are removed from the cache one by one until there is enough free space in the cache to accommodate the new entry. The exception to this is if the new entry is larger than the size of the cache itself. In this case, no cache replacement is performed. Five cache replacement policies are modelled:

  - **RAND** – A random entry is chosen for replacement.

  - **LRU** – The least-recently-used entry is chosen for replacement.
  - **LFU** – The least-frequently-used entry is chosen for replacement.
  - **Keep Large** – The smallest entry is chosen for replacement.
  - **Keep Small** – The largest entry is chosen for replacement.

- **Hoarding** – When the client is in hoarding mode (i.e. it has a strong network connection), files are gathered in the file cache according to the hoarding policy currently in use. To reduce the complexity of simulation, all the hoarding policies are essentially the same as the replacement policies described above.

- **Log File Optimisation** – Whenever a write is processed, a log entry is created, which is stored in the log file. However, it is common for multiple writes to occur to the same file. When this occurs, it is possible for a subsequent write to "overwrite" the same part of the file as an earlier write. This presents an opportunity to optimise the log file by reducing the number of entries in the log. Whenever a new entry is added to the log, it is compared against all existing entries. If an existing entry is found such that both entries describe an update to the same portion of the same file, the two entries are merged into a single entry. For example, if one log entry recorded a write of 100 bytes to file A starting at byte 0 and another entry recorded a write of 100 bytes to file A starting at byte 50, these two entries would be combined into a single entry recording a write of 150 bytes to file A starting at byte 0. Any time two such matching entries are combined, a *log overwrite* is said to have occurred.

- **Write Backs** – Whenever the client is weakly connected, the contents of the log file have a chance to be transfered back to the home file server. Each transfer of a log entry is called a *write back*. The size of a write back is equal to the size of the log file entry being transfered. Only a single log entry is written back at a time, and the oldest entry in the log is always written back first. The decision of when to perform a write back is made by the write-back manager. Five write back policies are modelled:

- **No Write Backs** – No write backs are performed while weakly connected.
- **Idle Write Back** – Write back the oldest log entry in its entirety only when the network has been idle for some fixed period I.
- **Delayed Write Back** – Write back the oldest log entry in its entirety if it has been in the log for at least some fixed aging period A, and if the network has been idle for some fixed period I.
- **Preemptive Write Back** – Begin writing back the oldest log entry if it has been in the log for at least some fixed aging period A, and if the network has been idle for some fixed period I. Halt the write back if a read request occurs.
- **Lotto** – Hold a lottery (with the write queue receiving W tickets and the read queue receiving R tickets) after each transfer (read or write) has completed, or if the network has been idle for some fixed period I. Write back the oldest log entry in its entirety whenever the write queue wins a lottery.

Each of these policies attempts to perform writes whenever possible while giving priority to read traffic. The No Write Backs policy is included simply to provide a basis of comparison. At the other extreme is the write through policy, in which writes are performed immediately with the application blocking until the write has been completed. We do not examine this policy, since its appropriateness is questionable even in a strongly-connected environment, and it is even more poorly suited for a mobile client. The Idle Write Back policy gives priority to reads by (ideally) performing writes only when the network is otherwise idle. It attempts to detect idleness using fixed timer-based prediction [13]. The same method is used in the Delayed and Preemptive policies. The Delayed policy attempts to capture the previously described advantages of delaying write backs. The Preemptive policy functions the same as the Delayed policy, with the difference that a write back in progress will be halted if a read request is received, and will be resumed from the same point at a later time. The Lotto policy is based on the policy described in [19], with modifications to work on a whole-file (rather than packet) basis.

The Lotto and Idle Write Back policies are called *aggressive* polices, whereas the Delayed and Preemptive policies are called *conservative*. These classifications are based on the fact that the Lotto and Idle Write Back policies have fewer restrictions on when a write back can be initiated, and therefore more write backs are performed when using one of these policies. The Delayed and Preemptive write back policies are by comparison conservative, since they require more conditions to be met before a write back is performed. The aggressive policies focus on performing more frequent write backs, with secondary consideration given to the possibility of degrading read performance. The conservative policies attempt to avoid interfering with reads as much as possible, and as a result perform fewer write backs.

4.3 The Simulator

A discrete event simulator was implemented in C++ for use in these performance experiments. The simulator is based on the simulation model described in Section 4.2, and consists of 67 object classes in over 10,000 lines of code. Details of the algorithms used by the simulator are given in Appendix B.

Since no measured results exist to compare simulation results against, validation of the simulator was done through the use of small "test traces" – truncated versions of the traces used in this work. Detailed output from the simulator was checked for correctness as each event was processed. The object-oriented design of the simulator facilitated this type of testing, since each object could be tested individually.

Chapter 5

Hoarding Experiments

The ability of a mobile client's file cache to successfully hoard files in preparation for later work in a weakly connected or disconnected environment is critical to ensuring acceptable performance. Without successful hoarding, disconnected operation is impossible, and weakly connected operation will be extremely frustrating for the user.

A series of experiments was performed to address two questions related to hoarding. These experiments involved a strongly-connected client, which after some period of time becomes disconnected and continues to operate without a network connection. The first question examined is how effectively various hoarding policies prepare the cache for operation in a mobile environment. Since hoarding is such a critical procedure, understanding the performance implications of different approaches to hoarding is necessary to ensure acceptable levels of performance at the mobile client. As well, the level of disconnected performance that can be obtained through the use of a successful hoarding mechanism is examined. This is relevant not only to disconnected operation, but also to weakly-connected operation, since it provides baseline measurements against which client performance while weakly connected can be compared.

This chapter describes the performance study of hoarding policies undertaken. The policies examined and the metrics used to evaluate performance are discussed. The experimental design is then described and the experimental results are presented and discussed.

5.1 Policies Investigated

One of the major cache management decisions which must be made when supporting disconnected operation is the choice of hoarding policy. Unfortunately, the use of file system activity traces does not lend itself to a study of advanced hoarding mechanism, such as those used in the Coda file system [21, 46], since these mechanisms rely on user advice in choosing which files should be hoarded. Therefore, only relatively simple hoarding algorithms could be used in this study. This means that the results of these experiments should be viewed as a lower bound on cache performance. More sophisticated hoarding policies should be able to perform at least as well as the policies used in this study.

Each of the policies examined operates in the same manner. As files are referenced they are added to the cache. If the cache becomes full, some files in the cache are replaced. The set of files in the cache at the time of disconnection forms the set of files available to the client.

Five hoarding policies were examined (see Section 4.2.2):

- **RAND** – Random replacement is used. This policy provides a lower bound on performance, since no policy should perform worse than RAND.

- **LRU** – The least-recently-used cache entries are replaced first. This is the most commonly used cache replacement policy. LRU will perform well as a hoarding policy only if files which are commonly referenced in the short term are also commonly referenced in the long term.

- **LFU** – The least-frequently-used cache entries are replaced first. Each cache entry has a reference count associated with it, which is incremented each time it is accessed. The entries with the lowest reference counts are the least frequently used. LFU does not perform well in most caches because some entries will build up large reference counts in a short period of time, then never be replaced even if they are never referenced again for a long period of time. However, this behaviour may actually be desirable in a hoarding policy, since the goal of hoarding is much longer-term than traditional caching. To the extent that frequency of reference is a good indicator of long-term file reference patterns, LFU will perform well as a hoarding policy.

- **Keep Large** – The Keep Large policy replaces the smallest files in the cache first. This policy ignores both the recency and frequency of reference patterns, instead attempting to reduce the size of file transfers when misses do occur. Since the largest files referenced are those which will stay in the cache, fewer total files will present. For this policy to work well time saved by reducing the average size of a transfer caused by a cache missed must offset the larger number of cache misses which will occur.

- **Keep Small** – The Keep Small policy replaces the largest files in the cache first. This policy is the opposite of the Keep Large policy, and it assumes that good performance can be achieved by maximising the number of files in the cache. For the Keep Small policy to perform well the miss rate must be low enough to make up for the large size of transfers which must be performed when misses do occur.

5.2 Metrics

Three metrics are used in the evaluation of hoarding policies: miss rate, bytes missed, and time until first miss (see Section 3.2). Miss rate is included simply to show how it can be misleading as a metric in a mobile environment. Bytes missed is the primary metric of interest, since it provides the best indication of how well the client cache has been prepared for operation in a mobile environment. Time until first miss is also measured, providing an indication of how well the cache is prepared to support disconnected operation.

5.3 Experimental Design

The simulation parameters used in these experiments are shown in Table 5.1. All five hoarding policies described in Section 5.1 were used. Simulations were performed using both 24-hour (U of S and CMU) and seven-day (U of S and CMU) traces. This is to provide an indication of performance over both moderate and long periods of disconnection, across a wide range of workloads. For all experiments the client is initially strongly connected, and after some period of time the client becomes disconnected. This simulates a user working at his/her home location, then voluntarily becoming disconnected, moving to a new location, and resuming work from this new (disconnected) location. The length of time the client is strongly connected is referred to as the *hoarding period*. Hoarding periods ranging from half an hour to ten hours were used for the 24-hour traces, and from six hours to six days for the seven-day traces. The wide range in hoarding period allows for an examination of the impact hoarding period has on disconnected cache performance. Client cache sizes ranging from 1 MB to 20 MB were simulated. Additionally, experiments were performed using an "infinite" client cache. This is done simply by setting the cache size to be larger than the number of unique bytes referenced in any of the traces. When an "infinite" cache is used, no files are ever replaced in the cache, and thus the performance of the cache will be as high as possible given the other parameters in use.

Four sets of experiments were run. The first set examines the time until first miss. This provides an indication of the level of performance achieved through the use of hoarding. The next two sets focus on the miss rate and the bytes missed by the cache. These help show how much bandwidth would be required to service cache misses in a weakly-connected situation. If few bytes are missed by the cache while disconnected, it is a good indication that even a low-bandwidth network connection would result in acceptable levels of performance in a weakly-connected environment. The final set of experiments examines the impact of hoarding policy on strongly-connected performance. This is done by measuring the bytes missed by the cache during

Table 5.1: Factors and levels used in hoarding experiments

Factor	Levels
Trace Files	U of S 24-hour, CMU 24-hour, U of S seven-day, CMU seven-day
Hoarding Policy	RAND, LRU, LFU, Keep Large, Keep Small
Hoarding Period	0.5, 1, 2, 3, 4, 5, 6, 8, 10 (hours) (24-hour traces) 0.25, 0.5, 1, 2, 3, 4, 5, 6 (days) (seven-day traces)
Cache Size	1, 5, 10, 15, 20 (MB), infinite
Dependant Variables	hit rate, bytes missed, time until first miss

the hoarding period. These measurements indicate the extent to which strongly-connected performance must be sacrificed in order to achieve good disconnected performance, if at all.

The results of these experiments are based on cache performance after the hoarding period has ended. For example, the number of bytes missed is counted only during the period that the client is disconnected. The exceptions to this are the experiments which measure bytes missed while strongly connected – here results are gathered only during the hoarding period.

5.4 Simulation Results

The findings of the hoarding experiments are presented in four parts, each outlining and discussing the simulation results from one of the four sets of experiments described above (see Section 5.3).

5.4.1 Time Until First Miss

Two approaches were taken to measure the ability of a mobile client's file cache to support disconnected operation. The first was to measure the number of references (i.e. read and write events) from the time of disconnection until the first miss occurred. The results of these experiments when performed with an "infinite" client cache are shown in Figure 5.1 (note the logarithmic scale).

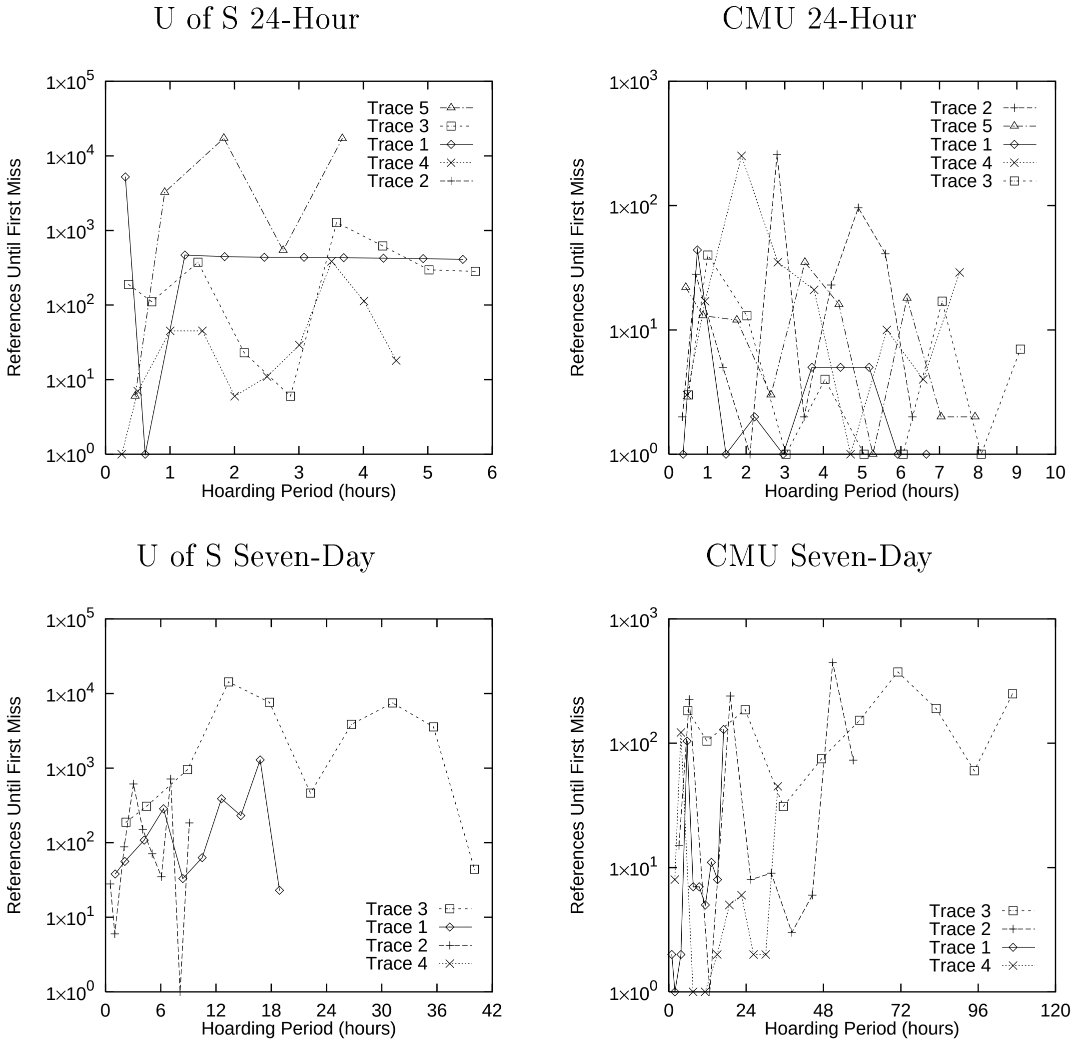

Figure 5.1: References until first miss with an infinite cache

Several observations are apparent from these results. First, there is no consistent relationship between the length of the hoarding period and the number of references until the first miss occurs. Second, in absolute terms, the number of references until the first miss is generally quite low. Finally, the results vary considerably from trace to trace, although the results from the U of S traces more closely resemble each other than those from the CMU traces of the same duration.

The fact that increasing the length of the hoarding period does not always result in an increase in the number of references until first miss likely indicates that user activity spanning several unrelated tasks is present in the traces. The sawtooth shape of the curves supports this possibility. If the hoarding period ends just before a user begins a new task (thereby starting a new working set of files), the number of references until first miss will be very short. However, if the hoarding period ends sometime after work on the new task has begun, it is possible that most of the user's working set will be present in the cache, and therefore many references will be satisfied by the cache before the first miss occurs.

The low number of references until the first cache miss occurs indicates that hoarding techniques can not simply rely on past behaviour to predict which files will be requested in the future. Since the results shown are from the use of an "infinite" client cache, every file referenced during the hoarding period is present in the cache for the remainder of the simulation. Even so, the client cache is in most cases unable to properly provide file availability for more than a small number of references.

It is interesting that the results obtained from traces originating from the same source (i.e. either the U of S or CMU) resemble each other more closely than the traces of the same duration (i.e. the 24-hour traces or the seven-day traces). This likely reflects different types of user work captured by the traces from each location. This demonstrates the degree to which the benefit of hoarding (at least of the type examined in these experiments) is workload dependent. The greater number of references until first miss seen in the U of S traces indicates the presence of longer-term locality of reference than in the CMU traces. If better hoarding policies were used (i.e. ones which have the ability to hoard files other than those recently referenced by a user), it is possible that the variation in results between the U of S and CMU traces would be smaller.

Although measuring the time until first miss by reference count is an accurate method, it provides only limited intuition of "real world" performance. Therefore, results are also presented using the actual time from the start of disconnection until the first miss occurs. The problem with this approach, however, is that the results can be greatly influenced by the amount of idle time present in a trace (see Section 4.1.3). If a trace has a long period of time where no events occur, the time until the first miss could be large even if the first reference after disconnection was also the first miss. Ideally, the time which passes while the client is active should be counted, while the time the client spends idle should not be. Unfortunately, it is impossible to know exactly when the client is active and when it is idle. The best that can be done is to use an estimation of when the client is idle. For these experiments, if the time between events is less than fifteen minutes, the time is counted (i.e. the client is assumed to be active). The time between events more than fifteen minutes apart is not counted (i.e. the client is assumed to be idle).

Figure 5.2 shows the time until first miss measured in "active" hours. The same erratic relationship can be seen between the length of the hoarding period and the time until the first miss. The large difference in time until first miss between traces also remains. Both these results occur for the same reason as when references are used to measure the time until first miss.

For the majority of the traces, the time until the first miss occurs is less than one hour. This is to be expected given the low number of references until first miss. It is unlikely that this level of performance would be considered acceptable by most mobile users. Again, this shows that more sophisticated hoarding policies are needed to properly support disconnected operation.

5.4.2 Bytes Missed

The number of bytes missed by the cache while disconnected was examined for both "infinite" client caches as well as for smaller caches using various hoarding polices. The results using an "infinite" cache provide a basis of comparison for the performance of the various hoarding polices operating with smaller caches.

Figure 5.3 shows the number of bytes missed while disconnected for an "infinite" client cache (note the logarithmic scale). All values to the right of where a curve intersects the x-axis are zero. The number of bytes missed with a hoarding period of zero show the total number of bytes transfered in each trace. Since no cache replacement is performed, the number of bytes missed must decrease (or stay the

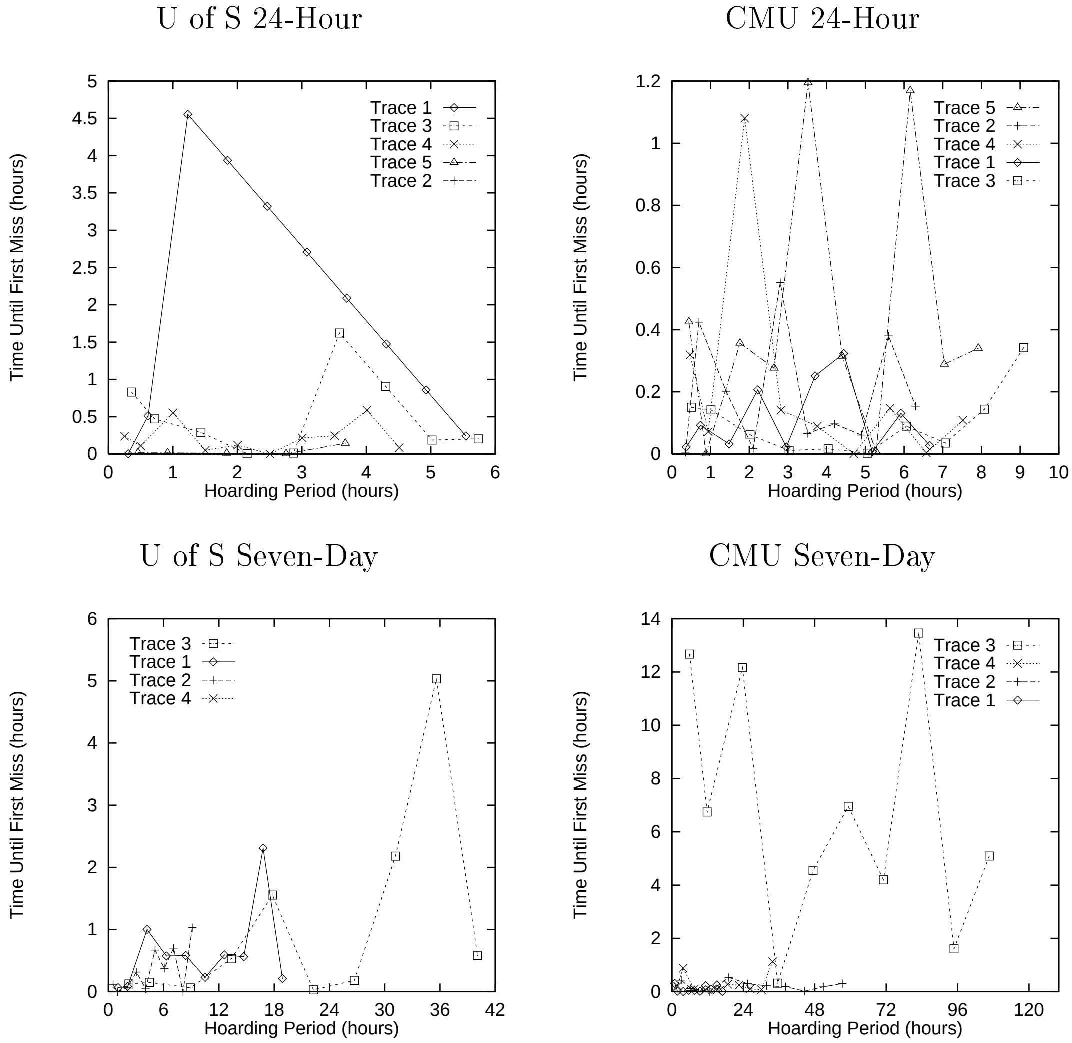

Figure 5.2: Active time until first miss with an infinite cache

same) as the hoarding period increases. The curves are flat where no new files are referenced in the trace during the period corresponding to a given hoarding period.

The results for the 24-hour traces show a significant decrease in the number of bytes missed when the hoarding period is increased from zero to five hours. After this time, the decrease in bytes missed is small. However, the number of bytes missed after five hours of hoarding is generally quite small – less than 20 MB. Amortised over the rest of the simulation period (19 hours), this would require only roughly a 2.4 kb/sec network link to complete all transfers (less for the majority of 24-hour traces). More bandwidth would be required to ensure acceptable performance if

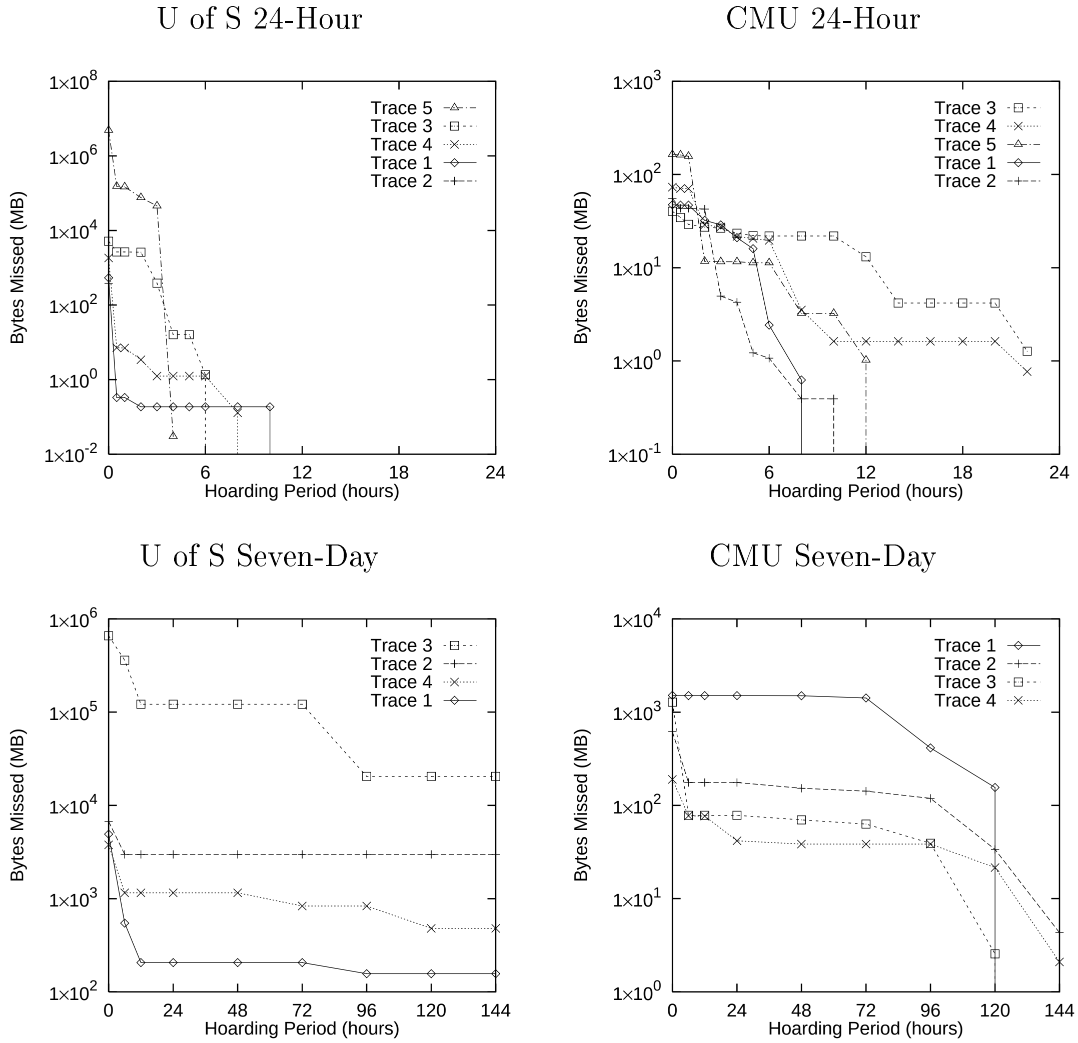

Figure 5.3: Bytes missed while disconnected (infinite client cache)

misses were very bursty, however.

The results for the seven-day traces follow the same pattern as the 24-hour traces, but with the number of bytes missed increased by roughly an order of magnitude in most cases. After six hours of hoarding, the number of bytes missed is reduced to less than 200 MB for most seven-day traces. The bandwidth requirements to complete these transfers during the rest of the simulation period (6.75 days) are only slightly larger than those for the 24-day traces: approximately 2.8 kb/sec.

These results are encouraging from the point of view of weakly connected operation, since the amount of bandwidth needed to satisfy cache misses is not unreasonably large. While the bandwidth requirements calculated above are only a minimum requirement, they serve to show that even with a network connection of, say, 14.4 kb/sec, good performance should be achievable.

Since the above results are all based on the use of an "infinite" client cache, it is necessary to determine the effect of smaller caches at the client, as well as the effect of various hoarding policies. If the number of bytes missed when smaller caches are used can approach the values observed for "infinite" caches, it should be possible to provide good performance in a weakly connected environment.

Figures 5.4, 5.5, and 5.6 show the bytes missed results for three of the U of S 24-hour traces. The effect of cache size on bytes missed is clear. In Traces 1 and 4, the number of bytes missed is reduced by three orders of magnitude when client cache size is increased from 1 MB to 5 MB (for Trace 1) and from 5 MB to 15 MB (for Trace 4). Trace 3 does not experience this decrease, likely because of the much larger number of bytes transfered in that trace.

In many cases, each hoarding policy performs roughly the same. This is particularly true when very small caches are used, or when a larger (e.g. 15 MB) cache is used. If a cache is so small that only a few files can be stored, the hoarding policy has little influence on which files are in the cache at any given time. When a larger cache is used, very little replacement occurs, and again the hoarding policy has little impact on performance. When moderate cache sizes are used, several observations can be made. First, the RAND policy, as expected, experiences a wide range in performance. For example, for Trace 1 with a client cache of 3 MB, there are some hoarding periods for which RAND outperforms all other polices, and others for which RAND performs worse than all other polices. In Trace 4, The Keep Small and LRU policies also experience some sensitivity to hoarding period. Overall, LFU and LRU are the two policies which most consistently outperform other policies.

The results for the CMU 24-hour traces are shown in Figures 5.7, 5.8, and 5.9. Little variation is seen in the performance of the hoarding policies. Similarly to the U of S 24-hour traces, the LFU hoarding policy demonstrates consistently good performance.

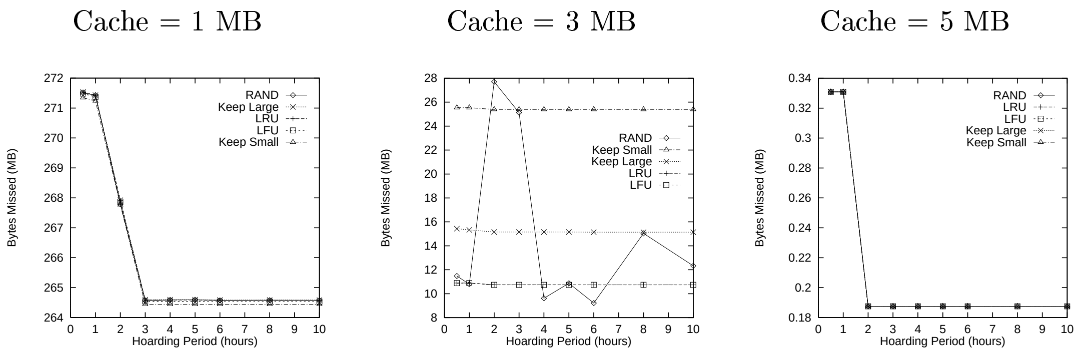

Figure 5.4: Bytes missed while disconnected - 24 hour U of S Trace 1

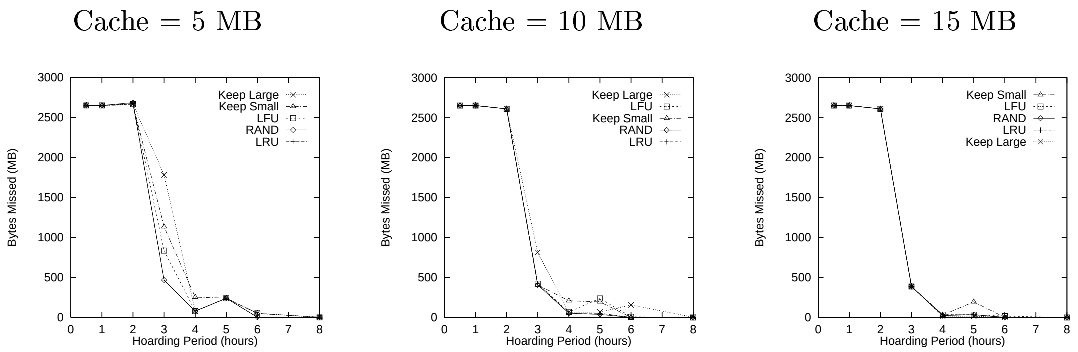

Figure 5.5: Bytes missed while disconnected - 24 hour U of S Trace 3

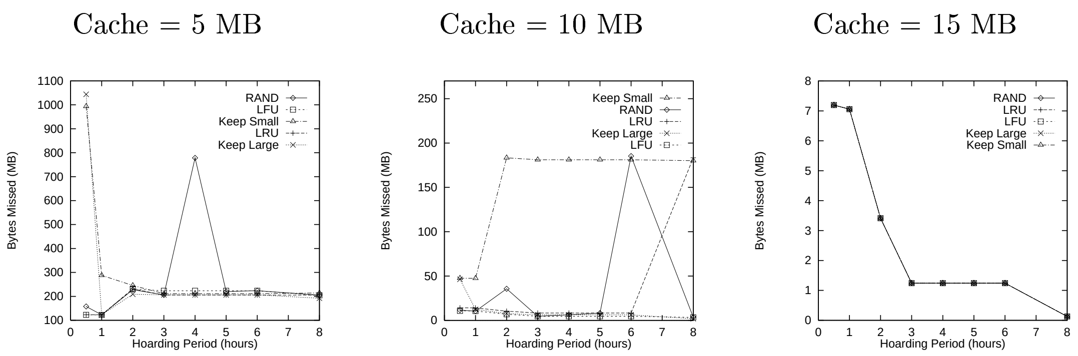

Figure 5.6: Bytes missed while disconnected - 24 hour U of S Trace 4

Cache = 1 MB   Cache = 10 MB   Cache = 20 MB

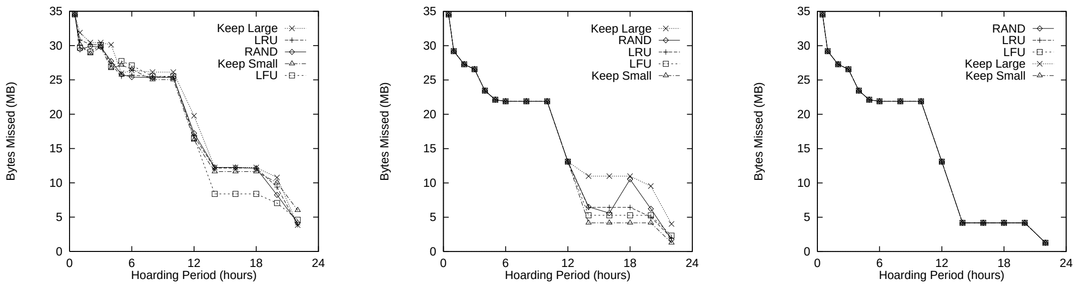

Figure 5.7: Bytes missed while disconnected - 24 hour CMU Trace 3

Cache = 1 MB   Cache = 10 MB   Cache = 20 MB

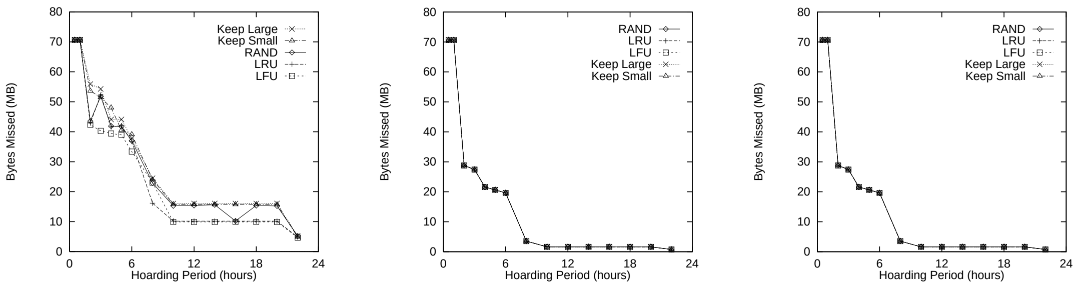

Figure 5.8: Bytes missed while disconnected - 24 hour CMU Trace 4

Cache = 1 MB   Cache = 10 MB   Cache = 20 MB

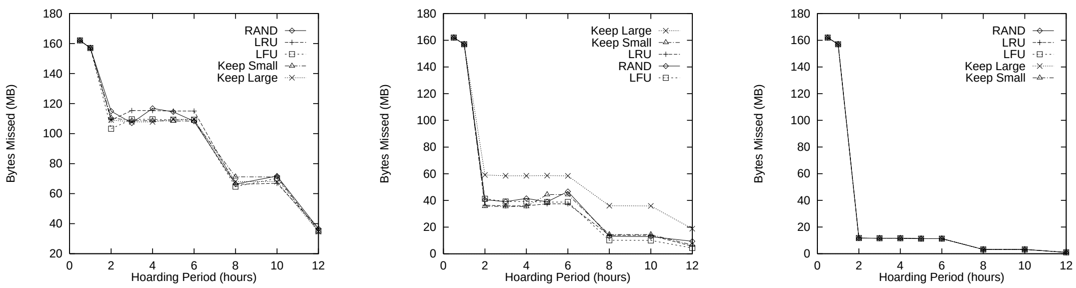

Figure 5.9: Bytes missed while disconnected - 24 hour CMU Trace 5

The most significant difference in the results of the CMU and U of S 24-hour traces is the fact that cache size has a much smaller impact on the number of bytes missed in the CMU traces. In many cases, increasing the size of the client cache has virtually no effect on the number of bytes missed for small hoarding periods. For larger hoarding periods, the number of bytes missed is reduced with larger caches, but the reduction is only about one order of magnitude.

Figures 5.10, 5.11, and 5.12 show the bytes missed results for the U of S seven-day traces. As was seen with the 24-hour traces, the number of bytes missed decreases with increased cache size. The same erratic performance of the RAND policy is also seen. The Keep Small policy performs particularly well for Trace 3, even better than LRU and LFU. However, this performance is not consistent across traces – for Trace 2, for example, the Keep Small policy performs significantly worse than all other policies.

The results for the CMU seven-day traces are shown in Figures 5.13, 5.14, and 5.15. More variation is seen in the performance of the hoarding policies than was seen for the 24-hour CMU traces, but the smaller impact of cache size on bytes missed continues. In many cases the RAND and Keep Large policies do not perform well. As seen in other traces, the LRU and LFU policies generally offer the most consistent performance.

Figures 5.16, 5.17, 5.18, and 5.19 present more clearly the impact of client cache size on bytes missed. The results show the number of bytes missed after a hoarding periods of five hours (for the 24-hour traces), and 24 hours (for the seven-day traces). The benefits of larger cache sizes is clear for most hoarding policies. Interestingly, the performance impact of cache size on the LFU policy is somewhat erratic. In several cases the number of bytes missed by the LFU policy either increases dramatically as the cache size is increased, or it does not decrease in the same manner as other policies. This is not a desirable performance characteristic, since a user will expect improved performance if more of the client's disk is devoted to caching.

Several key observations can be made from these results as a whole. First, workload has a very large impact on the performance of hoarding policies. The

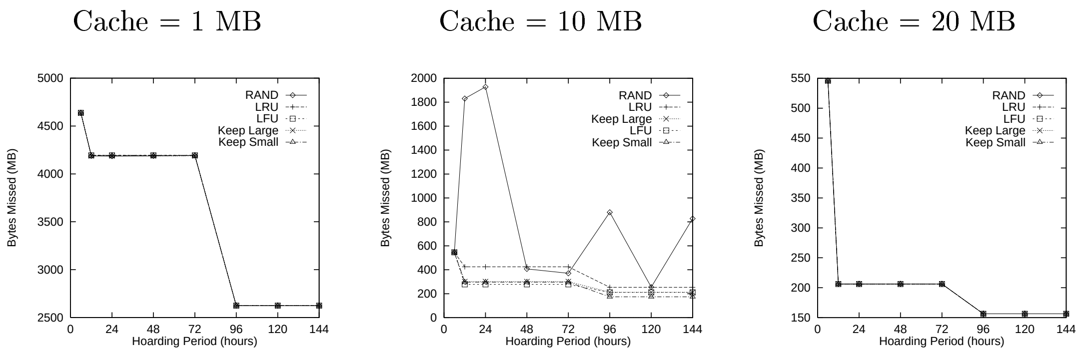

Figure 5.10: Bytes missed while disconnected - seven-day U of S Trace 1

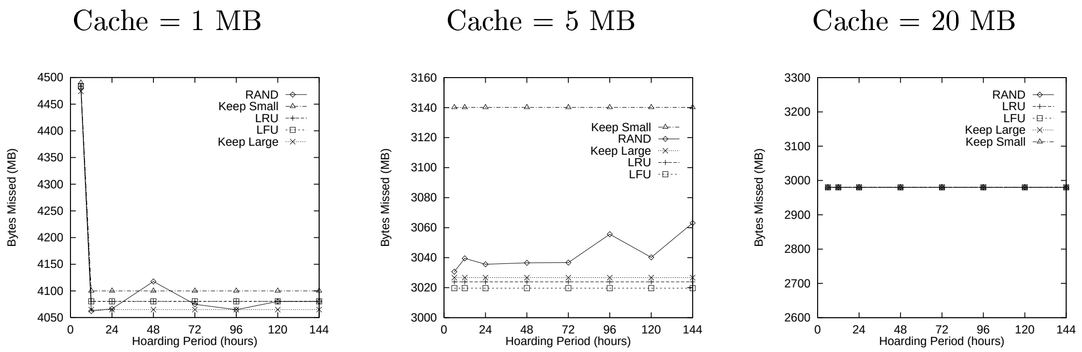

Figure 5.11: Bytes missed while disconnected - seven-day U of S Trace 2

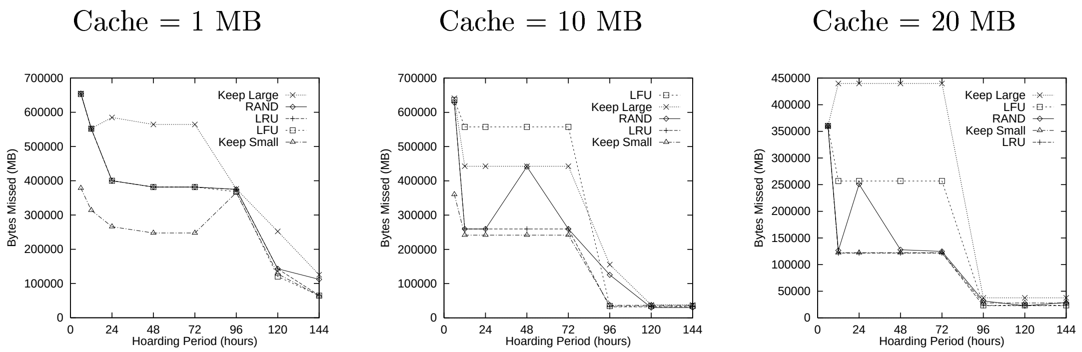

Figure 5.12: Bytes missed while disconnected - seven-day U of S Trace 3

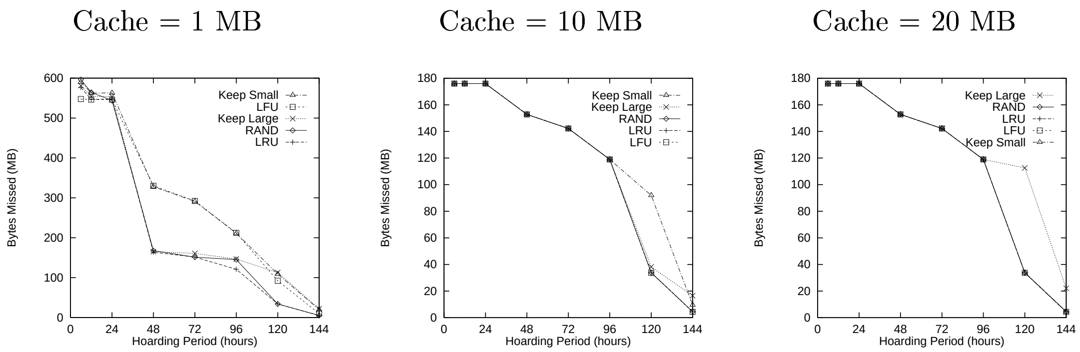

Figure 5.13: Bytes missed while disconnected - seven-day CMU Trace 2

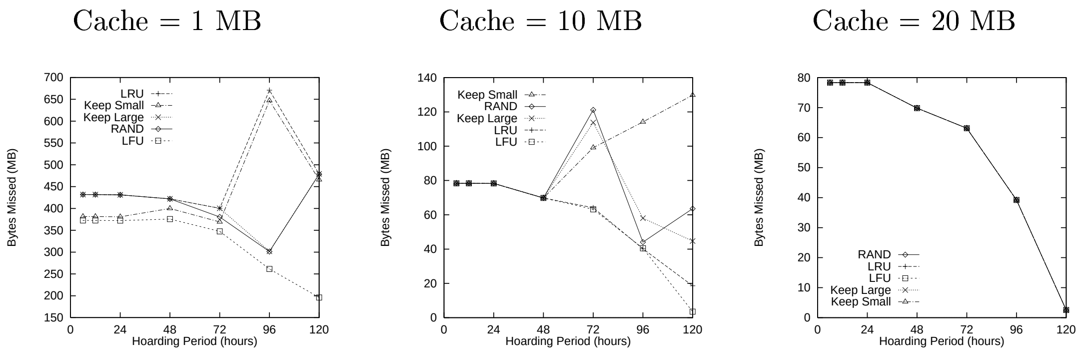

Figure 5.14: Bytes missed while disconnected - seven-day CMU Trace 3

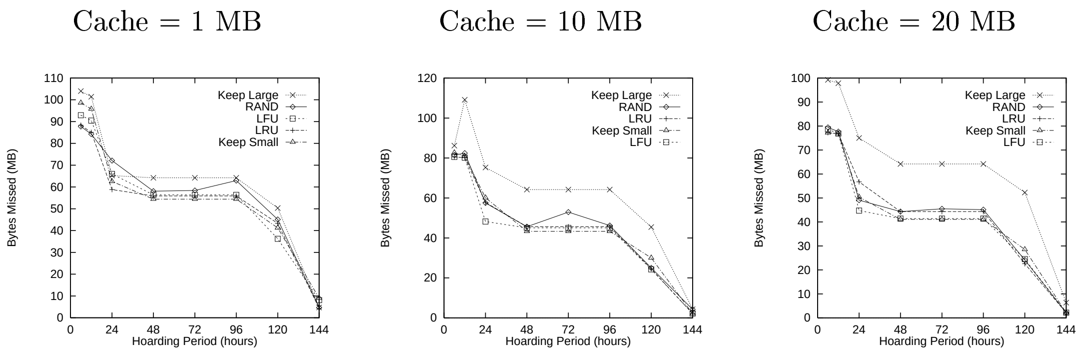

Figure 5.15: Bytes missed while disconnected - seven-day CMU Trace 4

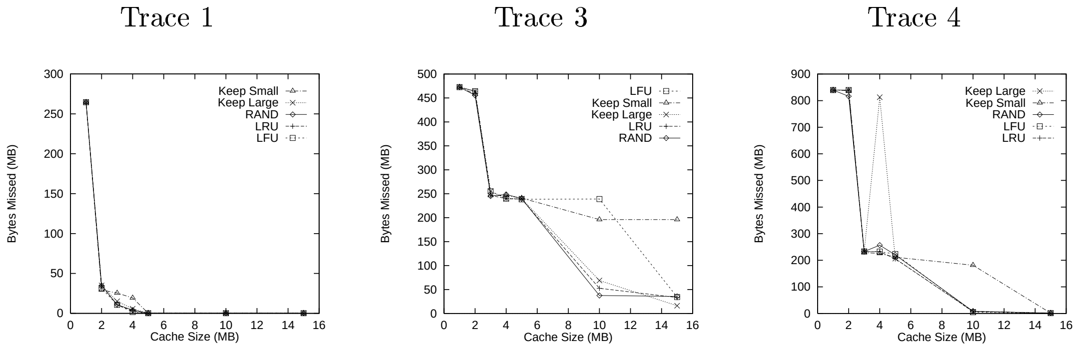

Figure 5.16: Bytes missed while disconnected – 24 hour U of S Traces, hoarding period = 5 hours

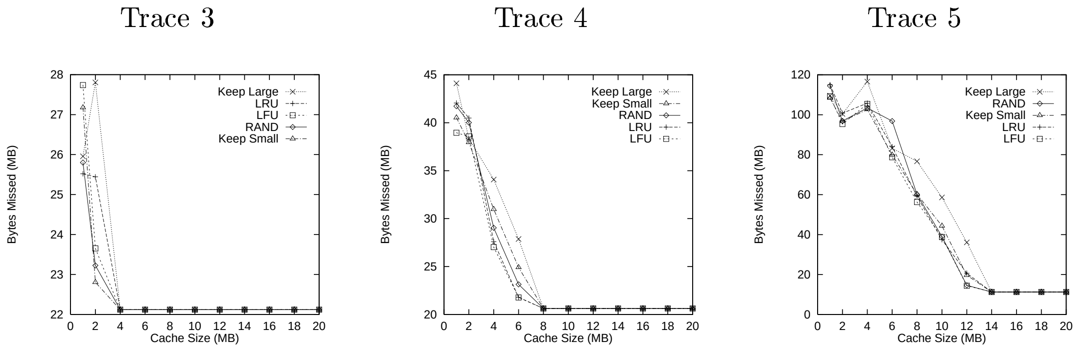

Figure 5.17: Bytes missed while disconnected – 24 hour CMU Traces, hoarding period = 5 hours

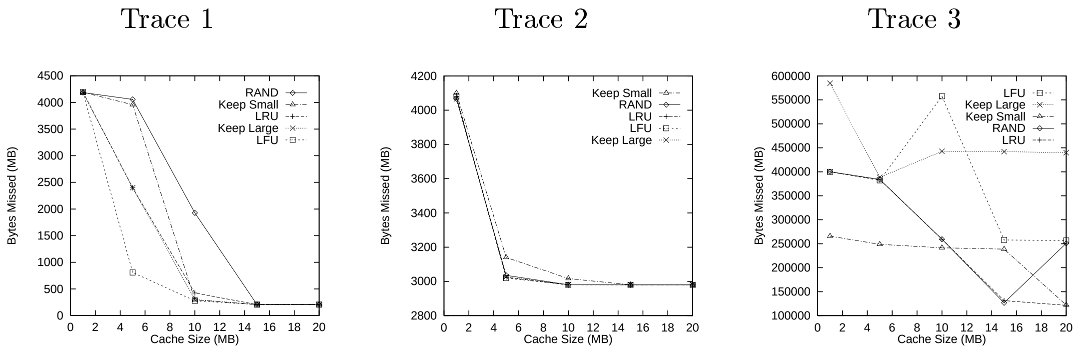

Figure 5.18: Bytes missed while disconnected – seven-day U of S Traces, hoarding period = 24 hours

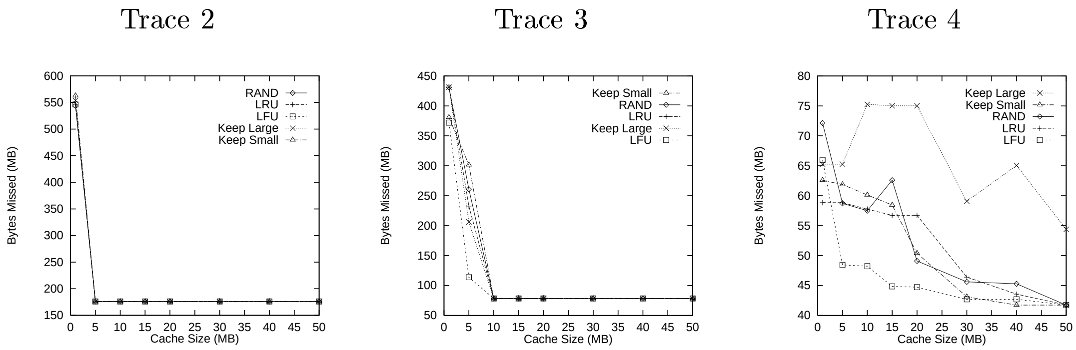

Figure 5.19: Bytes missed while disconnected – seven-day CMU Traces, hoarding period = 24 hours

differences in the performance of many of the hoarding policies across traces are very large. This means that a key characteristic of a successful hoarding policy must be consistency of performance across a wide range of workloads. The RAND, Keep Large, and Keep Small policies seem to be the most susceptible to differences in workloads. This is likely because the distribution of file sizes referenced has more influence on which files are cached than the actual pattern of references to these files. The LRU and LFU policies seemed to provide somewhat more consistent performance. In many cases, the LFU policy resulted in fewer bytes missed than the LRU policy. However, the LRU policy seems to provide more consistent behaviour over a wide range in cache sizes.

Another observation is that the size of the client cache often has much more impact on the number of bytes missed than the hoarding policy in use. While this is not surprising, it verifies the intuition that the number of bytes missed can be significantly reduced through the use of adequately large file caches. It is encouraging that for the workloads examined, "adequately large" file caches appear to be on the order of only 20 MB – certainly not unrealistic for a mobile client.

Finally, the number of bytes missed when caches of 10-20 MB are used in conjunction with moderate-length hoarding periods is encouragingly small. This means that in a weakly connected environment the number of bytes missed should not overwhelm even a relatively slow network connection.

5.4.3 Miss Rate

As previous discussed, the use of miss rate as a metric in a disconnected environment is likely to provide little insight into the performance of a hoarding policy, and may even provide false indications of performance. The results presented here demonstrate this (using hit rate instead of miss rate).

The hit rate of each hoarding policy when "infinite" client caches are used is shown in Figure 5.20. The wide range in hit rates between traces is immediately obvious. The 24-hour traces have higher hit rates overall, and their hit rates are more greatly affected by the length of the hoarding period. An interesting observation is the fact that occasionally a longer hoarding period results in a lower hit rate, even when no cache replacement occurs. This is because hit rate is a function of both the number of cache hits and the total number of references. If no new files are added to the cache during some part of the hoarding period (i.e. no misses occur), the hit rate of the remaining time may be reduced, since the same number of misses will occur, but in a span of fewer references.

In general, the hit rates seen for most traces are quite low, especially for the seven-day traces where the hit rates are generally below fifty percent. As well, increasing the hoarding period does not significantly improve the hit rate for these traces. These results are in contrast to those seen when using bytes missed as a metric. Then, the low number of bytes missed when longer hoarding periods were used indicated relatively good performance. If only the hit rate was presented, it would appear that the cache is performing much worse than it actually is.

For the reasons seen in these examples, hit rate is not used as a metric for any subsequent experiments.

5.4.4 Bytes Missed During Hoarding Period

The previous experiments have focussed exclusively on client file cache performance after the hoarding period has ended. This is the time when cache performance has the most impact on the user's perception of performance. However, a cache's

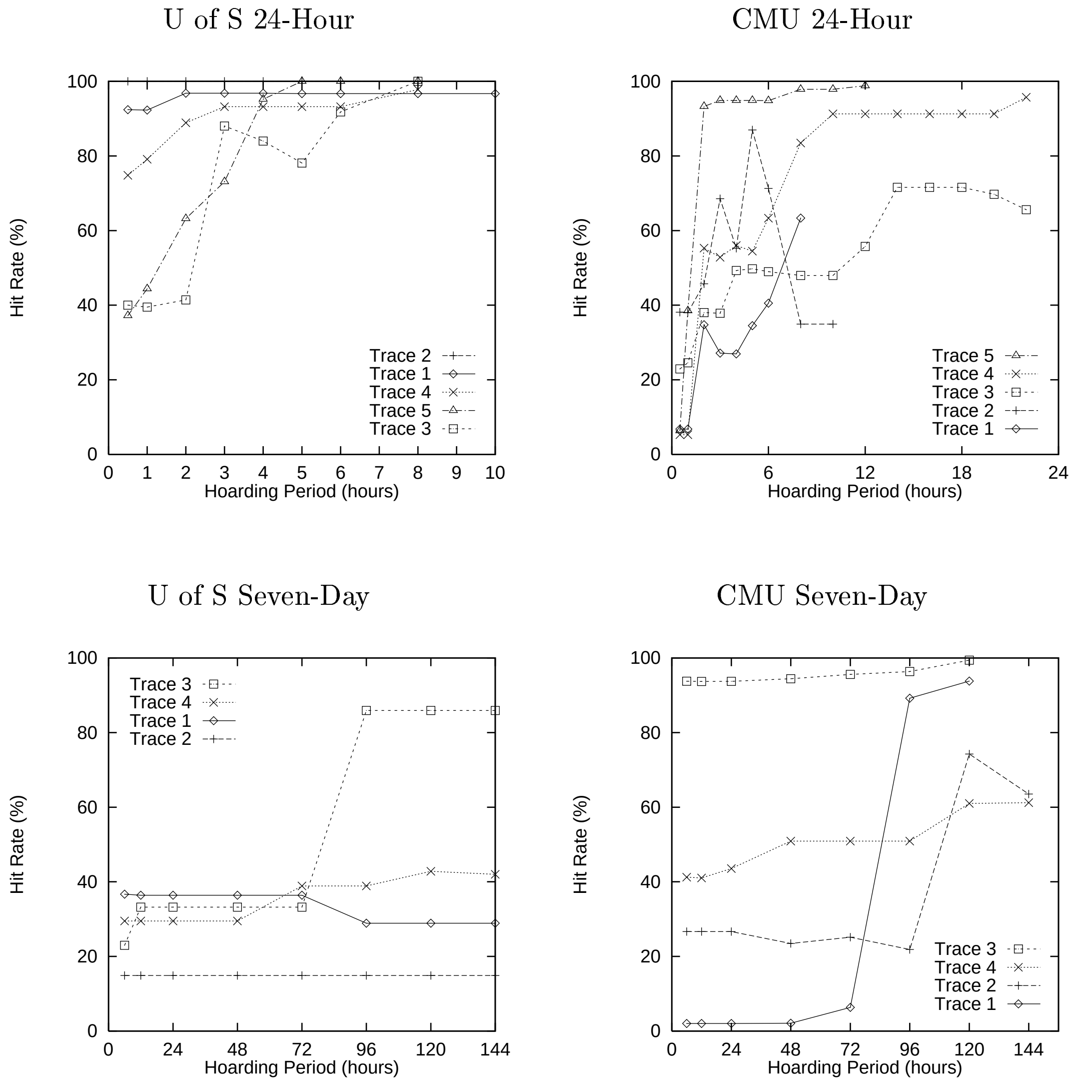

Figure 5.20: Hit Rate while disconnected (infinite client cache)

hoarding policy will also have an impact on performance while the client is strongly connected. Even though the performance impact of the cache is smaller in a strongly-connected environment (compared to a weakly or disconnected environment), it would be undesirable for strongly-connected performance to be greatly decreased by the choice of hoarding policy.

The number of bytes missed during the time the client is strongly connected (i.e. during the hoarding period) is shown in Figures 5.21, 5.22, 5.23, and 5.24. In almost every case, the number of bytes missed is high for very small caches, but for even

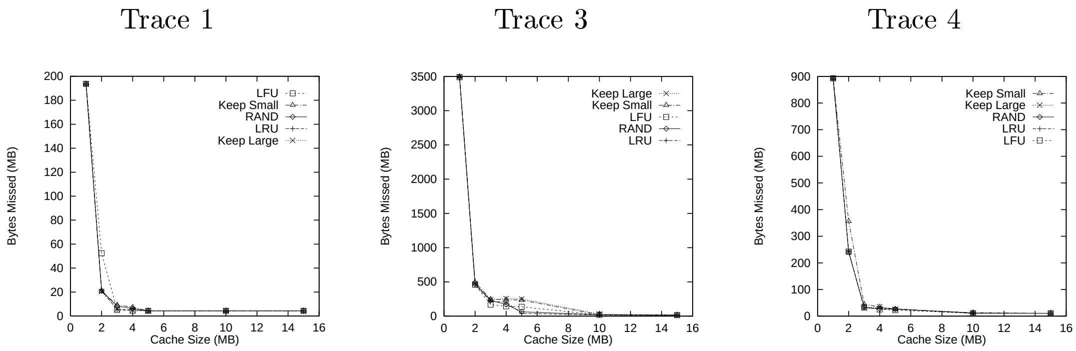

Figure 5.21: Bytes missed while hoarding – 24 hour U of S Traces, hoarding period = 5 hours

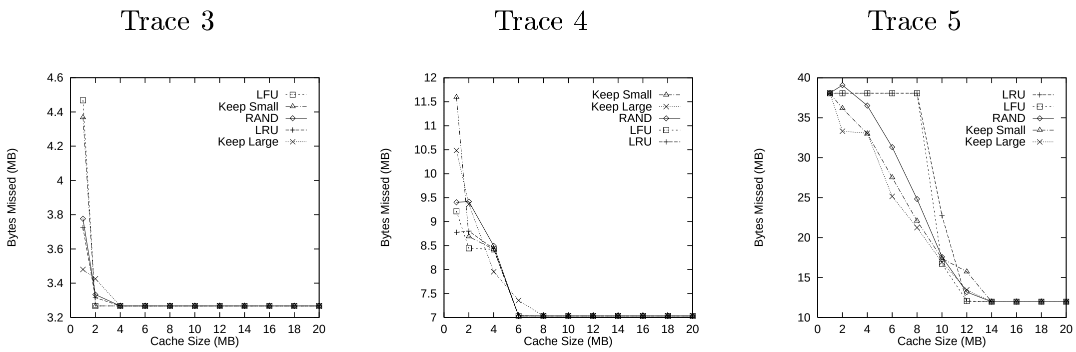

Figure 5.22: Bytes missed while hoarding – 24 hour CMU Traces, hoarding period = 5 hours

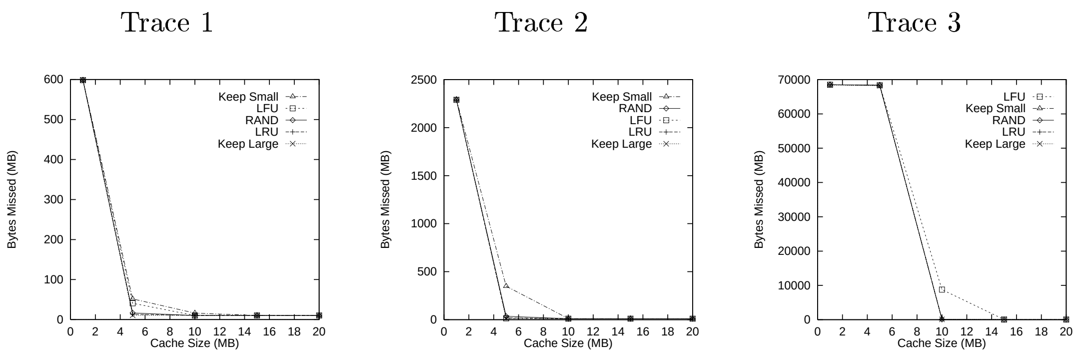

Figure 5.23: Bytes missed while hoarding – seven-day U of S Traces, hoarding period = 24 hours

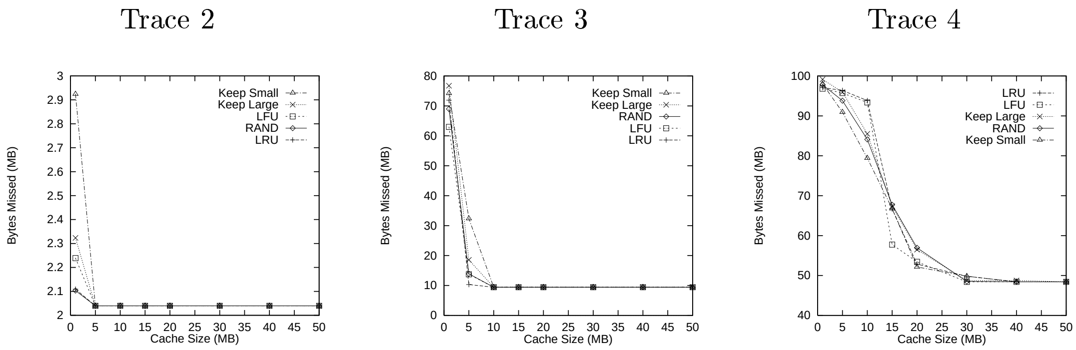

Figure 5.24: Bytes missed while hoarding – seven-day CMU Traces, hoarding period = 24 hours

moderate sized caches (5-10 MB) very few bytes are missed. This pattern is seen across all hoarding polices.

Not surprisingly, LRU is the policy which generally performs the best when very small caches are used, with the performance of the Keep Small and LFU policies usually being worse than even RAND. However, the fact that none of the hoarding polices examined causes a large number of bytes to be missed while hoarding when even moderate client caches are used is encouraging since it means that a user will not likely have to sacrifice performance while working in strongly-connected environment in order to achieve better post-hoarding performance.

5.5 Conclusions

The goal of these experiments was to determine the effectiveness of various hoarding policies in preparing a mobile client's file cache for work in a mobile environment, and to examine the performance of a disconnected mobile client using these hoarding polices. Simulation results show that the size of the cache has a large impact on cache performance. However, very large caches are not necessary to achieve good performance. Workload was also found to greatly impact cache performance. The use of different traces of the same duration resulted in widely varying results in many cases.

In general, increased hoarding time was found to reduce the number of bytes missed by the cache while disconnected. The LRU policy appears to be the most consistent at reducing bytes missed across various hoarding periods and cache sizes. The LFU policy also performed quite well at times, and may perform even better if much longer hoarding period were used (on the order of several days). The actual number of bytes missed by the cache while disconnected were (in the best cases) found to be small enough that even a weak network connection should be able to service those cache misses which do occur in a reasonable amount of time.

File hit rate was found to be a poor performance metric. The performance of the cache indicated by file hit rate was much lower than when measured with bytes missed. Additionally, the file hit rate of even an infinite cache can decrease as the hoarding period is increased. These characteristics severely reduce the value of file hit rate as a metric for mobile client file cache performance.

The ability of the hoarding policies used in these experiments to support disconnected operation was found to be inadequate. Experiments found that the time until first miss for all policies was too small to make disconnected operation successful – less than one hour in most cases. To properly support disconnected operation, either much longer hoarding periods are required, or more sophisticated hoarding polices must be used. The unpredictable relationship between hoarding period and time until first miss seen in these experiments indicates that the former solution may not be successful. Therefore, hoarding policies which do not rely simply on short-term user reference patterns to identify the files which should be hoarded are likely needed to support disconnected operation.

The performance impact of hoarding policy on strongly-connected cache performance was also investigated. When moderate to large caches are used, each of the policies examined was found to perform roughly the same. For smaller cache sizes, the Keep Small and LFU policies performed poorly, while LRU generally performed quite well.

It should be restated that the results of these experiments should be viewed as lower bounds on the ability of hoarding to support operation in a mobile environment. Any hoarding policy which is aided by user advice should be able to perform better than the policies examined in this study. Any such policy would likely combine user advice and past reference behaviour to determine which files to cache. The results of this study would be relevant to determining how past reference behaviour will affect the performance of a hoarding policy.

Chapter 6

Write Backs

The goal of file system support for mobile computing is to provide users with functionality comparable to that of a LAN environment regardless of their location. While maintaining file availability is a large part of this, the ability to update files and have these updates be reintegrated with the home file system in a timely manner is also very important. To accomplish both these tasks, the limited bandwidth of a weak connection must be managed carefully. Ideally, the cache manager should provide the advantages of performing write backs without decreasing the performance of reads. Since these goals conflict with each other, the problem becomes one of trying to gain the most benefit from performing write backs with minimal interference to reads.

A series of experiments was performed to investigate various approaches to accommodating both reads and writes over a low bandwidth connection. These experiments involve a scenario in which a mobile client hoards for some period of time, after which it changes location, and resumes working with a weak network connection. While working, read misses are serviced over the network link, and file updates are propagated back to the file server. The resource and performance implications of using various write-back policies in this environment are examined.

This chapter describes the write-back policies investigated, as well as the metrics used to evaluate these policies. The presentation of results is followed by a discussion of the performance impact of performing write backs in a weakly-connected environment.

6.1 Policies Investigated

Accommodating both read and write traffic in a weakly-connected environment is much more difficult than in a strongly-connected environment. Since the demand placed on a mobile client's network connection will often exceed the bandwidth available, it is necessary to implement policies which control access to the network. For this study, there are two sources of network traffic – a read queue and a write queue. Requests in the read queue are given access to the network immediately upon the network becoming idle. While weakly connected, requests in the write queue (i.e. write backs) are given access according to the write-back policy currently in use.

Five write-backs policies are examined: No Write Backs, Idle Write Back, Delayed Write Back, Preemptive Write Back, and Lotto (see Section 4.2.2 for a detailed description of these policies).

6.2 Metrics

The metrics used in this study attempt to identify some of the important relationships between resource consumption and performance benefits which exist when scheduling reads and writes in a weakly-connected mobile environment. Six metrics are used: time expansion, write interference time, bytes written back, time in log, log size, and log overwrites (see Section 3.2).

Time expansion is the metric which relates most directly to the interests of the user. The longer a user must wait for network operations to complete, the less satisfied he/she is likely to be. Write interference time indicates how much of the time expansion is due to the decisions made by the write-back policy. The number of bytes written back while weakly connected and the amount of time entries spend in the log help show how the scheduling decisions made by a write-back policy affect the chance of file conflicts occurring. The size of the log file and the number of log overwrites which occur provide an indication of the effect of write-back policy on the amount of local storage required at the client.

6.3 Experimental Design

The experiments performed for this study are summarised in Table 6.1. Since the CMU traces do not contain read or write events, only U of S traces could be used. The set of seven-day traces was chosen since it was felt that longer traces would provide more detailed results. For this study, the client is strongly connected for the first 24 hours, during which it hoards files using the LRU hoarding policy. The LRU hoarding policy is used since it demonstrated consistent performance in previous simulations (see Section 5.4). Similarly, a hoarding period of 24 hours was chosen since previous results indicate that it is long enough to prepare the file cache for weakly-connected operation. After this time, the client becomes weakly connected, with a network connection ranging from 9.6 to 57.6 kb/sec. While weakly connected, LRU replacement is performed at the client. Client cache sizes ranging from 1 to 20 MB are used. Although it is unlikely that a mobile client would operate with a cache of only 1 MB, the results obtained when such a small cache is used provide an indication of how increased read traffic (due to the greater number of cache misses) affects the performance of write-back policies.

An initial set of experiments was performed to determine reasonable values for write-back policy parameters. The values used for each parameter are shown in Table 6.2. The Idle Write Back policy was used to examine the effect of the Idle Threshold (I) parameter, while the Delayed Write Back policy was used to investigate the Minimum Aging Period (A) parameter. From these experiments values for I and A were chosen to be used for all remaining experiments.

6.4 Simulation Results

The results of the write-back policy simulations are presented in several sections, each presenting results with respect to a different metric.

Table 6.1: Factors and levels used in write-back experiments

Factor	Levels
Trace Files	U of S seven-day
Hoarding Policy	LRU
Hoarding Period	1 day
Replacement Policy	LRU
Bandwidth	9.6, 14.4, 28.8, 57.6 (kb/sec)
Cache Size	1, 10, 20 (MB)
Write Back Policy	No Write Back, Idle Write Back, Delayed Write Back, Preemptive Write Back, Lotto
Dependant Variables	time expansion, write interference time, bytes written back, time in log, log size, log overwrites

Table 6.2: Write-back policy parameter settings tested

Parameter	Values
Idle Threshold (I)	0, 0.001, 0.01, 0.1, 1 (sec)
Minimum Aging Period (A)	1, 5, 15, 60 (min)
Number of Read Tickets (R)	11
Number of Write Tickets (W)	1

6.4.1 Parameter Settings

Two parameters govern the behaviour of most of the write-back policies examined. The Idle Threshold (I) parameter is used in all policies except for No Write Backs, and the Minimum Aging Period (A) parameter is used by the Delayed Write Back and Preemptive Write Back policies. These parameters can have a large impact on how write backs are performed by a given policy. Simulations were performed to attempt to identify reasonable values for these parameters which could be used for subsequent experiments.

The effect of the Idle Threshold parameter I was investigated by examining its effect on write interference time and log size. Since the purpose of the Idle Threshold parameter is to attempt to ensure that write backs are performed only when the network is otherwise idle, measuring write interference time provides a good indication of the appropriateness of the Idle Threshold setting. An appropriate setting for I should result in a small write interference time. However, it is not desirable simply to set the threshold to an extremely large value, since this will result in too few write backs being performed. The size of the log file is measured to provide an indication of this effect. A small log file indicates a large number of write backs performed, while a large log file suggests that the Idle Threshold is too large.

The effect of I on write interference time across a range of bandwidths is shown in Figure 6.1. Results are shown for Traces 1, 2, and 4, using a client cache size of 10 MB and the Idle Write Backs policy. Write interference time rises sharply for values of I less than 0.1 seconds in each trace, and for all bandwidths. The effect of I on log size is shown in Figure 6.2. As expected, the size of the log increases as I is increased, but this increase is slower for values of I greater than 0.1 seconds. These results each suggest that 0.1 seconds is an appropriate value for I, and this value is used in all further experiments.

A suitable value for the Minimum Aging Period parameter A was chosen based on the parameter's effect on the number of log overwrites and the size of the log. The purpose of this parameter is to reduce the number of write backs which need to be performed by allowing subsequent writes to "overwrite" previous entries in the log (see Section 4.2.2). As A increases, the number of log overwrites should increase, but at the same time the size of the log should increase. A value for A which results in a large number of log overwrites without causing the log to become too large is desirable.

Figure 6.3 shows the effect of A on the number of log overwrites which occur. The Delayed Write Backs policy is used, with a client cache size of 10 MB (and I set to 0.1 seconds). Although there is a wide range in the fraction of log entries overwritten between traces, the effect of A is the same in each case. The number of overwrites decreases rapidly for values of A less than 5 minutes, and increases very slowly for values greater than 15 minutes. The effect of A on log size is shown in Figure 6.4. The size of the log increases significantly as A becomes large, with

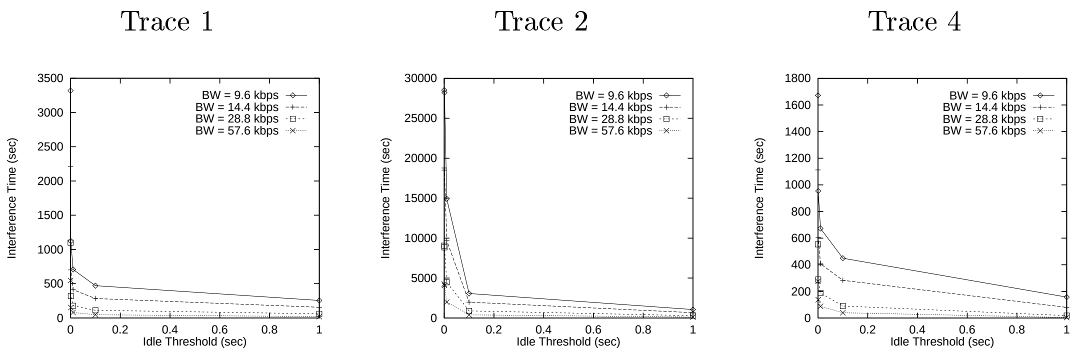

Figure 6.1: Effect of I on write interference time

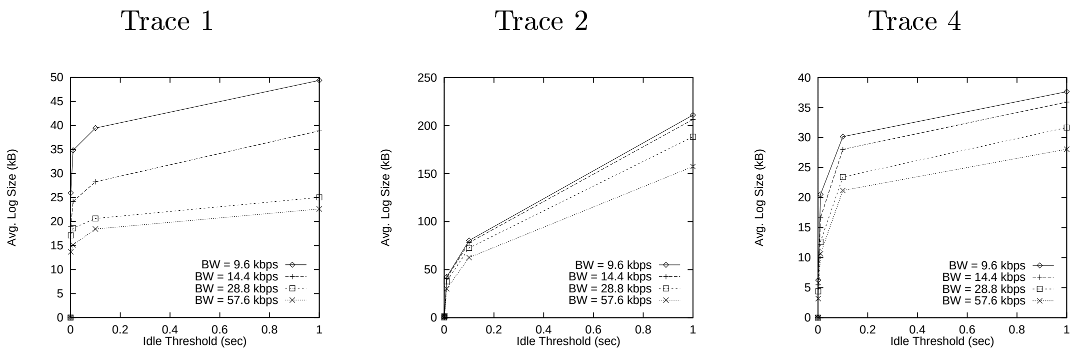

Figure 6.2: Effect of I on log size

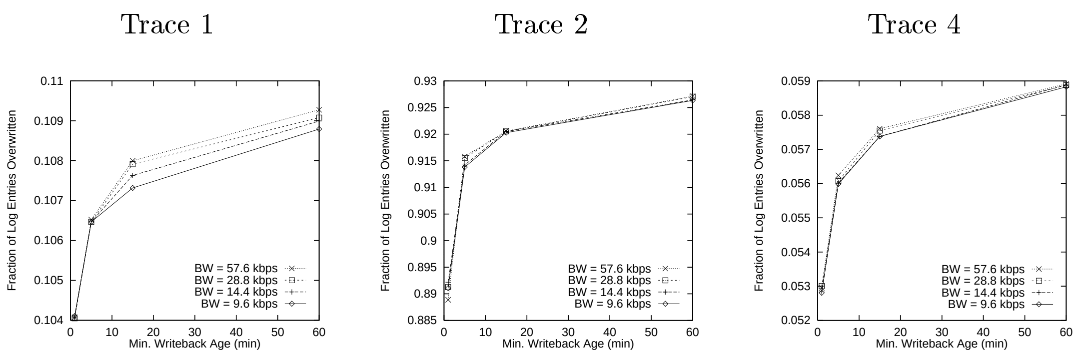

Figure 6.3: Effect of A on log overwrites

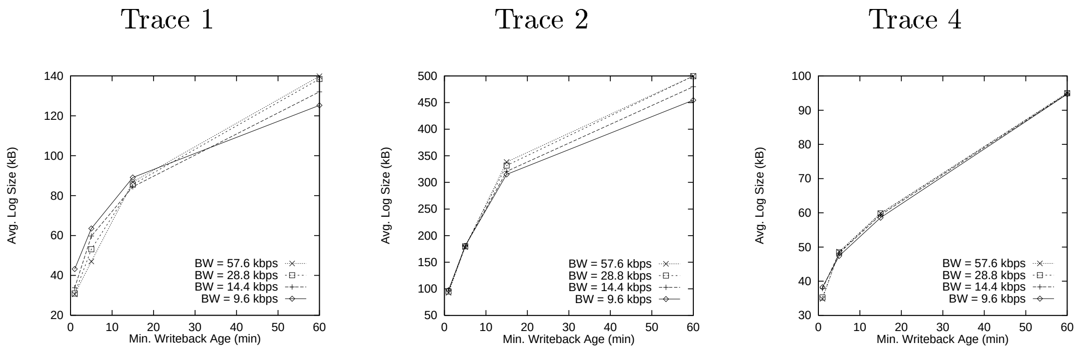

Figure 6.4: Effect of A on log size

this increase slowing only slightly for values of A greater than 15 minutes. Based on these results, A is set to 15 minutes for the remaining experiments. This value results in a large number of log overwrites, with a corresponding increase in log size which is considered reasonable.

6.4.2 Time Expansion

While time expansion is affected primarily by cache size and bandwidth, a cache's write-back policy can also have an influence on the amount of time required to process a stream of file requests. Figures 6.5, 6.6, and 6.7 show the time expansion for Traces 1, 2, and 4. The large impact of cache size on expansion time is clear, especially between caches of 1 MB and 10 MB. The impact of increased bandwidth can also be seen. When bandwidth is doubled, the time expansion drops by nearly half. The impact of write-back policy on time expansion is small in most cases. However, the more writes that are performed, the larger the impact of write-back policy on time expansion. This can be seen in Trace 2, which has a very large fraction of write events – roughly 68% (see Table 4.5). Here, the No Write Backs and Preemptive Write Back policies can be seen to experience the smallest time expansion. In both these policies, reads are never required to wait for network access due to write traffic. The Delayed Write Back policy results in slightly longer run time, while the more aggressive policies (Idle Write Back and Lotto) result in

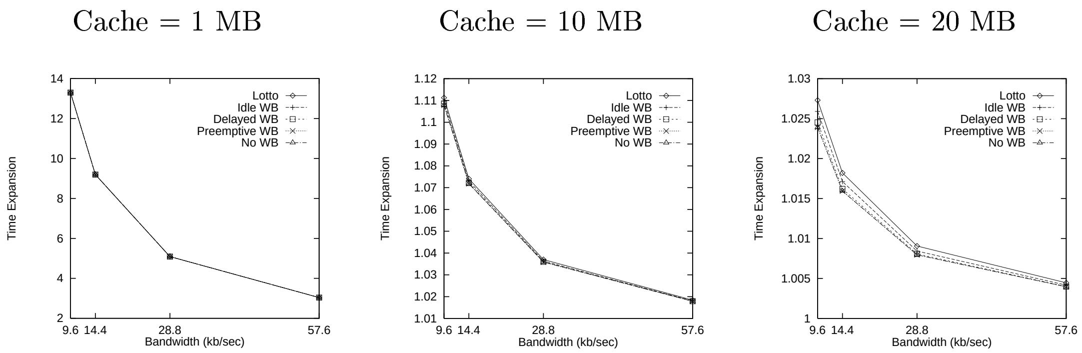

Figure 6.5: Time Expansion – Trace 1

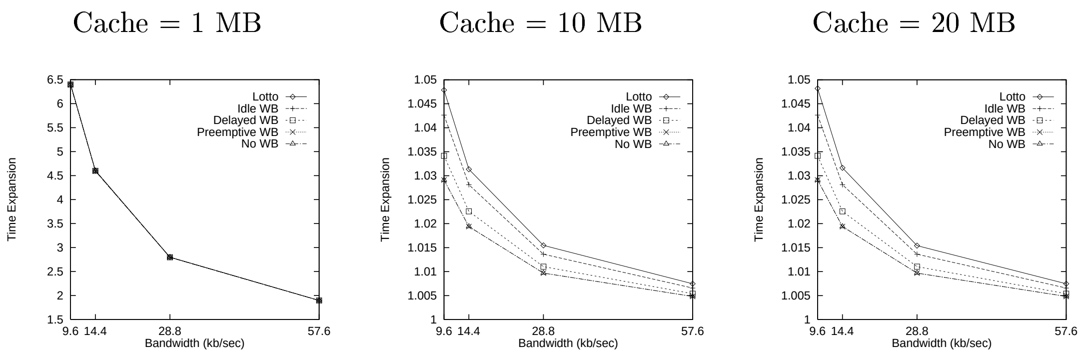

Figure 6.6: Time Expansion – Trace 2

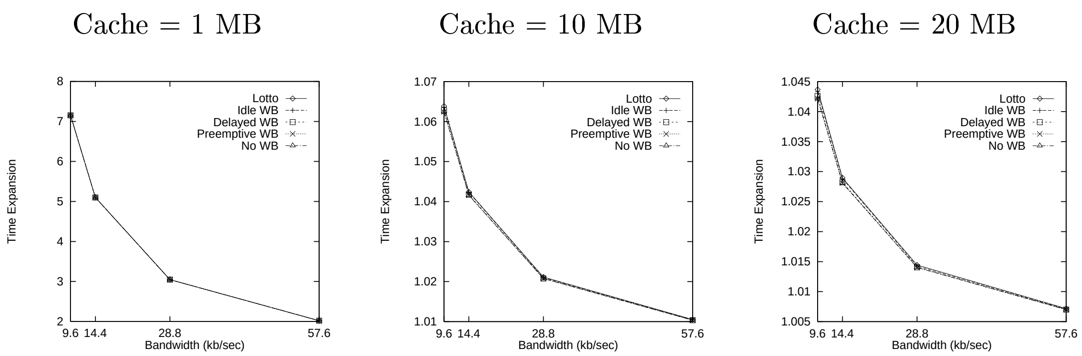

Figure 6.7: Time Expansion – Trace 4

the largest time expansions. As expected, the write-back policies which perform the most write backs also experience the largest time expansion.

6.4.3 Write Interference Time

The fraction of total run time attributable to write interference can be seen in Figures 6.8, 6.9, and 6.10. Interference is significantly reduced by increasing bandwidth. Since all transfers require less time at higher bandwidths, any write backs which interfere with reads will finish more quickly, thus reducing interference time. Cache size has only a small effect on write interference time. The smaller values seen with a 1 MB cache are the result of the large increase in I/O time experienced with a small cache, reducing the relative impact of write interference on total run time.

There is a significant difference in the write interference times experienced by each write-back policy. The Preemptive Write Back policy by design experiences no write interference, while the Delayed policy creates very little interference. The Idle Write Back and Lotto policies, being more aggressive, result in the largest amount of write interference.

Even for the aggressive write-back policies, the amount of write interference, as a fraction of total time, is quite small. This is a good indication that all policies are largely successful at performing write backs only when the network is otherwise idle.

6.4.4 Bytes Written Back

The number of bytes written back by each policy is shown in Figures 6.11, 6.12, and 6.13. All other factors being equal, the more bytes written back to the file server, the less chance there is of update conflicts occurring. The number of bytes written back is almost solely a function of the write back policy in use. Cache size has almost no impact, while the effect of bandwidth is relatively minor in most cases. The small decreases in the number of bytes written back as bandwidth increases is due to the fact that higher bandwidths allow more read traffic to occur, and this may result in

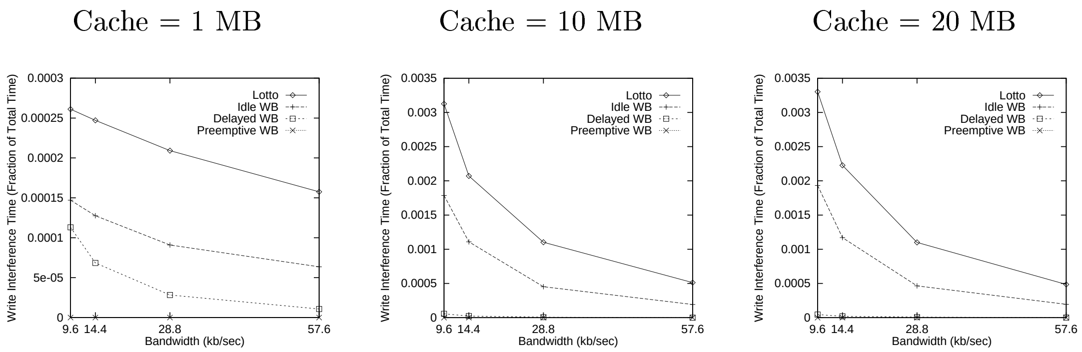

Figure 6.8: Write interference time – Trace 1

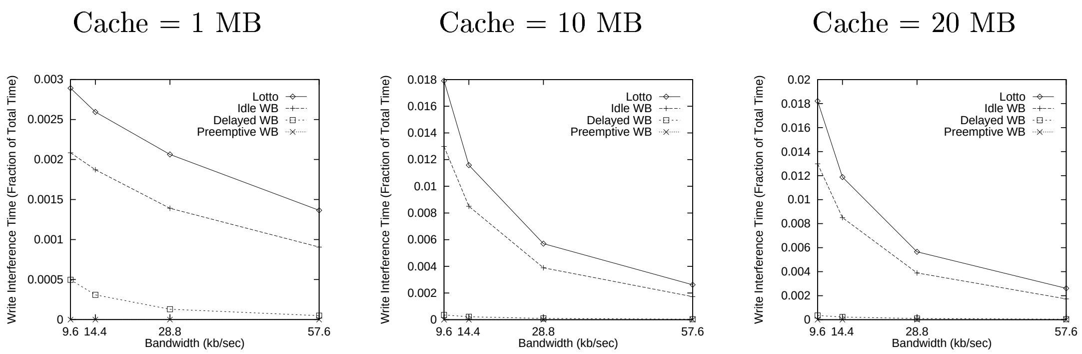

Figure 6.9: Write interference time – Trace 2

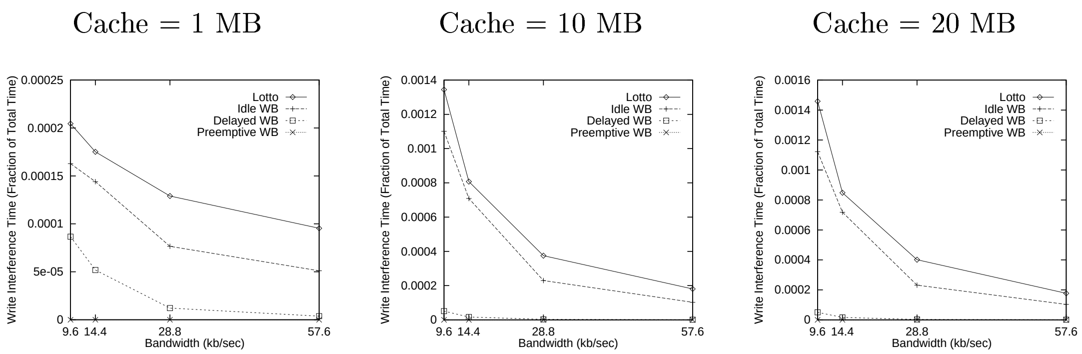

Figure 6.10: Write interference time – Trace 4

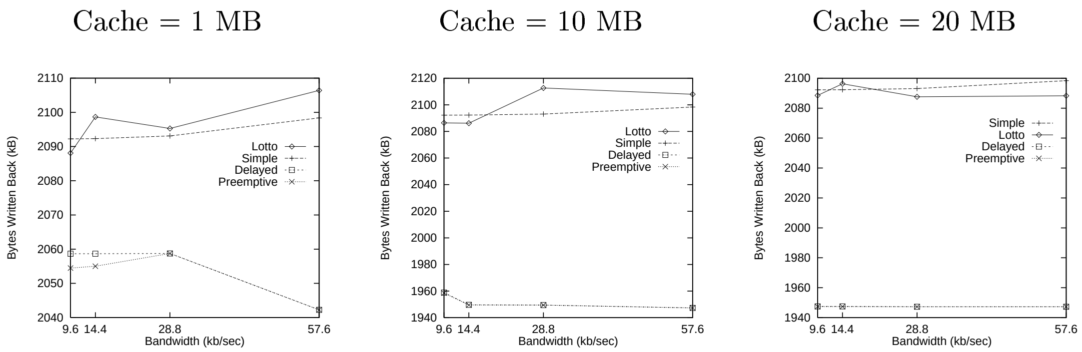

Figure 6.11: Bytes written back – Trace 1

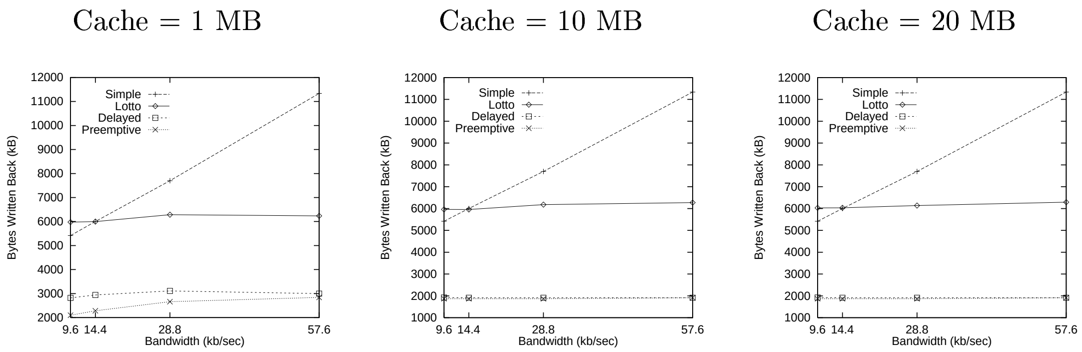

Figure 6.12: Bytes written back – Trace 2

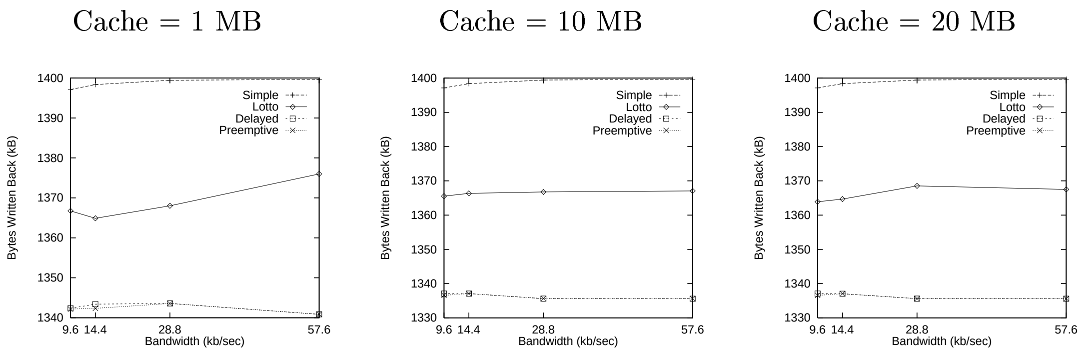

Figure 6.13: Bytes written back – Trace 4

fewer opportunities for write backs to be performed.

The Preemptive and Delayed write-back policies perform almost identically, writing back fewer bytes than other policies. The Idle policy generally writes back the largest number of bytes. This is interesting since the Idle policy has lower write interference times than the Lotto policy. This indicates that the Idle policy makes better use of idle network time than the Lotto policy. For all other policies, the more bytes written back, the larger the write interference time experienced by the policy.

6.4.5 Log Size

There is a strong relationship between the write-back policy used by a cache and the size of the log file. The average size of the client's log file while weakly connected is shown in Figures 6.14, 6.15, and 6.16. The size of the log file is not greatly affected by bandwidth or cache size for most write-back policies. The log size of the No Write Backs policy is provided to show the extent to which each write-back policy reduces the size of the log.

As is expected, the Delayed and Preemptive write-back policies have the largest average log sizes, since these policies generally perform fewer write backs. Even these conservative polices, however, have average log sizes which are, in the worst case, only half to two-thirds the size of the log when no write backs are performed – much smaller in other cases. The Idle Write Back and Lotto policies have even smaller logs, on average about one-sixth to one-tenth the size of that for the No Write Backs policy.

Whereas the average log size shows the behaviour of individual write-back policies, the maximum log size shows the actual amount of disk space the client must have available. Figures 6.17, 6.18, and 6.19 show the maximum size reached by the log file. In all cases, the maximum size of the log file is under 2 MB, which is surprising small given that up to 35 MB of write activity are present in the traces. Maximum log sizes are significantly larger than average sizes, however. The maxi-

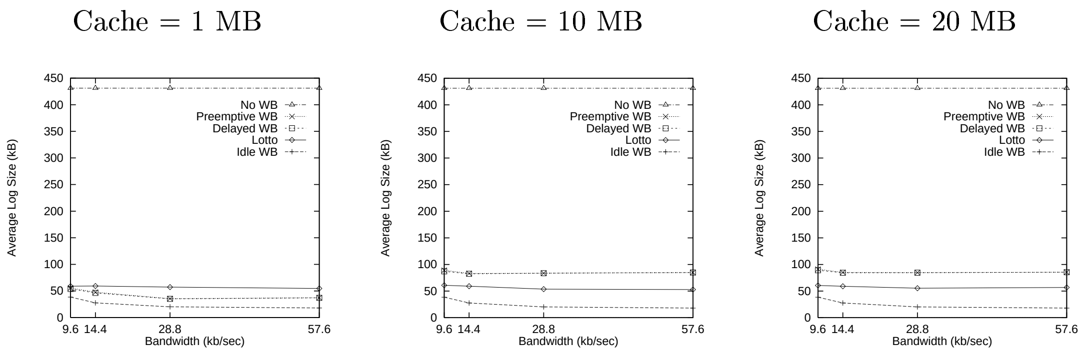

Figure 6.14: Average log size – Trace 1

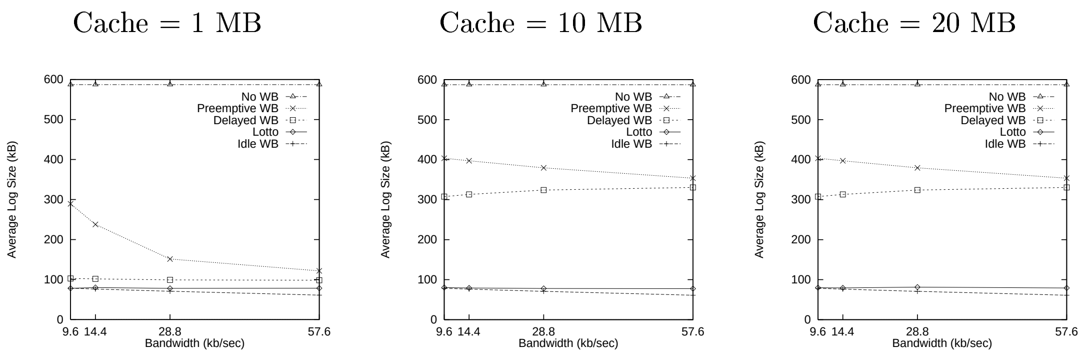

Figure 6.15: Average log size – Trace 2

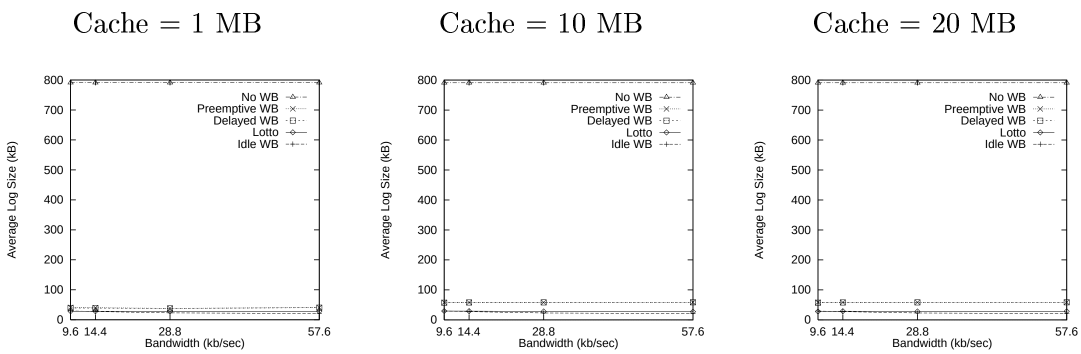

Figure 6.16: Average log size – Trace 4

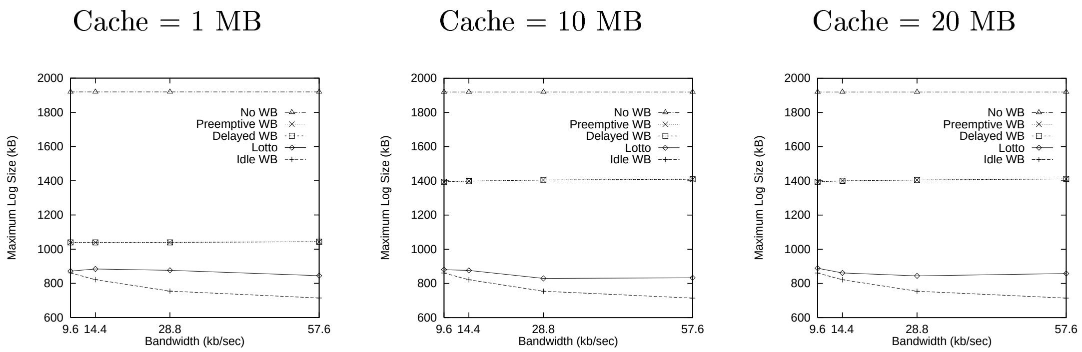

Figure 6.17: Maximum log size – Trace 1

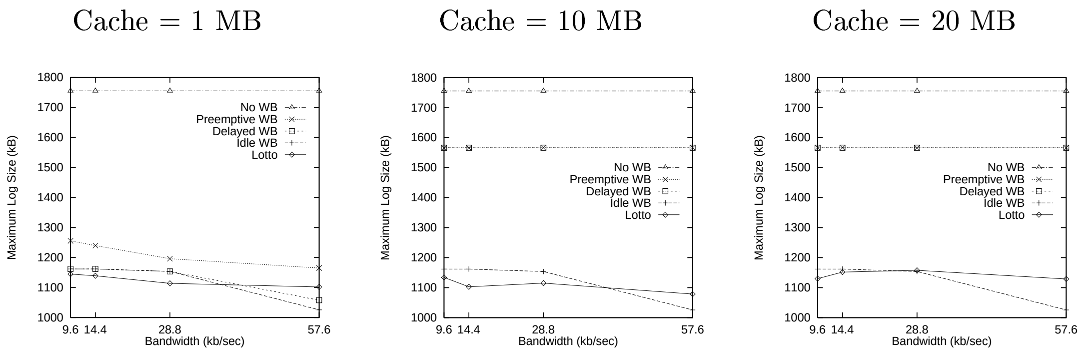

Figure 6.18: Maximum log size – Trace 2

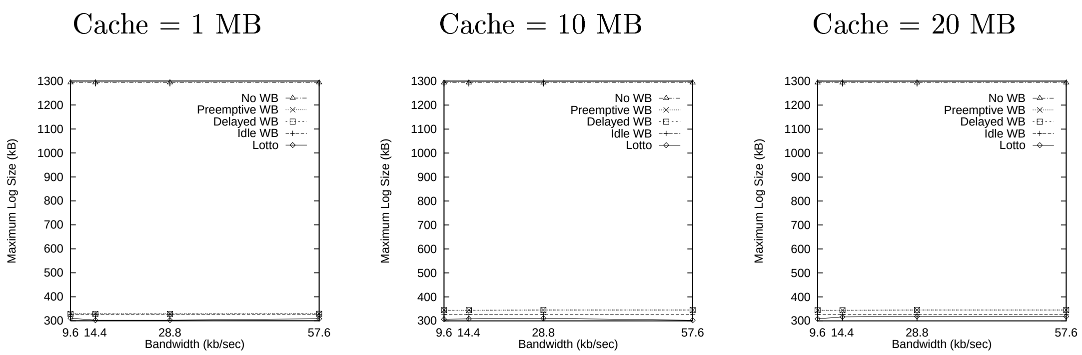

Figure 6.19: Maximum log size – Trace 4

mum log size ranges from roughly one-and-a-half to over four times larger than the average log size when no write backs are performed, and up to nearly fifteen times larger when write backs are performed.

6.4.6 Log Overwrites

The small log sizes observed for all policies are due to the large number of log entries which are overwritten while in the log, as seen in Figures 6.20, 6.21, and 6.22. For the traces examined, from 35 to 90 percent of all log entries are overwritten before they are written back to the file server. The high number of log overwrites for Trace 2 is likely due to the large percentage of write requests present in that trace.

A difference is seen again between the performance of the aggressive and conservative policies. The aging parameter in the Preemptive and Delayed policies is successful in keeping entries in the log longer, ensuring that all log entries have a chance to be overwritten by subsequent update activity. The Lotto and Idle Write Back policies experience roughly five to 25 percent fewer log overwrites. The drop in log overwrites for the Idle policy at high bandwidths for Trace 2 corresponds to the increased number of bytes written back which occurs in this configuration (see Figure 6.12).

6.4.7 Time In Log

Figures 6.23, 6.24, and 6.25 show the average period of time log entries are present in the log file. The time in log is significantly reduced as bandwidth increases, since it is possible to perform more write backs. Increasing cache size also reduces the average time spent in the log, since less bandwidth is needed to service read misses.

As is expected, use of an aggressive write-back policy results in log entries being reintegrated much more quickly than with a conservative write-back policy. With a client file cache of 10 MB or more, the average time a log entry remains in the log is quite short even with a conservative write-back policy, ranging from 15 to 35 minutes (depending on network bandwidth). For more aggressive policies, this time

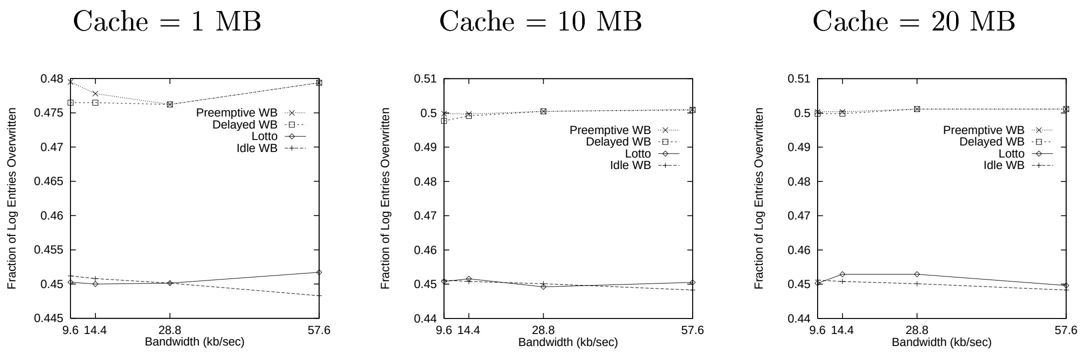

Figure 6.20: Log Overwrites – Trace 1

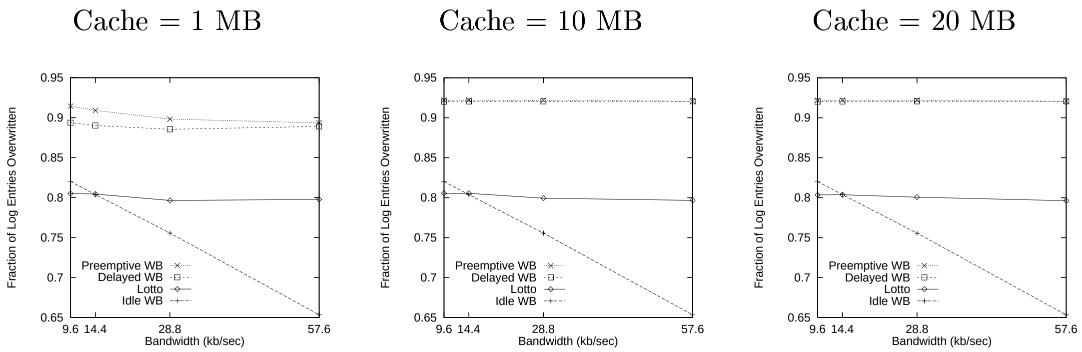

Figure 6.21: Log Overwrites – Trace 2

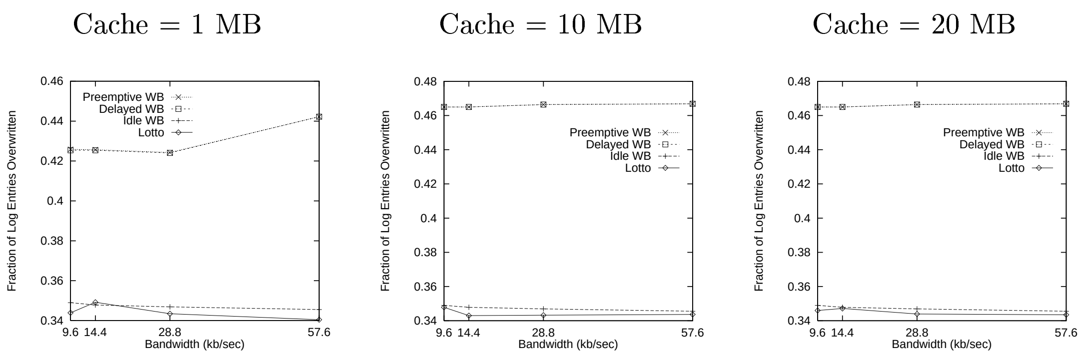

Figure 6.22: Log Overwrites – Trace 4

Cache = 1 MB   Cache = 10 MB   Cache = 20 MB

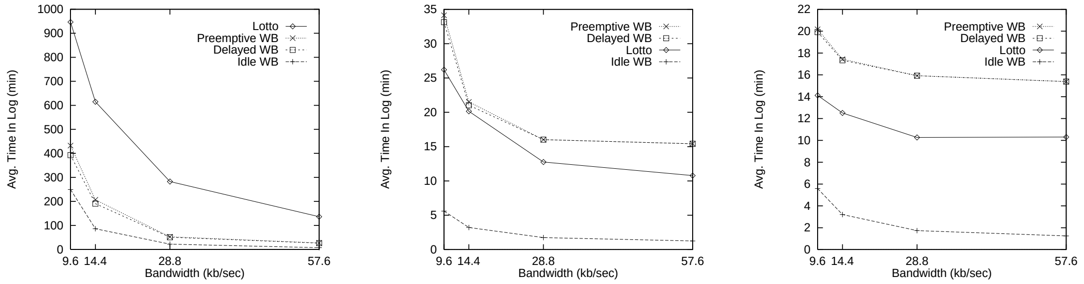

Figure 6.23: Average time log entries remain in log file – Trace 1

Cache = 1 MB   Cache = 10 MB   Cache = 20 MB

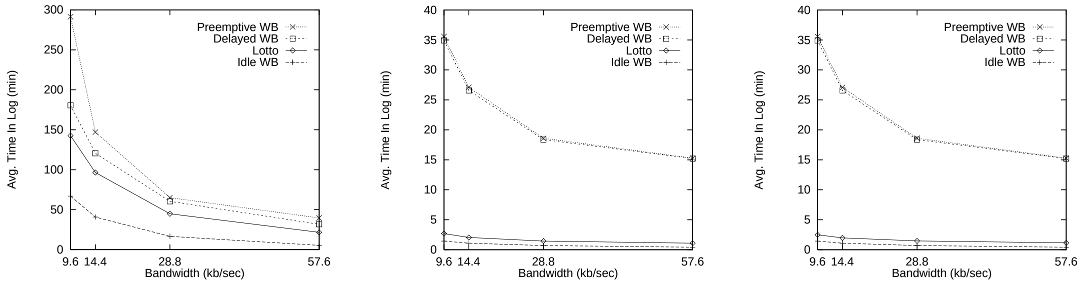

Figure 6.24: Average time log entries remain in log file – Trace 2

Cache = 1 MB   Cache = 10 MB   Cache = 20 MB

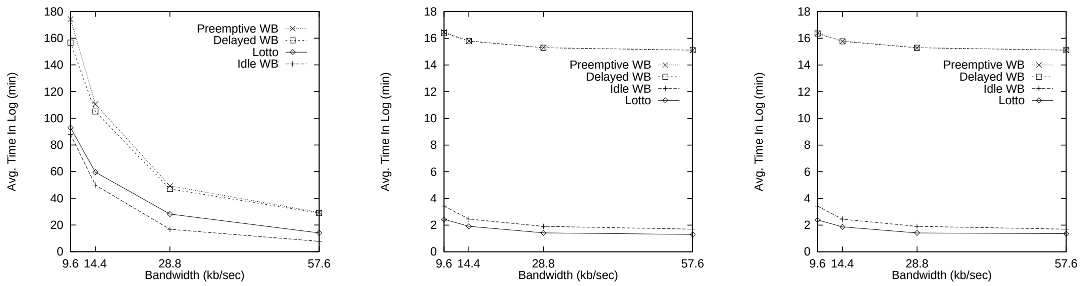

Figure 6.25: Average time log entries remain in log file – Trace 4

is reduced to from one to six minutes. For most situations, reintegrating updates within this time frame would likely drastically reduce the chance of update conflicts occurring.

6.5 Conclusions

The goal of this study was to examine the performance and resource implications of sharing a low-bandwidth connection between read and write-back traffic in a mobile environment. Five write-back policies were examined: No Write Backs, Idle Write Back, Delayed Write Back, Preemptive Write Back, and Lottery.

The simulation results show that it is possible to provide good file system support in a weakly-connected environment, by servicing read requests in a timely manner while at the same time providing acceptable write service. Even at low bandwidths, there is enough idle network time that even a conservative write-back policy, such as Delayed Write Back, results in writes being reintegrated with the home file system in an acceptable period of time – less than 30 minutes in most cases. This is accomplished with very little interference with read traffic, increasing the total time required to perform all read operations by less than 2%. This level of performance can be achieved even at a resource-limited client. For the traces examined in this study, a 10 MB file cache was found to be large enough to support this type of activity. While there are definitely performance benefits to be gained from higher-bandwidth connections, even a 9.6 kb/sec network link proved to be enough to provide acceptable levels of performance.

The performance benefits of delaying write backs were found to be quite substantial. By ensuring that log entries remain in the log for even 15 minutes, the number of entries in the log can be reduced dramatically by overlapping writes, in turn reducing significantly the number of write backs which need to be performed.

Interestingly, the Preemptive Write Back policy, despite its greater complexity, did not perform significantly better than the Delayed Write Back policy in most cases. (Although it would have performed slightly better if no Idle Threshold had been enforced.) This is encouraging since it shows that it is not necessary for write back policies to be complex in order to achieve acceptable levels of performance.

Chapter 7

Caching Unit

When operating in a disconnected environment, whole-file caching is the only practical approach – having only part of file cached is likely of little value if the rest of the file can not be accessed. This is not necessarily true in a weakly-connected environment, however. In this respect, a weakly-connected environment is similar to a strongly-connected environment. With even a low-bandwidth network connection block caching is an option.

Two features of block caching make it attractive for use in weakly-connected operation. First, 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 at low bandwidths 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. Again, this is very useful in a low-bandwidth situation, since it allows more bandwidth to remain free for other read or write-back activity.

A set of experiments was performed to investigate the performance impact of using block caching in a weakly-connected environment. These experiments use the same format as was used for the write-back policy experiments, with a 24-hour hoarding period followed by a period of weakly-connected operation (see Chapter 6). Hoarding is still performed on a whole-file basis, but block caching is used once weakly-connected.

This chapter presents the simulation results obtained when block caching is used.

These results are separated into two sections: in the first, the effect of different block sizes is examined; in the second, cache performance when using blocks is compared to that when whole files are cached.

7.1 Experimental Design

The experiments performed for this study follow the same basic design as the write-back policy experiments, as shown in Table 7.1. The U of S seven-day traces are used, and LRU (whole-file) hoarding is performed for a 24-hour period at the start of the simulation. The primary difference is that when the client becomes weakly connected, block caching is initiated, with all further reads being performed in fix-sized blocks. (Note that block fragmentation is not modelled, so even the last block of a file will be the same size as other blocks. This means that for this study the limit of increasing block size is not the same as whole-file caching.) Block sizes of 0.5 to 8 kB are used, and LRU replacement is performed. The same range of bandwidths is used (9.6 to 57.6 kb/sec), with the same write-back policies. Slightly smaller caches are used, however, ranging from 1 to 10 MB. The same values are used for the write-back policy parameters (0.1 seconds for I and 15 minutes for A).

7.2 Block Caching Performance

The performance of the cache when block caching is used depends on the size of blocks used. Experiments were performed across a range of block sizes to quantify the effect of block size on cache performance. The results presented are for Traces 1, 2 and 4 of the U of S seven-day traces. A cache size of 10 MB is used, and bandwidth is fixed at 28.8 kb/sec.

Figure 7.1 shows the time expansion experienced by each write-back policy for a range of block sizes. The performance of the write-back policies with respect to each other is similar to when whole-file caching is used, with the aggressive policies experiencing greater time expansion than the conservative policies. As block size

Table 7.1: Factors and levels used in caching unit experiments

Factor	Levels
Trace Files	U of S seven-day
Hoarding Policy	LRU
Hoarding Period	1 day
Replacement Policy	LRU
Bandwidth	28.8 kb/sec
Cache Size	1, 5, 10 (MB)
Block Size	0.5, 1, 2, 4, 8 (kB)
Write Back Policy	No Write Back, Idle Write Back, Delayed Write Back Preempitve Write Back, Lotto
Dependant Variables	time expansion, write interference time, bytes written back, time in log, log size, log overwrites, hit rate

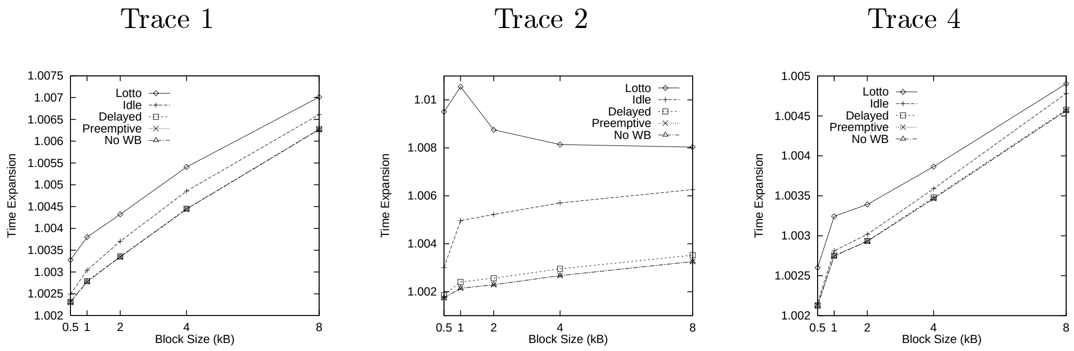

Figure 7.1: Time Expansion

is increased, time expansion also increases. This behaviour is expected since the smaller the block size, the less time is "wasted" by transferring portions of a file which are never referenced by the client.

The number of write backs initiated by any of the policies depends on the time between read requests. As block size is increased, read transfers will take longer, but the hit rate of the cache will also increase (see Figure 7.8), decreasing the number of read transfers performed. This makes the impact of block size on the number of bytes written back extremely hard to predict, as different workloads will be affected differently by these factors. This can been seen in Figure 7.2. While the aggressive policies write back more bytes than the conservative policies, the effect of block size

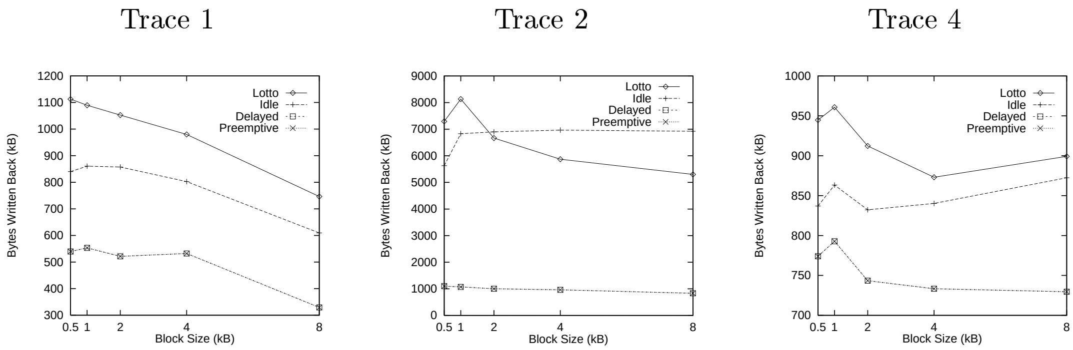

Figure 7.2: Bytes written back

on bytes written back is indeed sensitive to the workload. For Trace 1, each policy experiences a decrease in bytes written back as block size increases. For Trace 2, block size does not have a large impact on the number of bytes written back, except when the Lotto policy is used. In this case, bytes written back again drops as block size increases. For Trace 4, each policy experiences both increases and decreases in the number of bytes written back as block size increases.

The effect of block size on write interference time is shown in Figure 7.3. The conservative write-back policies experience almost no write interference. For the aggressive policies, the amount of interference time generally parallels the number of bytes written back for a given block size. When the Idle Write Back policy is used, large read transfers increase the likelihood of interference. The Lotto policy actually experiences *less* interference when block sizes are larger. This is likely because lotteries are held after each read transfer completes, and therefore more lotteries will be held when the block size is small. An increased number of lotteries will result in more write interference, especially if the token distribution is not set appropriately.

Figure 7.4 shows the fraction of log entries overwritten. As block size is increased, the number of log overwrites also increases. This is because the larger the block size, the greater the chance that multiple updates will be made to the same file block. The smaller number of log overwrites for the Idle Write Back policy seen for Trace 2 corresponds to the larger number of bytes written back under this policy

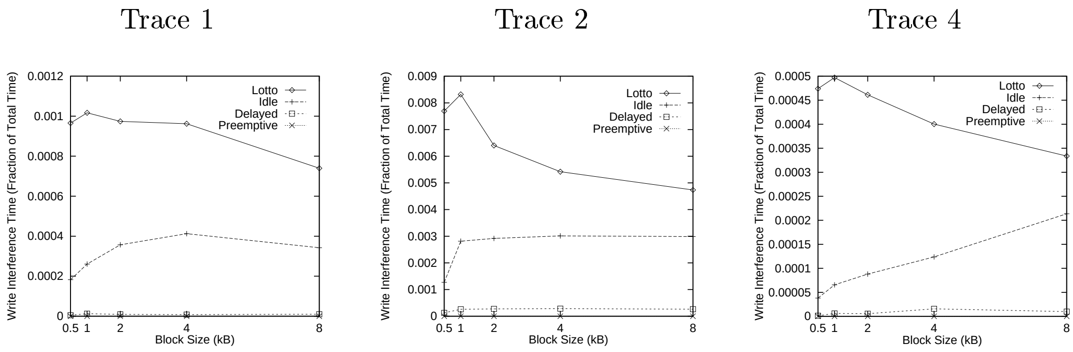

Figure 7.3: Write interference time

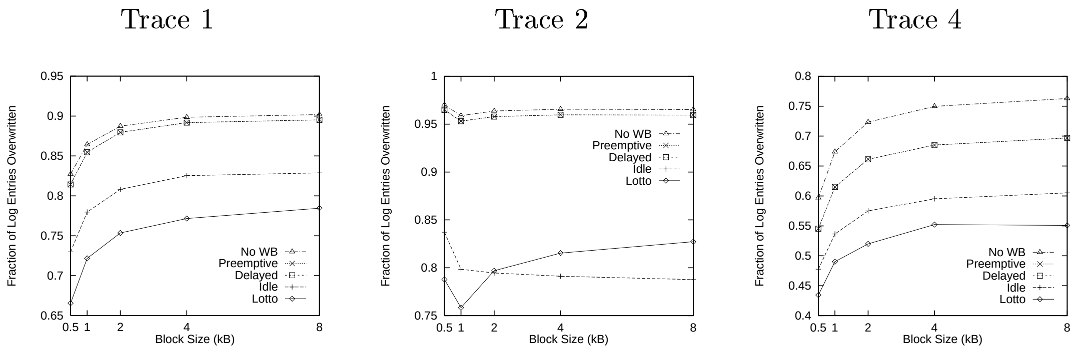

Figure 7.4: Log overwrites

(see Figure 7.2).

Figures 7.5 and 7.6 show the impact of block size on the size of the log file. The impact of write-back policy on log size is very similar to when whole-file caching is used, with the aggressive policies having significantly smaller log sizes than the conservative policies. Block size has little impact on log size when an aggressive write-back policy is used. For the conservative policies, the size of the log (both maximum and average) decreases as block size increases. This is a result of the increased number of log overwrites which occur when larger block sizes are used.

The average time log entries remain in the log file before being written back to the file server is largely unaffected by block size, as seen in Figure 7.7. The same grouping of aggressive and conservative policies is seen as when whole-file caching is performed.

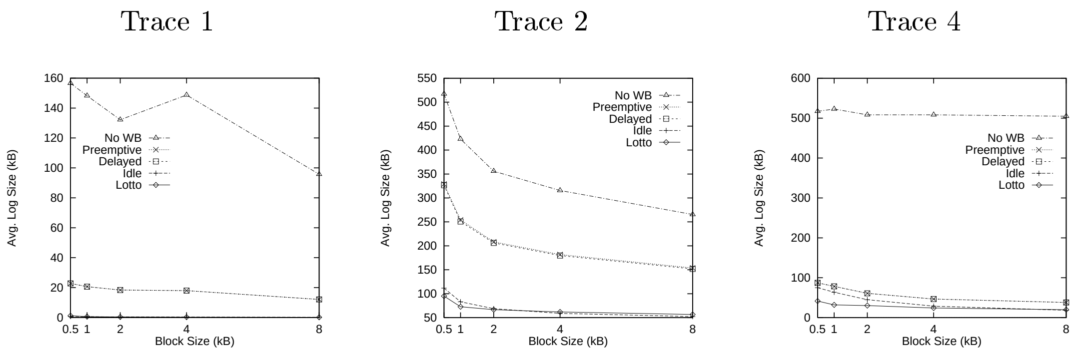

Figure 7.5: Average log size

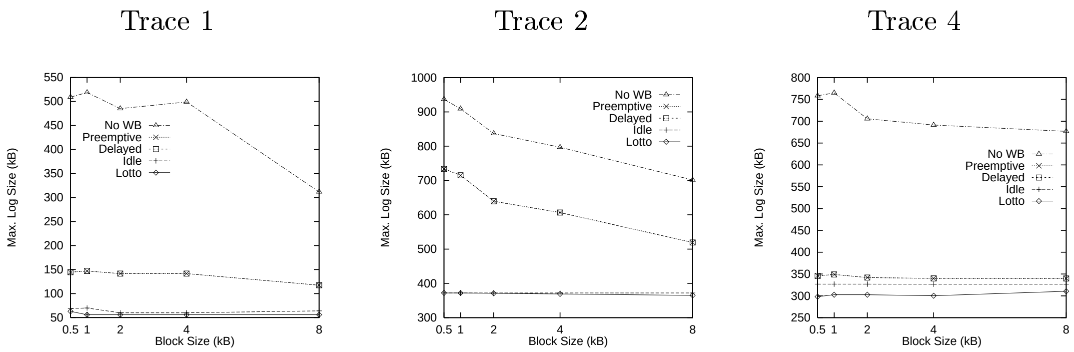

Figure 7.6: Maximum log size

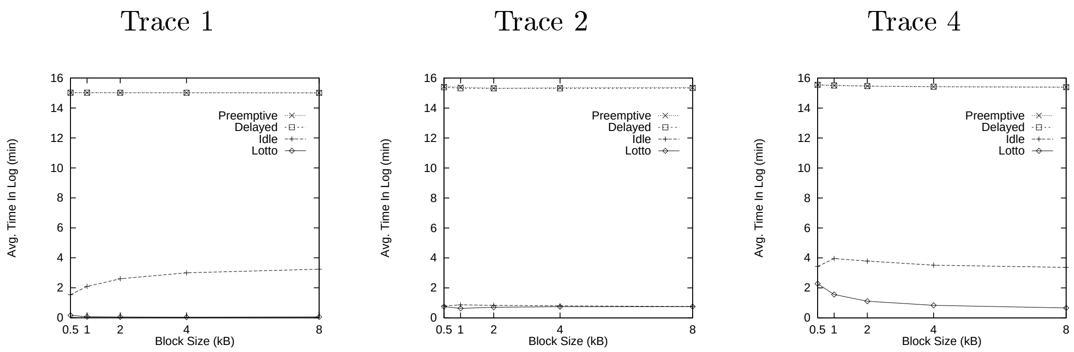

Figure 7.7: Average time log entries remain in log file

The use of block caching means that miss (or hit) rate may once again becomes a reasonable metric of file cache performance, since each miss results in the same number of bytes transfered from the server to the client. Figure 7.8 shows the hit rate at the cache for a range of cache sizes. Larger hit rates are achieved with the use of larger block sizes, across cache sizes. The high hit rate observed for Trace 2 with a block size of 0.5 kb and a 0.1 kB cache is consistent with the results reported in [47], where small blocks resulted in higher hit rates for small caches, and larger blocks produced higher hit rates in large caches. For each trace, very high hit rates (usually over 85%) are achieved when a cache of 1 MB or larger is used.

The hit rate results, taken by themselves, suggest that larger block sizes are desirable. However, as seen previously, time expansion, the metric of most concern to a user, is reduced when small blocks are used. Thus, even when block caching is used, hit rate can be deceiving as a metric of mobile client file cache performance if it is not used in conjunction with other metrics.

Relative to each other, the write-back policies examined perform quite similarly when either block or whole-file caching is used. The Lotto and Idle Write Back policies perform write backs aggressively, resulting in more bytes written back, but with a corresponding increase in interference time compared to conservative policies.

Several trade-offs must be considered when choosing a block size for a client file cache. Small blocks offer a reduction in the total time required to service file requests, at the cost of increased log size, and more network transfers. Large blocks increase total file transfer time, but have the advantage of reducing the size of the log file and performing fewer network operations. Performing a large number of short transfers, as is done when block sizes are small, may not be desirable if there is any significant amount of overhead (or other cost) associated with initiating a transfer.

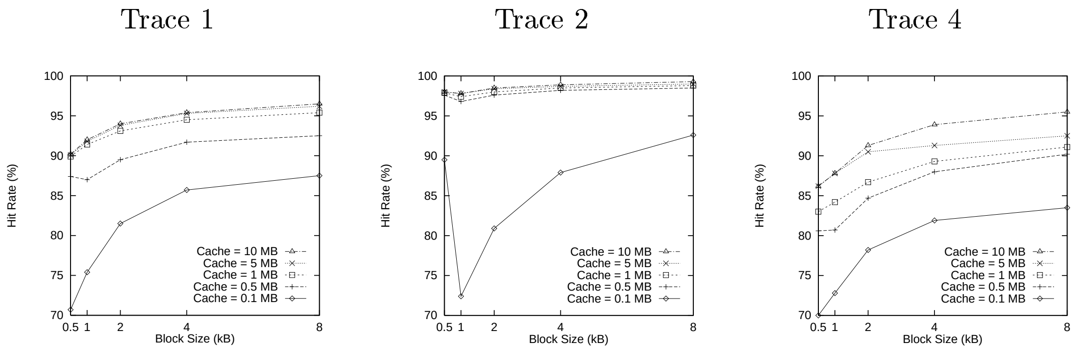

Figure 7.8: Hit Rate

7.3 Block Caching Versus File Caching

Since block caching appears to offer some potential performance benefits for weakly-connected operation, a comparison of block caching and file caching was performed. A client cache of 10 MB was used, with network connections ranging in speed from 9.6 to 57.6 kb/sec. When block caching is performed, 4 kB blocks are used. Three write-back policies are considered: No Write Backs, Idle Write Back, and Delayed Write Back.

Figure 7.9 compares the time expansion experienced by block and whole-file caches. The use of block caching has a larger effect on time expansion than the choice of write-back policy. In each trace, whole-file caching resulted in greater time expansion than block caching, regardless of the write-back policy in use. At greater bandwidths, however, less difference is seen. With a 9.6 kb/sec network connection, block caching results in a 2-10% reduction in time expansion, depending on the trace. When the bandwidth is 57.6 kb/sec, this reduction is only 1-2%.

The effect of block caching on write interference time is shown in Figure 7.10. Given the same write-back policy, in most cases block caching results in less write interference time – a reduction of 15-25% is seen in many cases. As with time expansion, the differences between block and whole-file caching are greatest when bandwidth is low. However, the write-back policy still has a greater effect on write interference time than caching unit. That is, use of an aggressive write-back policy

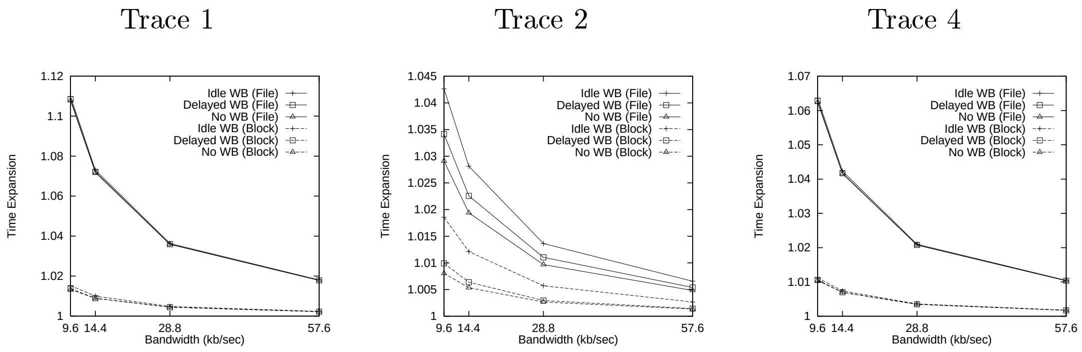

Figure 7.9: Time Expansion – block vs. file caching

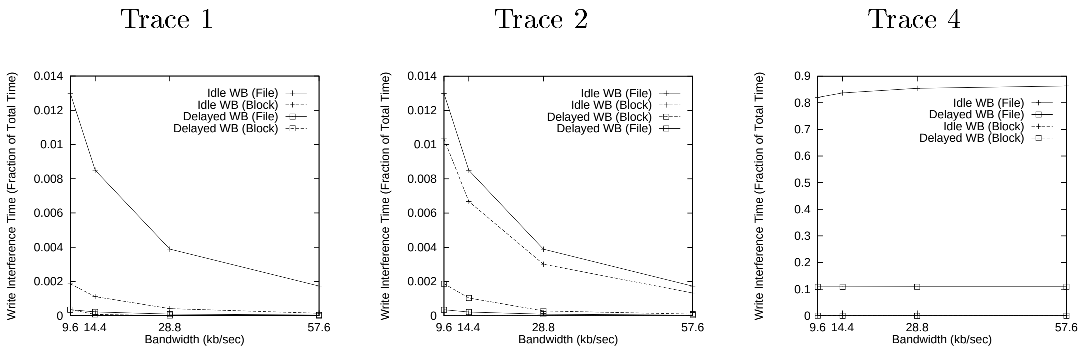

Figure 7.10: Write interference time – block vs. file caching

and block caching results in more write interference than use of a conservative policy and whole-file caching.

Figure 7.11 shows the number of bytes written back for block and whole-file caching. More bytes are written back when whole-file caching is used – over 50% more in many cases. When there are a large number of writes, as for Trace 2, write-back policy has a greater effect on the number of bytes written back than caching unit. When there are fewer writes (Traces 1 and 4), the effect of block caching overshadows the effect of write-back policy.

The effect of caching unit on the size of the log file is shown in Figures 7.12 and 7.13. In terms of average log size, write-back policy has a larger effect on the size of the log than caching unit. The use of block caching does result in a significant reduction in log size, however. When no write backs are performed, the average

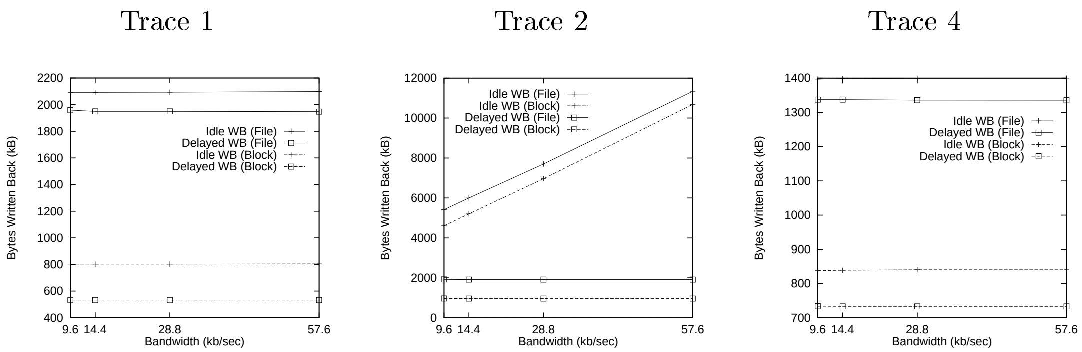

Figure 7.11: Bytes written back – block vs. file caching

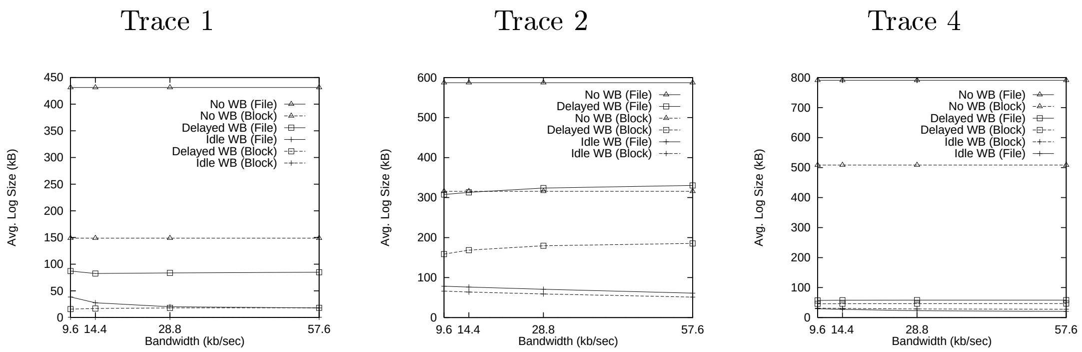

Figure 7.12: Average log size – block vs. file caching

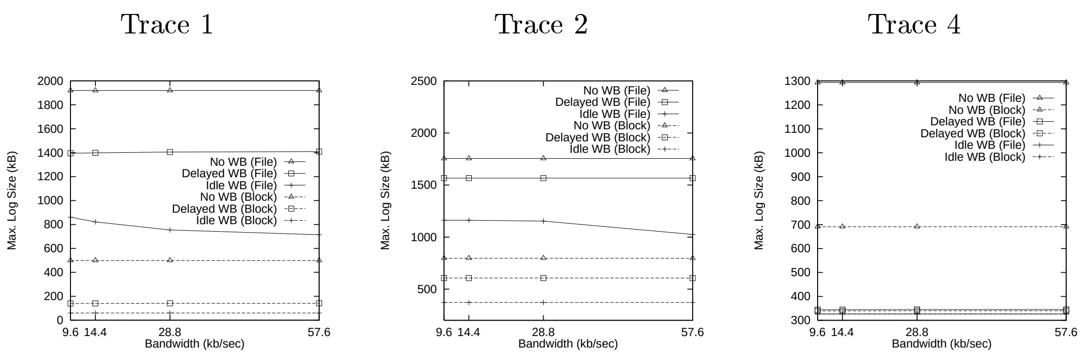

Figure 7.13: Maximum log size – block vs. file caching

log size is 35-60% smaller than when whole-file caching is used. The effect of block caching on the maximum size of the log is even more dramatic. When no write backs are performed, the use of block caching results in the maximum size of the log being reduced by two to four times. Similar decreases in maximum log size are seen even when write backs are performed.

Figure 7.14 shows the number of log overwrites that occur. The results are similar to the number of bytes written back, with block caching having a greater effect for the traces with fewer writes. Block caching results in fewer bytes written back (see Figure 7.11), which in turn results in more log overwrites occurring. The increase in log overwrites, ranging from 5-40% when no write backs are performed, is smaller than the decrease in bytes written back.

Finally, the amount of time log entries remain in the log file is shown in Figure 7.15. The choice of caching unit has a smaller effect on the time entries remain in the log than on other metrics. This is especially true at higher bandwidths. The major difference in performance is seen for the Delayed Write Back policy. When whole-file caching is used, entries remain in the log much longer when bandwidth is low. When block caching is used, bandwidth has very little effect on the time entries remain in the log. This is likely because less time is required for read transfers when block caching is used, and when a conservative write-back policy is used, this extra idle network time can result in more write backs being performed. This effect is greater at low bandwidths since this is where block caching has the largest effect on the time required to service read traffic.

7.4 Conclusions

The presence of a weak network connection presents a mobile client's file cache with the opportunity to use block-based caching, as opposed to whole-file caching which is necessary in a disconnected environment. The goal of this study was to investigate the performance and resource implications of using block caching in a weakly-connected environment, and to compare this performance to that observed

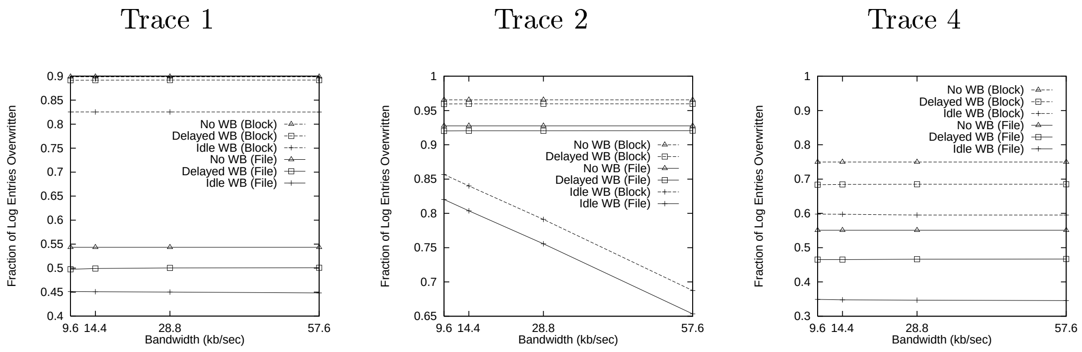

Figure 7.14: Log overwrites – block vs. file caching

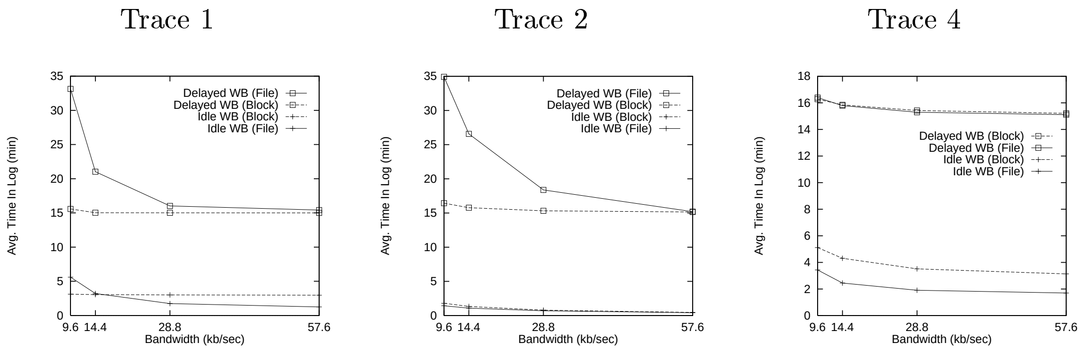

Figure 7.15: Average time log entries remain in log file – block vs. file caching

when whole-file caching is used.

A study of the effect of block size on client file cache performance showed that several performance/resource trade-offs exist with respect to block size. The use of small blocks reduces the time required to service read requests, but this reduction comes at the cost of increased log size, and an increase in the number of file transfers performed. There were several cases, however, where the performance impact of block size was significantly different across various workloads. Examination of a larger set of workloads would help provide a better understanding of the effect of block size on cache performance in these cases.

A comparison of cache performance when block and whole-file caching is used was also performed. Several advantages to block caching are apparent from this study. The time expansion, write interference time, log size, and time log entries remain in the log are all reduced when block caching is used. These advantages come at the cost of fewer bytes written back to the file server, but this could likely be changed by "tuning" write-back policies for better performance when block-caching is used – something which was not done in this study.

Although brief, the results of this study indicate that there are definite advantages to performing block caching in a weakly-connected environment. Although whole-file caching is required to support disconnected operation, designing a cache manager to adapt to a weak connection by switching to block caching when possible could prove beneficial. It could also be possible to allow the cache manager to change block size as bandwidth changes. Further investigation would be required to examine issues such as how to cope with returning to a disconnected, whole-file mode of operation after performing block caching while weakly-connected.

Chapter 8

Summary and Conclusions

This thesis has presented a performance study of techniques for managing a mobile client's file cache. This chapter summarises the work that was done, discusses the contributions made by this work, and describes areas for future research.

8.1 Summary

Trace-driven simulation was used to investigate the performance of three aspects of mobile client file cache management – hoarding, write backs, and caching unit. The traces of user file activity used for this study were obtained from the Computer Science Departments of the University of Saskatchewan and Carnegie Mellon University. These traces contained records of file activity over durations of 24-hours and seven-days.

In order to evaluate mobile client file cache performance, it was first necessary to identify factors which affect cache performance in a mobile environment, then choose metrics which could be used to measure cache performance. This was necessary since miss rate, the metric traditionally used to measure cache performance, is usually not a good metric in a mobile environment. Nine metrics were chosen to be used in this study:

- bytes missed
- time until first miss
- hit rate
- time expansion
- write interference time
- bytes written back
- log size
- time in log
- log overwrites

The first part of the simulation study was an investigation of the impact of hoarding policy on client file cache performance in a disconnected environment. Five hoarding policies were examined: RAND, LRU, LFU, Keep Large, and Keep Small. Results from this study showed that workload and cache size each had a very large impact on disconnected cache performance. However, for the workloads examined, caches on the order of 20 MB provided sufficiently good performance to provide a basis for weakly-connected operation. The LRU hoarding policy was found to provide the most consistent performance across various workloads and cache sizes. The inappropriateness of miss rate as metric of mobile client file cache performance was also confirmed.

Overall, the ability of the hoarding policies examined to support disconnected operation was found to be inadequate, with the time until the first miss being less than one hour in most cases. Hoarding policies which do not rely simply on short-term user reference patterns are needed to properly support disconnected operation.

The second part of the study examined the issue of how to schedule reads and writes in a weakly-connected environment. Five write-back policies were used: No Write Backs, Idle Write Back, Delayed Write Back, Preemptive Write Back, and Lotto. Results show that it is possible to provide acceptable levels of write service while still servicing reads in a timely manner. Even with a low-bandwidth network connection, there is enough idle network time that log entries can be reintegrated with the home file system, usually in less than 30 minutes. Delaying write backs by as little as 15 minutes was also found to drastically reduce the number of write backs which need to be performed.

The final issue investigated was the choice of caching unit for use in a weakly-connected file cache. Two alternatives were examined: whole-file caching, and block caching. Whole-file caching is required for disconnected operation, and has been used in previous systems which support mobility. However, when a client possesses even a weak network connection, block caching is possible. Cache performance when various block sizes are used was studied. The use of small blocks was found to reduce the total time required to perform read transfers, at the cost of a larger log file, and a greater number of individual transfers. A comparison of block and whole-file caching showed that definite performance benefits can be obtained through the use of block caching. Time expansion, write interference time, log size, and the time updates spend in the log were all smaller when block caching was used than when whole-file caching was performed.

File access is an important requirement of a mobile computing environment. The presence of a weak network connection provides a mobile client with an opportunity to have the same file access functionality as a strongly connected client. Although such functionality comes with an attendant performance cost, this study suggests that these costs can be kept in check through the use of well-designed cache management strategies.

8.2 Contributions and Relevance

Providing file system support for mobile computing requires that many issues which are taken for granted in a LAN environment need to be re-examined with the constraints of the mobile environment in mind. This work has done this for one specific issue – that of file cache management.

While several file systems have been developed to support mobile computing, the performance of the cache management policies used in these systems has not been examined in detail. To date, functionality rather than performance has been the focus of research efforts. This work has examined the performance of several policies for management of a mobile client's file cache, and has quantified some of the key resource-performance trade-offs which exist. In order to do this, a set of metrics for evaluating mobile client file cache performance were developed, the need for which has been noted in other work [45].

This work also showed the potential for improved weakly-connected cache performance through the use of block caching. Previous work related to mobile file systems initially developed support for disconnected operation. Support for weakly-connected operation was added later, and followed the same basic approach. However, in many respects weakly-connected operation has more in common with strongly-connected operation than disconnected operation, and cache management for weakly-connected operation should be examined with this in mind. The use of block caching is one example of how a weak network connection can be used for more than simply servicing read misses.

8.3 Future Research

There are many avenues for future research into file cache management for mobile computing which follow from this thesis.

First, the analysis done in this study could be extended to cover a wider range of workloads. The use of file reference traces from environments other than academic research labs could provide valuable insight into how sensitive the results of this study are to workload. The use of traces containing read and write activity from sources other than the University of Saskatchewan would be a good first step. Additionally, workloads consisting exclusively of specific types of activity, such as programming, document preparation, or web browsing, could be used. This would allow an investigation into how cache management could be adapted to support specific types of user activity.

A more detailed examination of hoarding policies could prove insightful. Two approaches could be used to improve on the study done in this work. First, a longer-term study of hoarding would provide stronger results than those seen here. This would require traces several times longer than those used in this work. Another approach would be to study the performance of hoarding policies which do not simply rely on user reference patterns to make hoarding decisions. Such a study would require not only file reference traces, but also knowledge of the contents of entire file system, or at least some portion of it. This would allow policies such as those used in the Coda file system to be examined, along with other, more complex policies.

The write-back policies used in this work attempt to detect when the network is idle, and schedule write backs accordingly. Much more sophisticated "idleness" detection algorithms have been developed [13], and these could be the basis of other write-back policies. Such policies could be even more successful at avoiding write interference than those used in this study.

The use of block caching in weakly-connected environments was examined briefly in this work, but a more detailed study is warranted. Issues such as resuming whole-file disconnected operation after working in a block-based weakly-connected environment need to be examined. A more detailed network model would provide stronger results with respect to the performance impact of using block-based caching.

This work focussed on the performance of hoarding policy, write-back policy, and caching unit. There are other aspects of file cache management that could also be studied. One of the most interesting would be file prefetching polices. As with an investigation of more detailed hoarding policies, this would require knowledge of the contents of the file system from which traces were gathered. However, the use of file prefetching in a weakly-connected environment could likely result in significant performance improvements. The difficulty of file prefetching means that a performance study of different policies would be very valuable.

The use of adaptive cache management policies also has the potential to improve cache performance. A study of how various cache management policies could be made to adapt to changes in the environment, such as the speed of network connection or available disk space at the client, could be of interest. For example, the parameters governing the behaviour of the write back policies used in this study could be modified so that write backs are performed more aggressively when bandwidth is higher or when disk space at the client runs low. If block caching was used, there could also be performance benefits to dynamically setting block size based on the current environment. Prefetching policies may also benefit from this sort of adaptivity.

Finally, an examination of a wider range of network connections could be used to apply this work to other styles of mobility apart from location-independent computing. For example, cache management in a mobile environment where users change location *during* a session could be examined. This could be done by varying the bandwidth over time in some manner, or even having short periods of disconnection during an otherwise weakly-connected session.

References

[1] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proc. of the Thirteenth ACM Symposium on Operating System Principles*, pages 198–212, Pacific Grove, CA, Oct. 1991.

[2] A. Banerji. Answers to frequently asked questions for comp.os.research: Part 1. Available at `http://www.maths.tcd.ie/scrg/os-faq/FAQ-1.html`, 1995.

[3] A. Banerji, D. Cohn, and D. Kulkami. Mobile computing personae. Technical report, Department of Computer Science and Engineering, University of Notre Dame, South Bend, IN, 1993.

[4] P. Cao, E. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proc. First Symposium on Operating Systems Design and Implementation*, pages 165–177, Monterey, CA, 1994.

[5] K. Chen. Write caching in distributed file systems. Master's thesis, University of Saskatchewan, Saskatoon, SK, 1994.

[6] V. Chikarmane. Disconnected operation in a distributed system. In *Proc. Sixth Annual Graduate Symposium on Computer Science*, pages 23–32, Dept. of Computer Science, University of Saskatchewan, March 1994.

[7] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proc. 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150–160, Nashville, TN, May 1994.

[8] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Workshop on Mobile Computing Systems and Applications*, pages 192–195, Santa Cruz, CA, Dec. 1994.

[9] D. Dwyer and V. Bharghavan. A mobility-aware file system for partially connected operation. *ACM Operating Systems Review*, 31(1):24–30, Jan. 1997.

[10] M. Ebling, L. Mummert, and D. Steere. Overcoming the network bottleneck in mobile computing. In *Proc. Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994.

[11] G. Forman and J. Zahorjan. The challenges of mobile computing. *Computer*, 27(4):38–47, April 1994.

[12] K. Froese. File system support for weakly connected operation. In *Proc. Seventh Annual Graduate Symposium on Computer Science*, pages 229–238, Dept. of Computer Science, University of Saskatchewan, March 1995.

[13] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *Proc. of the USENIX Conf.*, pages 201–212, New Orleans, LA, Jan. 1995.

[14] J. Heidemann, T. Page, R. Guy, and G. Popek. Primarily disconnected operation: Experiences with Ficus. Technical report, Department of Computer Science, University of California, Los Angeles, Los Angeles, CA, 1992.

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

[16] L. Huston and P. Honeyman. Disconnected operation for AFS. In *Proc. First USENIX Symposium on Mobile and Location-Independent Computing*, pages 1–10, Cambridge, MA, April 1993.

[17] L. Huston and P. Honeyman. Distributed systems support for telecomputing. Technical report, Center for Information Technology Integration, University of Michigan, Ann Arbor, MI, 1994.

[18] L. Huston and P. Honeyman. Peephole log optimization. In *Proc. Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994.

[19] L. Huston and P. Honeyman. Partially connected operation. In *Proc. Second USENIX Symposium on Mobile and Location-Independent Computing*, pages 91–97, Ann Arbor, MI, April 1995.

[20] T. Imielinski and B. Badrinath. Wireless computing. *Communications of the ACM*, 37(10):19–28, Oct. 1994.

[21] J.J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–23, Jan. 1992.

[22] H. Koch, L. Krombholz, and O. Theel. A brief introduction into the world of 'mobile computing'. Technical report, University of Darmstadt, Darmstadt, Germany, 1993.

[23] G. Kuenning. The design of the Seer predictive caching system. In *Proc. Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994.

[24] G. Kuenning, G. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *Proc. 1994 Summer USENIX Conference*, pages 291–303, Los Angeles, CA, June 1994.

[25] P. Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Proc. Fourth IEEE Workshop on Workstation Operating Systems*, pages 66–70, Napa, CA, Oct. 1993.

[26] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proc. USENIX Winter 1995 Conference on Unix and Advanced Computing Systems*, pages 16–20, New Orleans, LA, Jan. 1995.

[27] J. Landay and T. Kaufmann. User interface issues in mobile computing. In *Proc. Fourth Workshop on Workstation Operating Systems*, Napa, CA, Oct. 1993.

[28] H Lei and D. Duchamp. Transparent file prefetching. Technical report, Department of Computer Science, Columbia University, New York, NY, 1995.

[29] Q. Lu and M. Satyanarayanan. Improving data consistency in mobile computing using isolation-only transactions. In *Proc. Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995.

[30] B. Marsh, F. Douglis, and R. Cáceres. System issues in mobile computing. Technical report, Matsushita Information Technology Laboratory, Princeton, NJ, 1993.

[31] N. Mirghafori and A. Fontaine. A design for file access in a mobile environment. In *Proc. Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994.

[32] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proc. Fifteenth ACM Symposium on Operating Sytems Principles*, pages 143–155, Copper Mountain Resort, CO, Dec. 1995.

[33] L. Mummert and M. Satyanarayanan. Large granularity cache coherence for intermittent connectivity. In *Proc. 1994 Summer USENIX Conference*, pages 279–289, Boston, MA, June 1994.

[34] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994.

[35] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, Jan. 1987.

[36] B. Noble, M. Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. In *Proc. Second USENIX*

Symposium on Mobile and Location-Independent Computing, pages 57–66, Ann Arbor, MI, April 1995.

[37] B. Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *Proc. 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 138–149, Nashville, TN, May 1994.

[38] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The sprite network operating system. *Computer*, 21(2), Feb. 1988.

[39] C. Perkins. IP mobility support. RFC 2002, Mobile IP Working Group, Oct. 1996.

[40] E. Pitoura and B. Bhargava. Maintaining consistency of data in mobile distributed environments. In *Proc. Fifteenth International Conference on Distributed Computing Systems*, pages 404–413, Vancouver, BC, May 1995.

[41] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. In *Proc. 1994 Summer USENIX Conference*, pages 183–195, Los Angeles, CA, June 1994.

[42] J. Saldanha. A file system for mobile computing. Technical report, Department of Computer Science and Engineering, University of Notre Dame, South Bend, IA, 1993.

[43] J. Saldanha and D. Cohn. A hybrid model for mobile file systems. In *Proc. Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994.

[44] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proc. 1985 Summer USENIX Conference*, pages 119–130, Portland, OR, June 1985.

[45] M. Satyanarayanan. Fundamental challenges of mobile computing. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996.

[46] M. Satyanarayanan, J.J. Kistler, L.B. Mummert, M.R. Ebling, P. Kumar, and Q. Lu. Experience with disconnected operation in a mobile computing environment. In *Proc. First USENIX Symposium on Mobile and Location-Independent Computing*, pages 11–28, Cambridge, MA, April 1993.

[47] A. Smith. Disk cache—miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, 1985.

[48] C. Tait. *A File System For Mobile Computing.* PhD thesis, Columbia University, New York, NY, 1993.

[49] C. Tait and D. Duchamp. Service interface and replica management algorithm for mobile file system clients. In *Proc. First Intl. Conf. on Parallel and Distributed Information Systems*, pages 190–197, Miami Beach, FL, Dec. 1991.

[50] C. Tait and D. Duchamp. An efficient variable-consistency replicated file service. In *Proc. USENIX File Systems Workshop*, pages 111–126, Ann Arbor, MI, May 1992.

[51] C. Tait and D. Duchamp. An interface to support lazy replicated file service. In *Proc. Second Workshop on Management of Replicated Data*, Monterey, CA, Nov. 1992.

[52] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. First Symposium on Operating System Design and Implementation*, pages 1–11, Monterey, CA, November 1994.

[53] M. Weiser. Hot topics: Ubiquitous computing. *Computer*, 26(10):71–72, 1993.

[54] D. Willick, D. Eager, and R. Bunt. Disk cache replacement policies for network file servers. In *Proc. Thirteenth International Conference on Distributed Computing Systems*, pages 2–11, Pittsburgh, PA, May 1993.

[55] Y. Zhang and S. Dao. A "persistent connection" model for mobile and distributed systems. In *Proc. Fourth International Conference on Computer Communications and Networks*, pages 300–307, Las Vegas, NV, Sept. 1995.

Appendix A

Trace Collection

This appendix describes the tracing software and trace collection process in detail. The post-processing of traces is also discussed.

Tracing was performed in the Distributed Computing Laboratory in the Department of Computer Science at the University of Saskatchewan. Specifically, a set of four HP 700 workstations running HP-UX 9.0 were instrumented to enable them to gather traces of file system activity. There were three main phases to the collection of traces. First, it was necessary to make the necessary software modifications to enable tracing on the HP 700 workstations; traces were then collected, and then processed into a form suitable for the simulator. Each of these phases is discussed in detail.

A.1 Tracing Software

Tracing was performed using *KItrace* (Kernel Instrumentation Trace), a user program which operates cooperatively with the operating system kernel to gather and record information about kernel and system calls. The operation of KItrace is illustrated in Figure A.1. There are two main interacting components: the KItrace kernel code, and the KItrace daemon program. The daemon program runs in user space and interacts with the kernel code. Whenever an application makes a system call (such as `open()` or `read()`), the appropriate kernel code is executed, but before the system call returns, an additional call is made to KItrace code (also in the kernel). This subroutine allocates a trace buffer, and fills in a trace record with the appropriate data (including a time stamp, with microsecond resolution). The

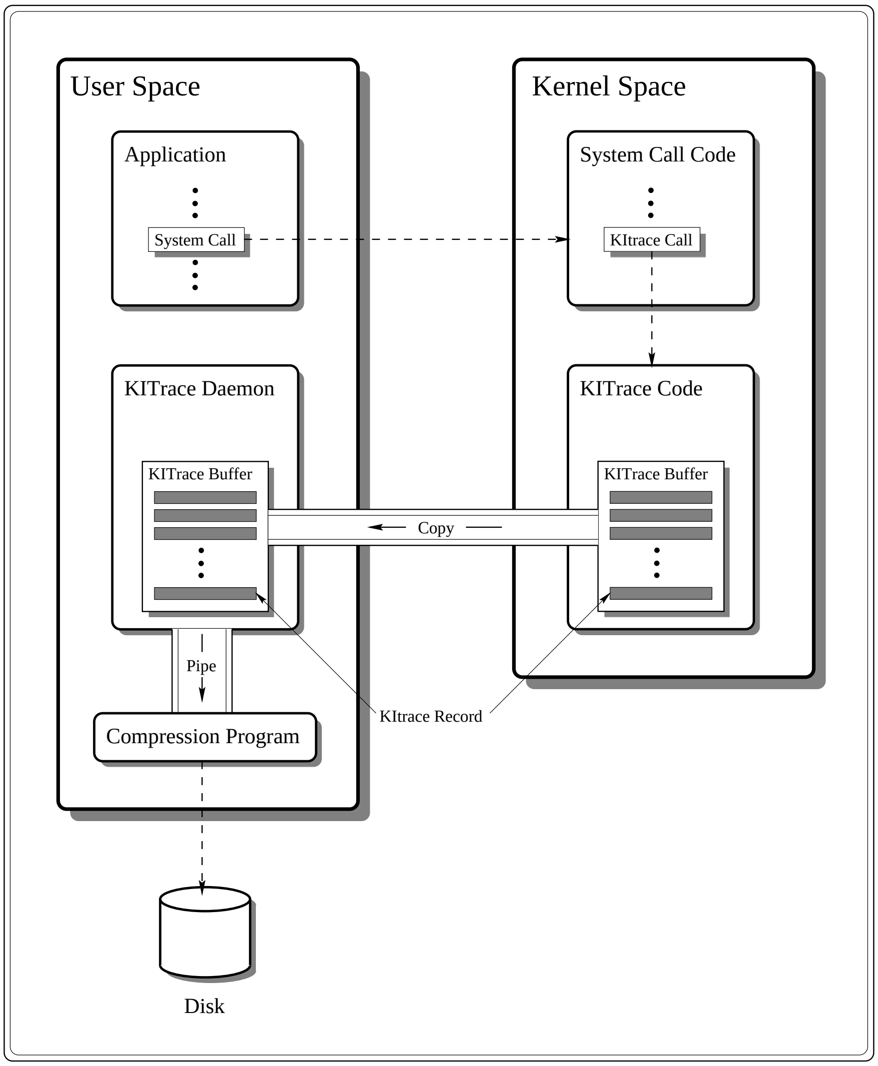

Figure A.1: KITrace operation

KItrace daemon program waits for a trace buffer to be filled, then copies the buffer, and writes it to disk. Trace buffers are large enough (8 kB) that the daemon program has enough time to write one buffer to disk before the next one fills up. Figure A.2 shows the contents of sample trace records.

It is important to note that the system calls made by the KItrace daemon program *are* recorded in the traces. This "contamination" of traces is removed during post-processing (See Section A.3).

Because KItrace requires kernel support to run, a single version of KItrace can not be used with different versions of the kernel. Initially, the only version of the

```
0 166.356 pid= 16054 uid= 5690 open ret= 3 err= 0 data= 657fec
type= VNODE dev= 119543296 inode= 206415 size= 69632
mode= 100555 name= '/lib/dld.sl'

0 166.920 pid= 16054 uid= 5690 closef data= 657fec type=VNODE
count= 0

0 725.457 pid= 16052 uid= 5690 read ret= 512 err= 0 data= 66e530
type= VNODE dev = 119543296 inode= 20967 size= 1188 mode= 100444
offset= 512 amount= 512 name= '.login'

1 372.279 pid= 16052 uid= 5690 write ret= 79 err= 0 data= 686108
type= VNODE dev = 119543296 inode= 9649 size= 0 mode= 20622
offset= 1158 amount= 79 name= '/rjg270/.login'
```

Figure A.2: Sample KItrace records

KItrace daemon program available was for use with the HP-UX 8.0 kernel, while HP-UX 9.0 was used on the workstations where tracing was to be performed. This meant that the KItrace daemon program needed to be modified to operate under the new kernel. This involved an identification of how the kernel data structures referenced by KItrace had changed, and modifying KItrace to use these new structures.

Unfortunately, not all system call information required for this study was present in the KItrace records. For example, user ID and file size information was not recorded. Therefore, further modifications were made to KItrace to collect this information. Three sets of modifications were required: new fields were added to the KItrace record data structures; for each modified data structure, KItrace code in the kernel was added to find the new information and record it; and the KItrace daemon program was modified to recognise the new trace record structures.

The final modification made to the KItrace daemon program was to have it pipe its output to a compression program, so that traces would be sent to disk compressed. This resulted in a 75% reduction in the size of trace files.

A.2 Trace Collection Methodology

Traces were collected on four workstations, each running the customised version of the HP-UX 9.0 kernel. The 24-hour traces were captured from midnight to midnight, simultaneously on all four workstations. This was done for several (non-consecutive) working days, after which the five traces containing the most activity were chosen for use in this study.

The seven-day traces were gathered in a similar manner. Tracing was performed on each workstation simultaneously. Due to storage limitations, traces were moved to off-line storage at the end of each day. This resulted in four sets of seven 24-hour traces collected on consecutive days.

A.3 Trace Post-Processing

Post-processing was required to convert the "raw" traces into a form suitable for use with the simulator. There were three main post-processing activities which needed to be performed. First, the individual 24-hour segments of the seven-day traces from each workstation needed to be combined to form a single trace. This was done by first incrementing the time stamps in each trace segment as appropriate for the day on which the segment was captured. For example, the time stamps in the second day's trace were incremented by 24 hours, the time stamps in the third day's trace were incremented by 48 hours, etc. Then, the trace segments were joined into a single trace file, equivalent to one which had been gathered for a continuous seven-day period.

The second post-processing step was to filter out unwanted events from the traces. These events included ones generated by the KItrace daemon program itself, as well as events produced by the moving of traces files to off-line storage at the end of the day. Additionally, events recording "unsuccessful" operations, such as attempts to read from a non-existent file, were discarded.

The final stage of post-processing was the extraction of the activity of individual users from the traces. This was done by first finding all unique user identification numbers (UIDs) in a trace, then extracting all events with a given UID from each trace and saving them in a new trace file. For the seven-day traces, this resulted in a set of four week-long traces for each user, one from each workstation. For each user, the four traces were combined into a single trace, maintaining the same time stamps on each event. The final result was a set of traces, each of which recorded the activity of a single user across all four workstations for a seven-day duration. The four such traces containing the greatest amount of activity were selected for use in this study.

Appendix B

Simulator Algorithms

The simulator's main algorithm is shown in Figure B.1. File system events are read from a trace file one at a time. After each event is read, the current simulation time is incremented by the difference in time stamps on the last two events read from the trace file (including the event just read). For example, if the current simulation time was 100.0 seconds, the time stamp of the last event was 20.0 seconds, and the time stamp of the newly read event was 21.5, then the new simulation time would be $(100.0 + 21.5 - 20.0 =)$ 101.5 seconds. Next, there is a chance for write backs to be performed. Whether or not a write back is performed depends on the current write-back policy in use. Figures B.2, B.3, B.4, and B.5 show the algorithms used by each policy. If the time consumed by write backs exceeds the time between the last two events read, write interference has occurred. The amount of interference time is recorded.

After any write backs have completed, the most recently read event is processed, as shown in Figure B.6. If the event is a write, one of two courses of action will be taken. If the client is strongly connected, the write is propagated back to the file server immediately. Otherwise, the log file is updated to record the write. Once the write has been processed, or if the event was a read, the cache is searched for the file referenced. If the file is present in the cache (i.e. a cache hit occurs), reference statistics are updated, but no further processing is required. If a cache miss occurs, cache replacement takes place (according the replacement policy currently in use), and the newly referenced file is added to the cache.

Once the event has been processed, the simulator checks to see if the network

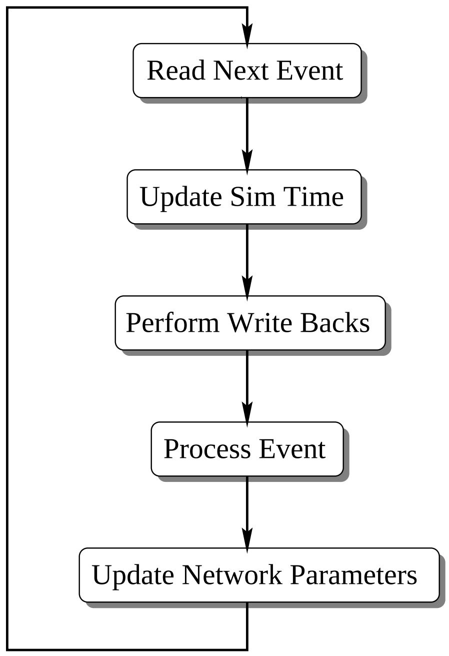

Figure B.1: Main simulator algorithm

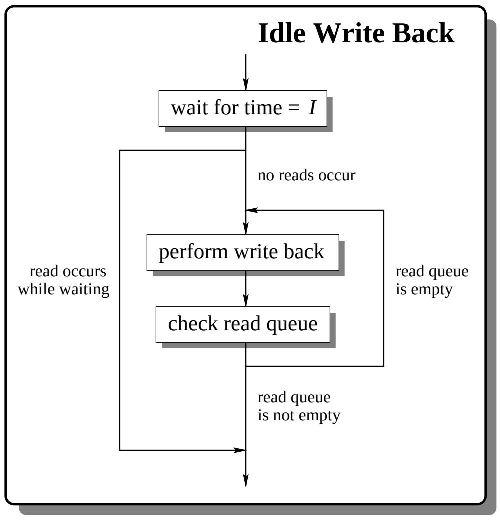

Figure B.2: Idle Write Back algorithm

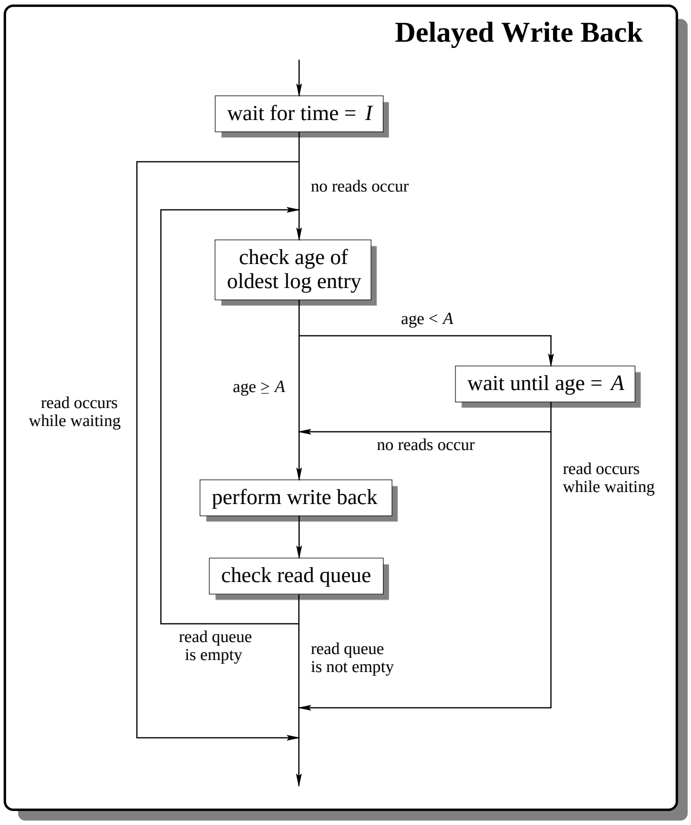

Figure B.3: Delayed Write Back algorithm

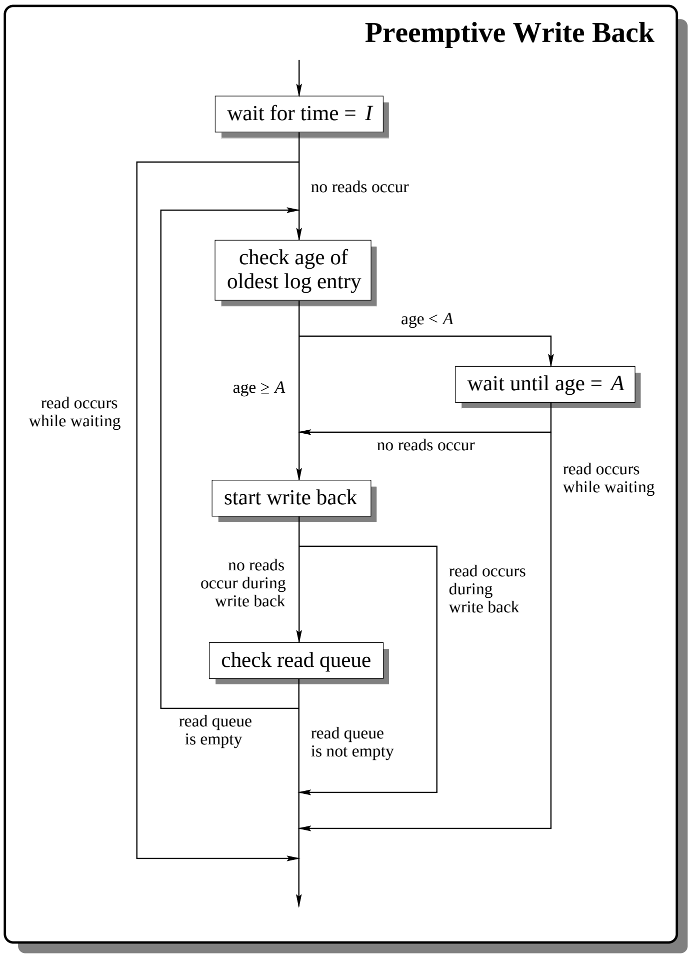

Figure B.4: Preemptive Write Back algorithm

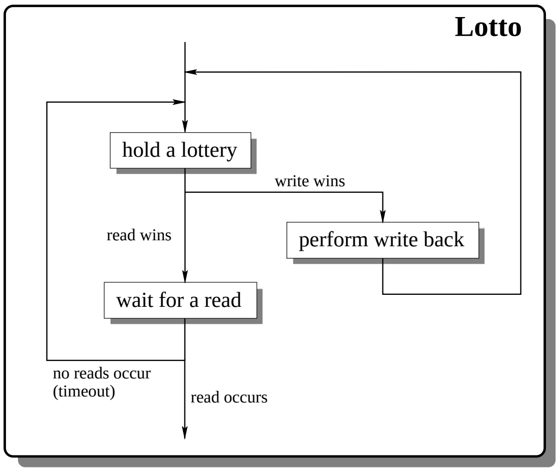

Figure B.5: Lotto algorithm

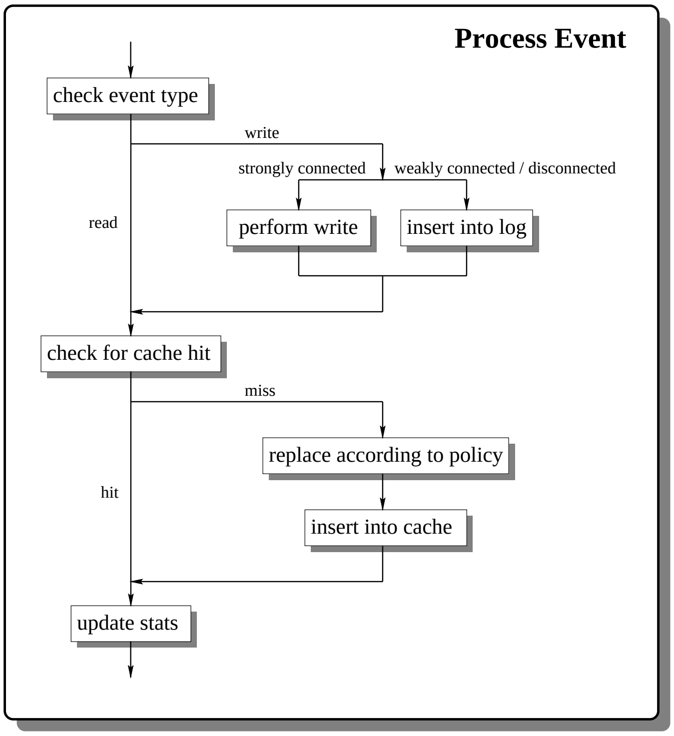

Figure B.6: Process event algorithm

parameters should be changed. A list of (simulation) times and corresponding set of network parameters is read at the start of the simulation. Whenever the simulation time equals or exceeds a listed time, the network parameters are changed accordingly.

Each event in the trace file is processed in this manner. Once every event has been processed, the simulation is ended and performance statistics are recorded.