

Intelligent File Hoarding for Mobile Computers

Carl Tait

IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
tait@watson.ibm.com

Swarup Acharya

Computer Science Department
Brown University
Providence, RI 02912
sa@cs.brown.edu

Hui Lei

Computer Science Department
Columbia University
New York, NY 10027
lei@cs.columbia.edu

Henry Chang

IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
hychang@watson.ibm.com

Abstract

Mobile computing adds a new wrinkle to the age-old problem of caching. Today's wireless links are both slow and expensive, and are not always available to a user. Therefore, when a mobile user is disconnected, a cache miss means (at best) a substantial cost in time and money, or (at worst) a complete halt to work if critical information has not been cached. Existing solutions to this problem rely on some combination of explicit *hoard profiles* and *spying* on a user's file accesses. Neither of these approaches is ideal in terms of reliability or user-friendliness. Our solution might be called *transparent analytical spying*. Instead of simply recording a list of file accesses, we analyze program executions and tie them to specific files. By observing multiple executions of a program, we are able to build sets of files associated with each program, and can heuristically separate application files from user files. We can then present a high-level view to the user that is similar to loading a briefcase. We have an implementation of our hoarding tool running under OS/2.

1 Introduction

We begin by describing the problem in question and enumerating the existing approaches to solving it.

1.1 The Hoarding Problem

In traditional computing, caching is used to improve performance. It is faster to fetch a file from memory than from a local disk; similarly, reading a file from a local disk is faster than obtaining it from a remote file server. Caching is an ancient and well-understood technique with obvious advantages.

The increasing popularity of mobile computing introduces a new wrinkle into this old idea. The expected benefits of caching still apply — and with even more force, if file servers can be reached only through slow wireless links. But a critical issue for mobile clients is *disconnection*. Temporary, involuntary disconnection from file servers is always possible, in which case an intelligently filled cache is certainly an advantage. The more interesting case, however, is *voluntary, client-initiated* disconnection, such as when a user takes a portable computer home from the office or on a trip. In fact, it could be argued that this characteristic is central to the whole notion of “mobile computing”: clients can be viewed as primarily disconnected entities that periodically connect to and disconnect from servers at the clients' discretion [3].

This offbeat manifestation of the client-server model forces a rethinking of standard cache management policies. Even assuming the existence of an anytime, anywhere, on-demand wireless communication service, a cache miss while disconnected is an expensive proposition, due to the cost and inconvenience of re-establishing communications with the server. So we are led to an interesting question: given that a client *informs* the system of an impending disconnection, how do we intelligently fill the cache with files that will allow the client to continue working while disconnected? This is known as the **hoarding problem**.

The goal of hoarding is to eliminate cache misses entirely during periods of client disconnection. Since hoarding is a relatively infrequent operation performed only at a client's request prior to disconnection, timing is not critical. A hoarding process that takes a few minutes prior to disconnecting for many hours is certainly acceptable. Furthermore, overkill is tolerable and probably even necessary. To ensure that a client can continue to access all necessary files, the hoarding scheme may well cache some files that the client never uses during that particular period of disconnected operation.

1.2 Existing Approaches

This subsection describes current techniques for solving the hoarding problem in order to lay the groundwork for our own ideas.

Do Nothing. In their implementation of disconnected AFS [4] and their extensions for partially connected operation [5], Huston and Honeyman have not provided any special mechanisms for hoarding. Prior to disconnection, the user simply “runs the applications she intends to use while traveling,” thus filling the local disk cache appropriately. In addition to being inconvenient, this method suffers from an intrinsic drawback: a program may require different files on different executions. No single execution will reveal the full gamut of an application’s file accesses.

User-Provided Information. A user might specify precisely which files and directories should be hoarded in a *hoard profile*, as is done in Coda [6, 9]. This is a straightforward approach, but can be both burdensome and unreliable. The burden can be eased by distributing common hoard profiles to naive users, but creating an accurate hoard profile in the first place is not at all trivial. Even a conscientious user might not be able to accurately specify (or even be aware of) all of the files needed by a specific program.

Snapshot Spying. In this approach, the user sets up *bookends* to delimit a period of activity. During this period, the hoarder spies on the user’s activity, keeping a record of which files are accessed. When snapshot spying is combined with a user-specified hoard profile, as Coda does [10], the results are encouraging. However, snapshot spying suffers from the same drawback as the “run all applications” approach described above: a single program execution will not necessarily reveal all the files an application needs. Furthermore, traditional spying is not program-based; it simply produces a named collection of accessed files.

Semantic Distance. Kuenning’s Seer system [7] uses a method for transparent hoarding (“predictive caching”) that is based on the notion of *semantic distance* between files. There are several ways such distance could be measured; Kuenning has chosen to analyze the sequence of file accesses (open and close operations). This measure is time-independent, and does not require the user to have any special knowledge. Seer uses *observer* and *correlator* processes to spy on the user’s activity and to detect clusters of files based on their proximity in the file access sequence. It is not yet clear how effective this method will be. One known restriction is the inability to separate file accesses performed by concurrently executing processes.

2 Our Approach

Our approach is based on two of the authors’ previous work in file *prefetching* [13, 12, 8]. Both hoarding and prefetching involve anticipatory file fetches: bringing files from a remote server into a local cache before

they are needed. The goals of these two techniques are quite different, however. Prefetching is an online technique that is mainly concerned with improving performance: by bringing soon-to-be-needed files over the network during slack periods when the link is not busy, prefetching can create the illusion of low-latency access to remote files.

When prefetching, a cache miss carries the usual penalty: a slower read operation. Although prefetching attempts to minimize the number of cache misses, some misses may reasonably be expected to occur, and they do not have catastrophic consequences. Timing is important: the prefetcher must continually hurry to bring over files before they are accessed, and this requires adroit use of client resources such as excess CPU cycles and unused network bandwidth.

Our approach to hoarding might be called *transparent analytical spying*, and relies on the notion of *file working sets* used in our prefetching work. The method we use can be outlined as follows:

1. Automatically detect working sets for applications and data.
2. Provide *generalized bookends* to delimit periods of activity.
3. Present convenient bundles to the user through a graphical user interface (GUI).
4. Let the user “load the briefcase.”

Each of these activities will be described in the following subsections. A primary concern is that the system perform adequately for naive users, and even better for sophisticated users who are able to provide more information.

2.1 Detection of Working Sets

We detect file working sets by continuously monitoring user file accesses in the background and recording this information in a log. At hoard time, the log is analyzed, and trees representing program executions are constructed using the process ID (pid) and parent information in the log. Each file is a node in the graph; children of a node are the files that have been opened or executed by the parent node. Immediate children of shells are deemed to be roots of program trees. (Under OS/2, these correspond to programs started at the command line or through an icon.) After a tree is built, its pattern is recorded in a simple database on the local disk.

Figure 1 is an example of a program tree. A program `\a\b\c` has been executed, which has in turn forked three subprograms¹. The first and third subprograms, `c1` and `c3`, each open a data file provided by the user. The second subprogram, `c2`, opens two files associated with the application itself: for example, these might be font files if the application were a text processor.

The tree-based approach provides two significant advantages. First, we are able to associate files with specific applications for clarity of presentation to the user.

¹Unlike UNIX, both OS/2 and DOS use *backwards* slashes to delimit pathname components.

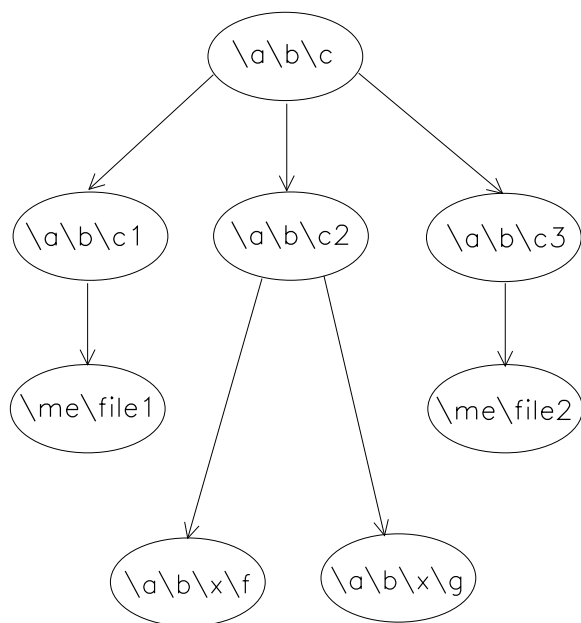


Figure 1: Sample Program Tree

Second, we are able to generalize across multiple executions of the same program, while traditional snapshot spying is limited to recording the files that were accessed during a specific execution. This is particularly important for OS/2 applications, which rely heavily on dynamic link libraries (DLLs) for runtime support. A particular DLL may be needed to execute the program in certain circumstances, but this will not necessarily be evident from a single execution of the application. (There is no general way to determine all of a program's DLL dependencies by examining executable files.)

Our collection of execution trees allows us to see a bigger picture: at hoard time, we can grab all of the files in the observed execution trees. Essentially, we perform a union of all the trees for a particular program. In order to prevent storage requirements from becoming unwieldy, we limit the maximum number of trees saved for each program. The current maximum is 15 trees, and they are maintained on an LRU basis.

In certain circumstances, the user may prefer not to generalize a program, but instead hoard only those files involved in a specific execution. For example, a program might have a large help facility that the user does not want to hoard. In this case, we provide a switch that allows the user to hoard only the files involved in the most recent execution of the program in question.

(Of course this means that the user will be relying on a single execution of the program to define hoarding information, which has the attendant problems discussed above.)

2.2 Applications vs. User Data

A problem with the scheme described above is that it fails to differentiate between application files and user data files. Hoarding the union of multiple trees will provide the user with needed application files, but it may also waste considerable space by hoarding the data files associated with multiple executions of a program. Also, if a global system-wide profile of a program (accrued from multiple users) is maintained, it is particularly important to have the ability to differentiate each user's data files from the program files. Hence, we have implemented a sequence of heuristics to help the user filter out data files, if desired. In each case, we are given a parent program, and are trying to decide if a file it has accessed should be classified as part of the program or part of the data.

A. Filename Extension. Like DOS, OS/2 observes certain conventions for the three-character filename extensions of executable files. **EXE** and **COM** files are executable binaries; **CMD** files are OS/2 batch files or REXX command language programs; **BAT** files are DOS batch files. In most cases, such executables should be considered part of the parent program. (There are certain exceptions: for example, the linker produces an **EXE** file as output data.) Furthermore, **INI** is the standard extension for a program initialization file that contains a user's startup preferences, so **INI** files are also classified as application files. If the filename's extension reveals no information, we proceed to the second heuristic.

B. Directory Inferencing. Assuming logical organization of applications and their associated files, it is likely that files related to the same program will have similar locations in the directory hierarchy. To put it another way, files that fail to share at least the top-level component of directory structure are unlikely to be parts of the same application. For example, if the parent program is `\piano\keyboard\prog1`, we will immediately classify `\anchovy\pizza` as a data file since the top level directories are different. However, `\piano\music` might be part of the program. In cases like this, we proceed to the third heuristic to make our final decision.

C. Timestamps. Modification time provides a further hint for the application-or-data classification. User files have typically been modified in the recent past, while application files may not have been updated for many months (or even longer). In addition, files that are part of the same application will usually have similar modification times. So our final heuristic can be stated as follows: if the parent and child files have not been modified in the last M months, and their modification times are within U months of each other, then we consider the child to be part of the parent program.

Otherwise, the child is classified as data. Our current values for M and U are three months and one month respectively, though this is subject to re-evaluation.

Note that files need not be executables in order to be classified as application files. For example, standard include files such as `stdio.h` would be associated with the compiler rather than with the user's source files, assuming that the compiler and the standard include files share the same top-level directory.

This hierarchy of heuristics allows us to decide with reasonable accuracy how to classify a file. By their nature, however, heuristics are not foolproof, and thus our default is to hoard *all* files in the collection of execution trees for a given program. Program/data separation is performed only when the user explicitly requests it.

2.3 Generalized Bookends

Generalized bookends are a natural extension of snapshot spying. As in the basic snapshot method, the user marks the beginning and end of a period of activity with *bookends*, and assigns a name to the collection of file accesses that take place during that time. We extend this idea in two ways.

First, we detect and record the individual programs executed during the delimited spying period, using the methods described in Section 2.1. This provides more information about the nature of the spying session than an unstructured list of files. Second, and more importantly, we generalize the constituent programs across multiple executions, using the same methods as described above. This allows the hoarder to see beyond the specific file accesses that happened to occur during a particular spying session. The user is not limited to the (possibly obscure) files that the program happened to use on that execution.

As with individual programs, application files and user data files can be separated at the user's discretion. Furthermore, we provide options at hoard time to disable generalization in two different ways:

- Hoard only the specific files observed during the spying session. (This is equivalent to snapshot spying.)
- Hoard only the files involved in the most recent execution of each program in the spying session.

2.4 Presentation and Selection

After detecting the relevant patterns at the program and bookend level, the matter of presenting this hard-won information to the user for selection seems almost trivial. It turns out, however, that designing a readily comprehensible user interface is surprisingly tricky. We will therefore defer discussion of the interface and selection methods until Section 3.3, after describing the details of the implementation.

3 Implementation

3.1 Environment

Our hoarding tool was developed using OS/2's Installable File System (IFS) mechanism, which supports the coexistence of multiple, active file systems. The IFS mechanism presents a standard set of file I/O APIs to users. Any specific file system installed as an IFS must implement a standard set of internal file system operation APIs. (This is similar to the vnode layer in UNIX.)

The mobile file system underpinning our efforts is IBM's Mobile File Sync (MFS) [11], a caching IFS similar in spirit to Coda [9] and Ficus [2]². MFS presents a single file system image regardless of whether the user is currently connected to the relevant file server. A conflict detection mechanism is provided to cope with the potential problems of optimistic replication. One interesting point is that MFS is a client-side-only solution: no server modifications are necessary.

File accesses are traced through hooks in the file operation router of MFS. Except for `fsctl` operations³, these hooks are activated after a file system operation completes, so we are able to distinguish between successful and failed operations. New `fsctl` modes have been added to enable and disable tracing as required.

This implementation environment has certain advantages. First, no kernel changes are required. This makes debugging substantially less difficult. Second, integrating the hoarding code into this well-established interface is a fairly straightforward task. Third, most of the hoarding code (with the exception of the tracing hooks) is portable to any IFS-based file system without modification.

Certain disadvantages are also apparent. Although it is true we are not *required* to modify the kernel, it is also true that we *cannot* modify it even if we wanted to. As it turns out, we would indeed prefer to extract more information than is possible through the IFS interface. There is no "hook" in OS/2 through which we can observe all file activity in the system; instead, we are limited to the files that are accessed through the IFS. This leads to a tricky problem that will be discussed in the next section.

3.2 The Orphan Problem

Due to the nature of our trace, local file accesses are completely invisible to us. Furthermore, the trace data we *can* gather is not fully complete even for IFS files: substantial programmatic tap-dancing is required to construct the execution trees. These factors lead to an *orphan problem* that can take one of two forms.

Data orphans. When a local program accesses remote data files, we have no way to associate the data files with a parent program. This is also true in the case of internal functions: if a user enters `type myfile`

²The hoarding code was originally built on top of the Shadow File System [1], a prototype mobile file system developed at IBM Research.

³`fsctl` is similar to UNIX `ioctl`.

(where the given file is on a remote file server), all the trace will show is an isolated reference to `myfile` by an unidentified process. The solution we have reluctantly adopted is to display all data orphans to the user at hoard time: “We know you’ve used these files, but we have no idea which program (if any) they belong to.” This will be discussed further in Section 3.3.

Program orphans. If a remote program accesses *only* local data — for example, an intermediate phase of a compiler operating purely on temporary files — we have no way of attaching the program to its parent. This seems bizarre (and it is), but this is an unfortunate side effect of our limited trace data. Because of the way we receive information about program executions, it is necessary for an executing program to make a file access we can *see* before the pid information can be resolved. The technique is similar to resolving a forward reference during compilation of a program.

Our fix for program orphans is simple and transparent, but not necessarily optimal: in each directory, we maintain a list of program orphans that have been detected in that directory. Whenever we hoard an executable file, the program orphans associated with the executable’s directory are brought along automatically. Since application files are usually hierarchically organized by system administrators or installation programs, this approach is plausible.

3.3 Graphical User Interface (GUI)

The GUI has been designed to be relatively unobtrusive. The casual user can accept the suggested defaults and be done with it. Those who prefer to dig deeper will find it relatively painless to customize their own profiles. Help screens are available on every panel to assist ambitious users.

Spying Control GUI. Figure 2 shows the Spying Control panel, which comes up when the hoarding tool is invoked. This panel may be immediately minimized and ignored. It serves three purposes:

1. Turning the trace on or off (tracing is off by default).
2. Opening and closing named bookends to delimit periods of spying.
3. Initiating the hoarding process.

When the user presses the **Hoard** button, the current trace is analyzed. After a moment, the Selection GUI appears.

Hoarding Selection GUI. Figure 3 shows the Hoarding Selection panel, which is the heart of the hoarding tool. However, it may be ignored by unadventurous users simply by pressing **OK** to accept the defaults. The three columns display user-defined bookends, automatically detected programs, and data orphan files. Programs are displayed as the final components of their pathnames (without the extension), though the full pathnames may optionally be displayed instead.

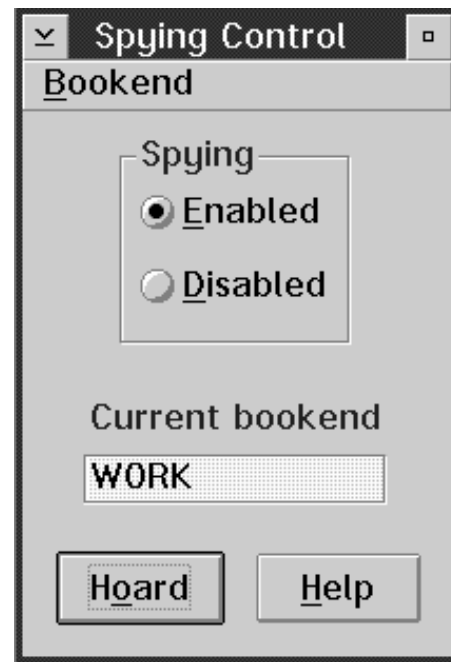


Figure 2: Spying Control GUI

In order to decide which programs and orphan files to display, a time-based filter is used. (Bookends are always displayed, since they are not time-based.) By default, only programs and orphans accessed during the last week are displayed, and all of them are marked for hoarding. The **Since** menu allows the user to change the filtering date, providing friendly choices such as **Yesterday**, **One month ago**, and **Forever** (show the whole database). Finicky users have the option to enter an explicit date. Changing the **Since** date automatically selects all displayed programs and orphans for hoarding. Individual items in all three columns may be selected or deselected as desired. The user may also choose to permanently delete certain items from the database.

The **Additional Files** area at the bottom of the Hoarding Selection panel corresponds to a traditional hoard profile. Items can be added to (or deleted from) this list through yet another dialog. As in most hoard profiles, entries can correspond to individual files, directories, or entire subtrees of the directory structure.

Figure 4 shows the Options panel provided for bookends and programs. As discussed in Section 2, the user has choices along two dimensions:

- Level of generalization, and
- Program files, data files, or both.

For programs, generalization is a binary choice: get all patterns, or hoard only the most recent execution tree. **Specific** is not selectable in this case. For bookends, the user has the additional choice of hoarding only the specific files accessed while the bookend was



Figure 3: Hoarding Selection GUI

open (snapshot spying). Programs and bookends may both have program or data files selectively filtered out. The default is always to grab everything: all patterns; programs and data.

When the user is satisfied with the selections and options, pressing the **OK** button initiates the hoarding process.

3.4 Time and Space Requirements

Our measurements show that tracing adds approximately 2% overhead to the time required to access a file. Each file record in the trace log occupies about 100 bytes. This includes the fully-qualified pathname, file

size, open flags and open mode, and four timestamps (create, open, update, and access). After the trace log is analyzed, each file reference requires a comparable amount of space in the permanent database. We believe the space required for both log and database records could be reduced to approximately 40 bytes per record, though we have not yet implemented such compression.

Analyzing the trace log takes a variable amount of time, and depends as much on the complexity of the trace log as on its size. Our hoarding database is based on programs and processes, so each process record in the trace log requires a corresponding new or additional record in the database. Long but simple logs involving only a few distinct processes can be analyzed fairly quickly: about 1.1 seconds per 10 kilobytes of log file, using a 25 MHz Intel 486SX. Log files that contain many separate processes and subprocesses may take considerably longer: up to about 8 seconds per 10 kilobytes of log file.

4 Status

We currently have developed a prototype and are using it with MFS. From a practical perspective, our main limitation is our lack of experience with the tool in a realistic setting. Since the hoarding code was developed in parallel with the mobile file system on which it runs, real-world testing has been necessarily limited. To date, our confidence in the hoarding tool stems primarily from usage traces we have gathered. Indeed, most of our design decisions grew from what we considered to be typical traces of commonly-performed tasks. Whether our traces reflect the usage patterns of the world at large remains to be seen.

5 Future Work

There are a number of possibilities for future work. Some potentially useful ideas include:

- Analyze the trace log in the background rather than deferring analysis until hoard time.
- Compress the size of the trace log.
- Implement sophisticated pattern merging of execution trees. This will both minimize the number of trees saved and allow us to see long-term patterns not apparent in the current LRU-based method.
- Add a priority scheme. This is important when there is not enough room in the cache to hoard all desired files. It is not yet clear what the priority relationship should be between automatically detected files and ones that are explicitly specified. Ideally, default priorities should be calculated automatically for frequently-used programs. Files associated with a program could be assigned priorities based on their perceived importance to the application: if program **P** uses file **F1** in 95% of its executions and file **F2** only 53% of the time, then **F1** should be assigned a higher priority than **F2**.
- Store program trees on the server. Since this information is needed only at hoard time, it is not

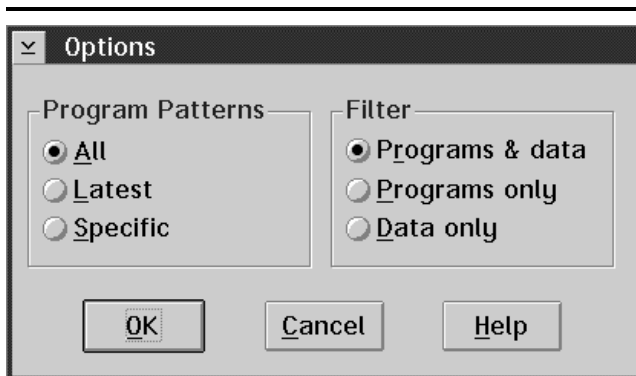


Figure 4: Hoarding Options GUI

necessary to save trees locally (though of course the trace log itself must always be local). In addition to saving local disk space, this will allow users to exchange patterns for their mutual benefit.

- Figure out how to handle multiple identical servers. For example, at the Watson Research Center, users are connected to one of several identical servers every time they reboot their machines. Traces gathered while attached to one server are of no use while attached to a different one. Previous work by Zadok and Duchamp [14] addresses the problem of discovering and using “equivalent” read-only file systems. In our case, however, a user is connected to only one server at a time, which makes comparison more difficult.
- When and if a lower-level tracing facility in OS/2 becomes available, modify the existing trace mechanism to fix the orphan problem.

6 Summary

We have presented the design and implementation of an intelligent file hoarding tool for mobile computing. The tool automatically detects a user’s file access patterns and presents them in a convenient form at disconnection time. Naive users can accept the chosen defaults; sophisticated users can fine-tune the selections as desired. Our initial experience with the hoarding code has been positive, though more rigorous analysis (and real-world testing) is needed.

References

- [1] H. Chang, F. Novak, C. Tait, and P. Hortensius. SFS: A Universal File System Cache for Disconnected FS Operations. In *Joint Conference on Information Sciences*, November 1994.
- [2] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *Proc. 1990 USENIX Summer Conf.*, pages 63–71, June 1990.
- [3] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily Disconnected Operation: Experiences with Ficus. In *Proc. Second Workshop on the Management of Replicated Data*, pages 2–5. IEEE, November 1992.
- [4] L. B. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proc. First USENIX Symposium on Mobile and Location-Independent Computing*, pages 1–10, August 1993.
- [5] L. B. Huston and P. Honeyman. Partially Connected Operation. In *Proc. Second USENIX Symposium on Mobile and Location-Independent Computing*, pages 91–97, April 1995.
- [6] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, February 1992.
- [7] G. H. Kuenning. The Design of the Seer Predictive Caching System. In *IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.
- [8] H. Lei and D. Duchamp. Transparent File Prefetching. In preparation, Columbia University, 1995.
- [9] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. on Computers*, 39(4):447–459, April 1990.
- [10] 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, August 1993.
- [11] A. Shaheen, S. Jaji, T. Porcaro, R. Ward, and K. Yellepeddy. Architecture of a Mobile File System. In *Joint Conference on Information Sciences*, November 1994.
- [12] C. D. Tait. *A File System for Mobile Computing*. PhD thesis, Columbia University, 1993.
- [13] C. D. Tait and D. Duchamp. Detection and Exploitation of File Working Sets. In *Proc. Eleventh Intl. Conf. on Distributed Computing Systems*, pages 2–9. IEEE, May 1991.
- [14] E. Zadok and D. Duchamp. Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing. In *Proc. 1993 Summer USENIX*, pages 69–85, June 1993.