Seer: Predictive File Hoarding for Disconnected Mobile Operation

Geoffrey H. Kuenning

University of California, Los Angeles

May, 1997

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

UCLA Computer Science Department
Technical Report UCLA-CSD-970015

In memory of my father, John Horace Kuenning,
November 12, 1919–May 2, 1997

For Alyssa, who set me free,
and for Pat, who gave me wings

Contents

List of Figures

List of Tables

List of Symbols

$\mathcal{F}$	Set of files	24
F	Number of files in $\mathcal{F}$	24
f_i	i^{th} file	24
$\mathcal{R}$	Basic reference sequence	24
r_i	i^{th} member of $\mathcal{R}$	24
T	Number of references in a sequence	24
d_{ij}	Basic local reference distance	25
*	All pairs	25
$\mathcal{D}$	Generic basic distance multiset	25
$^{\leftrightarrow}$	Pairwise	25
$^{\rightarrow}$	Reference-to	26
$^{\leftarrow}$	Reference-by	26
N	Cardinality of a reference set	27
$\mathcal{S}$	Open/close reference sequence	28
s	Member of $\mathcal{S}$	28
s^o	Open member of $\mathcal{S}$	28
s^c	Close member of $\mathcal{S}$	28
ℓ_{ij}	Lifetime local reference distance	28
$\mathcal{L}$	Generic lifetime reference distance multiset	28
$\mathcal{H}$	Summary histogram of $\mathcal{D}$	28
$\mathcal{J}$	Summary histogram of $\mathcal{L}$	28
h	Member of $\mathcal{H}$	28
p_{ij}	Local probability	29
L	Lookahead	29
${}^aD$	Arithmetic-mean reference distance	29
${}^aL$	Arithmetic-mean lifetime reference distance	29
${}^gD$	Geometric-mean reference distance	30
${}^gL$	Geometric-mean lifetime reference distance	30
s_i	i^{th} member of $\mathcal{S}$	35
f_i^o	Open of f_i	35
f_i^c	Close of f_i	35
O_{MAX}	Maximum open files	35
I	Increase in a value	38
$\widetilde{{}^aL}$	Approximate arithmetic-mean distance	40
$\widetilde{{}^gL}$	Approximate geometric-mean distance	40
$\mathcal{L}^+$	Files included in $\mathcal{L}$	40
$\mathcal{L}^-$	Files excluded from $\mathcal{L}$	40
N^+	Number of files included in a reference set	40
N^-	Number of files excluded from a reference set	40
ϵ_i	Relative error in distance estimate i	40

Abstract of the Dissertation

Seer: Predictive File Hoarding for Disconnected Mobile Operation

by

Geoffrey H. Kuenning
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1997
Professor Gerald J. Popek, Co-chair
Professor Wesley W. Chu, Co-chair

Because of the limited storage space available on portable computers, disconnected mobile users must restrict their work to a subset of the files available on their network. The list of files needed to accomplish useful work is large, non-intuitive, and constantly changing. Selecting a subset by hand is difficult, time-consuming, and error-prone, suggesting that an automated solution is desirable.

Our thesis is that it is possible and practical to automate the process of choosing files to be stored on a portable computer. To validate this thesis, we conducted a preliminary study in a live business environment, which demonstrated that the approach was feasible.

We then developed a new metric, *semantic distance*, that quantifies the relationships among files, so that the group of files needed to work on a particular project can be identified. Using this metric, we built an automated system named Seer, which dynamically analyzes user behavior to identify the files needed for various projects, predicts the projects on which the user will be working, and then arranges to store the files necessary for these projects on the portable computer.

After building the system, we developed new metrics to characterize the behavior of automated hoarding systems, and deployed Seer among a small group of users. To our knowledge, ours is the first quantitative study of a hoarding system that has been done anywhere. The results of the study showed that Seer performed superbly, usually requiring only about a third of the hoard space needed by previous algorithms, and generally performing within a few percent of optimality. In live usage, Seer nearly always hoards 100% of the files needed by the user.

Acknowledgments

I would like to thank the members of my committee for their encouragement, help, and support; Dr. Peter Reiher for providing invaluable advice and motivation and for frequently acting as a surrogate dissertation adviser; my parents for fostering my lifelong fascination with science and for their many financial contributions; my wife Pat for standing beside me throughout the arduous process; Alyssa Gale for giving me purpose; Mary Ellen Fitzpatrick for always lifting my spirits; Shelby, Brad, and Patty Kuenning for caring about my success; Sue and Dave Bozman for their conviction that I could achieve this goal; Julissa Bozman for laughing at and with me; Eric Bozman for always believing in me; Cecile Marder for being my surrogate grandmother; Ashvin Goel, Michial Gunter, John Heidemann, David Ratner, and the other current and former members of the File Mobility Group for their endless discussions, suggestions, and patience with my ideas; Zhimei Jiang for her indispensable advice on the material in Chapter 3; Alex Bozman for helping me to express Theorem 3.4.10 concisely and Eric Postpischil for providing its proof; Carl Tait and Brian Noble for their help with Chapter 9; Drs. Charlie McDowell and Linda Werner for unsuccessfully warning me of what I was getting into and then supporting me throughout the process; Bob Stapleford for encouraging me to extend myself; Andy Hill and Renee Safier for keeping me entertained; Brad Spear for writing the software used for the studies described in Chapter 2; Andrew Louie for implementing the include investigator; and especially my adviser, Dr. Jerry Popek, for guiding me from the day I decided to pursue a Ph.D. until the day I completed it.

Chapter 1

Introduction

The best profit [sic] *of the future is the past.*—Chinese fortune cookie

Two conflicting trends in modern computing are the rise of networking and the advent of mobility. The lack of inexpensive, ubiquitous, high-bandwidth mobile communication has made portable computing inherently different from its fixed-base equivalent by denying access to networked information. The necessity for *disconnected operation* has led to a demand for systems designed to simplify this mode of use. Although there are a number of approaches to this problem, to date a complete solution has not been created.

1.1 Network-Free Mobility

In the last few years, mobile (laptop) computing has become a dominant force in the personal-computing market. At the same time, corporate and organizational computing have been moving in the direction of increased connectivity. Where even five years ago personal computers and workstations were often stand-alone machines, with limited file exchange performed by carrying floppies from desk to desk, the modern office features tens or even thousands of computers connected by a high-speed network to file servers, printers, databases, and timesharing services.

As the office worker moves more and more tasks onto his own computer, the importance of the network constantly increases. Where once a spreadsheet was created by manually entering printed information generated by a mainframe, "hot links" now automatically collect the latest data from the corporate accounting department. Where once a salesman took orders without being sure of availability, he now updates the online inventory system to reserve the requested items. But this same increased importance of the network is in direct conflict with the need for mobility. The modern salesperson wishes to enter orders directly on his laptop while sitting in the customer's office. The executive flying to a board meeting wants to have the latest profit figures to include in the report he is writing. Both of these workers have grown used to having on-line information and begin to feel frustrated and handicapped when it becomes unavailable simply because they have stepped away from their desks.

1.2 Mobile Data Availability

1.2.1 Communication

The obvious solution to the lack of mobile networking is to create mobile communications solutions, and many researchers are working to do just that. Wireless communication is one of the most active topics in computing today. Unfortunately, there are reasons to believe that we are a long way from a practical solution.

Wireless communication requires a significant and ubiquitous infrastructure that does not currently exist. It also requires nontrivial bandwidth, which is expensive in the context of a limited and non-expandable radio spectrum, and consumes a large amount of power when a mobile station transmits. Furthermore, there are always going to be places, such as airplanes, oceans, campgrounds, or underground structures, where it is difficult, expensive, or impossible to provide a high-bandwidth wireless connection.

An alternative solution to the lack of networking is to make the network completely unnecessary. Even in a large installation, the primary purpose of a network is to provide access to resources. If we can arrange for those resources to be locally available, then the network connection is superfluous and can be dispensed with, at least for a time.

1.2.2 Disk Space

However, most of the resources on the typical network are locally meaningful. In particular, shared data is the lifeblood of modern interconnected computing. A natural solution to the problem, then, is to simply store replicas of all the shared data, using a system such as FICUS [Guy 1991, Heidemann *et al.* 1992], RUMOR [Reiher *et al.* 1996], or CODA [Kistler and Satyanarayanan 1992].

This approach assumes that the portable machine has enough free space to store this data. In general, even with the huge disks that are becoming available in small form factors, this assumption will not be true. Throughout the history of computing, data has expanded to fill the available disk space. There is no reason to believe that this phenomenon will change. To the contrary, the advent of multimedia and ever-more-complex applications, with their associated graphics and configuration files, will only exacerbate the shortage of disk space.

Furthermore, even though portable disks are large, they are inherently incapable of storing as much data as can be kept on a non-portable machine [Satyanarayanan 1996], and the very existence of a network means that the total available data will be larger than can fit on any one disk. One need only browse the Internet for a few minutes to realize what an incredible amount of data is available to a networked machine.

1.2.3 Local Storage

However, there is really no need to keep *all* of the world's data stored on a portable computer. In any given day, the user accesses only a tiny fraction of what is available, and the portable machine must store only what is actually accessed. It is still possible for this data to exceed the capacity of the local disk (especially if the user is browsing the network or using large video or sound files), but most of the time everything will fit quite nicely. Using a system that supports *selective* replication [Ratner 1995], the user can store copies of only the files needed for a day's (or week's) work, propagating updates back to the network when a connection is available.

The problem of selectively replicating files on a small scale is well-understood and amenable to efficient solution. However, selective replication covers only part of the user's needs. It is one thing (albeit important) to be able to store copies of files in more than one place, propagate updates and changes, and detect conflicts. It is far more difficult to decide *which* files should be stored. This problem is especially important because, as discussed above, it will not always be possible to use wireless communications to retrieve an important file once a disconnected hoard miss has occurred.

1.2.4 Predictive Hoarding

We have built an automated system that can predict short-term file access patterns and use the predictions to make decisions about which files to cache, or *hoard*, on a mobile computer. This system works sufficiently well to allow successful mobile use, in the absence of any network connection, for several days or even weeks. The system is general enough to be applicable to a wide range of styles of mobile use, from the computer science researcher to the computer-illiterate corporate executive. The system operates transparently, with little or no user intervention required to control it.

As will be discussed in Chapter 7, the traditional cache performance measure of *miss ratio* is inappropriate for this application, so we have developed alternative measures for characterizing the performance of predictive hoarding schemes, and use these measures to evaluate both our system and those developed by others.

As part of the system, we have built a simple yet portable replication substrate to support the selective replication of the files that the predictive system has chosen for hoarding.

1.2.5 Alternative Solutions

Of course, predictive hoarding is not the only solution to the problem of storing files on the local disk. Several other alternatives are available for consideration.

Huge Disks

One obvious approach would be to simply equip the portable computer with an extra-large disk that is capable of storing any file in which the user may be interested. However, although this method would certainly simplify the problem of choosing files (by allowing more sloppiness), it does not solve the basic difficulty. As discussed above in Section 1.2.2, no portable disk is likely to be able to hold 100% of the data that might interest a user.[1] Since even the largest disk cannot hold all of the data in the world, this approach still does not address the problem of selecting *which* subset of the world's data will actually interest the user.

Hand Specification

Several previous researchers have solved the problem of choosing files by leaving it up to the user. For example, CODA [Kistler 1993] requires the user to build a list of all important files, together with numerical values that indicates their relative worth. FACE [Alonso *et al.* 1990] and LITTLE WORK [Honeyman *et al.* 1992, Huston and Honeyman 1993] are even more invasive, suggesting that immediately before disconnection the user must actually access all files that will be needed until the next time a connection is achieved (FACE also supports user specification in the manner of CODA, but with less flexibility).

Hand specification is at best a clumsy solution, and one acceptable only to the most knowledgeable and dedicated of computer users. In the first place, hand specification is inconvenient, distracting the user from the real work that is the point of using a portable computer. In the second, hand specification is difficult and error-prone. Many modern applications use files of which the user is completely unaware [Kuenning 1994]. Kistler's dissertation tells of a incident in which even expert computer scientists were unable to explain why a windowing system was sometimes unusable due to missing files [Kistler 1993, p. 193]. If a computer researcher cannot do the job accurately, it is certain that the average businessman will be helpless in the face of such an exacting task.

The "access-all-files" approach suffers from the added drawback that the user is expected to be able to predict and reproduce the exact manner in which he will use a particular application. Although it is nearly impossible in practice to hand-construct a list of files needed for a given task, it is at least true that once such a list exists, it will continue to work in the future. Not so for "hoarding by example." Woe betide the user who, just before leaving on a cross-country journey, neglects to access the only part of a particular document that uses the italic font. Once he is on the airplane, he may be completely unable to even display the page in question.

LRU Hoarding

Given that hand-specification methods are unworkable or at best tremendously inconvenient, one might still ask why a complex hoarding mechanism is necessary. Computer systems have been using the least-

[1] Users with minimal requirements may sometimes be able to fit into a relatively small disk footprint, but as application packages grow, they will remain the exception rather than the rule.

recently-used (LRU) cache-management method successfully for decades, usually with great success. Why not implement a LRU hoarding system for the portable computer and be done with it?

The answer lies in the complexity of user behavior and the cost of hoard misses. In traditional caching systems, an attempt to access an uncached object incurs only a small performance penalty. For many applications, a miss ratio or failure rate of 10% or even 20% of all accesses is considered acceptable. In disconnected operation, however, the cost of a hoard miss is nearly infinite. In experience with the CODA system, Kistler found that even a single hoard miss usually stopped work on the project, forcing the user either to switch to a secondary task, or to cease computing altogether.[2] Thus, an LRU-based system would be required to perform to a much higher standard than such algorithms have been able to achieve in the past.

In fact, we found that the requirements are so stringent that *no* LRU-based system can possibly perform adequately. Consider the previous example of a report that requires a special font for one of its sections. The user may work for days on a different section of the document, never touching the part that needs the second font. An LRU system would have no way of knowing that the font was needed.[3] We will discuss this question further in Section 8.2.2, p. 94.

Cluster-Based Hoarding

An ideal system would do more than just hoard the most recently referenced files. Instead, it should recognize the *projects* on which a user works, and hoard those projects as complete entities. By hoarding projects, the system will ensure that the user has everything he needs to get a particular job done while disconnected.

We have built a system called SEER, which automatically predicts the files a user will need while disconnected and ensures that they are hoarded. SEER is based on the concept of locating groups or *clusters* of files that are used together to work on a particular project. To locate such a cluster, SEER discovers the relationships among the cluster's member files by observing the user's actual behavior. The relationships are used to infer the cluster (project) members, and the project is hoarded as a unit.

1.3 Overview of the Dissertation

It is our thesis that it is both possible and practical to predict the files needed by a user, and to arrange for these files to be hoarded on a mobile computer before disconnection.

In Chapter 2 we describe a study of real-world user behavior that supports the hypothesis that predictive caching is a feasible method of managing data on a laptop computer. Chapter 3 introduces the concept of *semantic distance*, one of the fundamental ideas behind SEER. It formally defines relevant concepts, specifies algorithms to compute measures of distance, develops efficient approximation methods, and proves necessary underlying properties. Chapter 4 then discusses the clustering algorithms used to group related files for hoarding.

Chapter 5 discusses the design and implementation of our system. Since the real world is never as simple as one would like, Chapter 6 discusses difficulties that had to be addressed to make SEER work.

Chapter 7 discusses the methodology used to quantify SEER's success at predicting user behavior, and Chapter 8 gives the results of this analysis. In Chapter 9 we discuss the relationship of our work to other mobile-hoarding systems. Finally, Chapter 10 presents ideas for future directions based upon this research.

In Appendix A, we briefly describe the simple replication system we built to support SEER. Appendix B provides details on the internal design and implementation of the system.

[2] An example of the impact of hoard misses occurred during the early testing of our system, in which the essential `.cshrc` startup file was omitted from the hoard because no login had occurred for a long time. The user was almost completely denied use of the system after an unexpected reboot. The solution to this particular difficulty is outlined in Section 6.3, on page 65.

[3] LITTLE WORK and FACE are simple LRU systems, and CODA has a large LRU component underneath the user-specified lists of files. All three systems suffer from the problem of failing to hoard important, but recently unused, files. See Chapter 9 for details on these systems.

1.4 Summary of Results

A complete performance analysis of our system, in terms of both its impact on the user and its predictive success, is given in Chapter 8.

Overall, SEER predicts exceedingly well in the tests that were conducted. The predictions outperform LRU-style algorithms using only 20% to 50% of the hoard space needed by those methods. SEER generally needs hoard space that is only a few percent over the amount that would be required by an optimal algorithm. In live tests, most users experienced only one or two hoarding failures over several months, far fewer than the number that would be expected from an LRU-style approach. For all users, 95% or more of disconnections operated without any hoarding failure whatsoever.

Finally, under normal operation, the analysis indicates that SEER has a negligible effect on the overall performance of a modern laptop equipped with sufficient main memory.

Chapter 2

Feasibility[1]

The idea of automated prediction of user behavior is a seductive one, with applications far beyond simply controlling a mobile hoard. But is it actually possible in the real world, or is user behavior simply too complex to ever be predicted by a mere computer?

We cannot answer the latter question in the large, but we have strong reason to believe that the specific problem of predictive hoarding is solvable, even outside the university environment. This chapter presents a study undertaken in a business setting, which demonstrates that users behave in a consistent and tractable manner, so that predictive hoarding is indeed feasible.

2.1 Motivation

The research described in this chapter was undertaken to investigate the practicality of hoarding for mobility in a wide set of application domains. Our approach was to collect traces of file-access activity in several environments over a long period of time, and analyze them for feasibility and predictability of hoarding.

We chose to collect our own traces, rather than using existing traces, for three reasons. First, few existing traces are long enough. Because most existing traces collect read/write activity, a few weeks of data is sufficient to tax resource limits. We were interested in observing longer-term periodic behaviors such as end-of-the-month billing work in an accounting department, which therefore required a several-month trace to establish a pattern.

Second, existing traces have tended to be limited to an engineering application domain, usually programming. We wanted to investigate the behavior of non-programmers as well, in the twin beliefs that this type of user will eventually be the largest population of portable users, and that these users may behave quite differently from programmers.

Third, most previous studies have generally been limited to analyzing working-set sizes, or file-system performance data [Baker *et al.* 1991, Blaze and Alonso 1992, Kistler 1993, Ousterhout *et al.* 1985, Satyanarayanan *et al.* 1993]. The latter is not relevant to this research, and the former, while very important, is not in itself sufficient to characterize the user behaviors critical to successful mobile hoarding.

Successful automated hoarding requires two characteristics in user behavior:

- The working set of files, as observed over a period of days or weeks, must be small enough to fit on a portable's disk.
- It must be possible to predict the working set in advance, using hints such as the current working set, historical file access patterns [Tait and Duchamp 1991], or known patterns in user behavior.

Analysis of the data we have collected shows that these characteristics are present in a number of different application domains.

[1]Most of the material in this chapter appeared previously as [Kuenning *et al.* 1994].

2.2 Methodology

We collected our traces at Locus Computing Corporation, a software development and consulting firm, during the summer of 1993. One of Locus' products, PC/INTERFACE™ (PCI) [Locus 1993], is a pseudo-disk driver that makes the UNIX® file system available to MS-DOS® computers over an Ethernet. In the environments monitored, the local MS-DOS filesystem was used to store some applications software, but all shared corporate data was accessed via PCI. The UNIX server for PCI was modified to log opens, closes, and deletes of files. By avoiding read/write logging, we minimized the performance impact and kept the log files small. Log entries contain an operation type and subtype (*e.g.*, open for read), the UNIX timestamp in seconds, the UNIX UID of the invoker, the process ID, the absolute pathname of the file, and the size of the file.

Three different user environments were monitored. In the first, referred to as "personal productivity," the server was a machine that acted as the network filesystem for 47 users running business-oriented applications such as e-mail, project and calendar scheduling, and word processing. These users did not tend to store important files on their own machines, so they generated high activity at the server. This server was traced for 1563 hours (65.1 days, or 9.3 weeks),[2] recording 4,637,924 accesses.

In the second environment, referred to as "programming," the server was a cluster of 10 machines running IBM's Transparent Computing Facility, an adaption of the Locus distributed operating system [Popek and Walker 1985], which provides a single-system image to users of multiple machines. Each machine ran a separate PCI server, and logs from these servers were later combined for analysis. Most of the users of this server were programmers working on DOS-based software. Because they performed much of their work locally, accessing the shared server mostly to retrieve or update shared source files, they generated relatively little server activity. The traces on this server essentially reflect commits to a shared database, while omitting most localized file activity. This server was accessed by 64 users and was traced for 1693 hours (70.5 days, or 10.1 weeks), recording 93,719 accesses.

In the third environment, referred to as "commercial," the server was a single machine used by the accounting department to run a commercial accounting application. The master corporate accounting database was kept on the UNIX server, but all access to this (shared) database was via DOS workstations running the commercial package. This server was accessed by 7 users and was traced for 1257 hours (52.4 days, or 7.5 weeks), recording 371,830 accesses.

The nature of the traced environment (local files stored on PC's, with shared files stored remotely) parallels the expected behavior of mobile users, who will probably store heavily-used applications locally[3] but make extensive use of shared resources when they are network-connected. However, based on preliminary analysis of these traces, we also generated two modified traces that omitted certain characteristics we felt might be absent on portable platforms due to different software and user behaviors. For the commercial environment, we reduced all file sizes to a maximum of 1 Mb, on the theory that very large databases would be represented by smaller slices of the full database in a portable environment. This change primarily affected the statistics on working-set sizes and the amount of data involved in *write conflicts* and *attention shifts*, which are measures of file sharing and working-set variability that will be defined in Section 2.3 (p. 9). For the productivity environment, we eliminated all references to fax spooling and mail files, because such files are handled in a queued manner (as opposed to being shared) in disconnected environments. This change affected all of the statistics we analyzed. These two data sets are referred to as the "reduced commercial" and "reduced productivity" environments in the tables and graphs.

Once the traces were collected, we canonicalized them using a simple `awk` script that converts relative pathnames to absolute form, correlates each close with the corresponding open and produces an output line whose format is independent of the operation type, to make subsequent processing easier. These canonicalized files were then compressed and used as the basis for our analysis. The largest of these files (from the productivity server) is nearly 18 megabytes in its compressed form, and about 10 times that size when

[2] 50 days into this trace, there was a data gap of approximately 48 hours due to an administrative error. It does not appear that this gap affects the validity of the analysis.

[3] We expect that even active applications will eventually fall under the purview of an automated hoarding system.

expanded.

The subsequent analysis is performed in two phases. First, a single-pass program reads the data and extracts summary information of interest. (For example, for each 24-hour day in the collected data, the extraction program writes a single line for each user giving the total size of that user's working set, measured in both megabytes and files.) A second pass then analyzes these summary files with general-purpose statistical tools, generating the final tables and graphs presented in this paper.

2.3 Statistics

We generated the same statistics for each parameter in each environment: mean, standard deviation, and maximum. Besides the traditional measure of working-set size, we looked at two measures that have special application to mobility: *write conflicts* and *attention shifts*.

We define a write conflict event to occur when two users write to the same file within a relatively short time span. In a mobile environment, a conflicted file might be replicated on two or more computers, and the system would be required to automatically resolve these conflicts after the fact in a manner similar to the CODA or FICUS distributed file systems [Kumar and Satyanarayanan 1995, Reiher *et al.* 1994], to force the user to resolve them by hand [Kistler 1993], or to limit writing to only one user. We examined conflicting writes within a 24-hour period (corresponding to taking a machine home overnight) and a 7-day period (corresponding to traveling with a machine).

An attention shift occurs when a single user radically changes his or her working set. We identified attention shifts by looking at the working sets in successive active n-hour time periods (which did not necessarily represent adjacent days or weeks in cases where the user's machine remained idle, such as weekends). Within each time period, we counted the total numbers of files accessed, k_1 and k_2, and then calculated $k = \min(k_1, k_2)$. Within the second period, we also counted the total number m of files that had not been referenced during the first period, but that had existed prior to either period.[4] An attention shift was defined to occur if $m \geq pk$, where $0 \leq p \leq 1$. Attention shifts can be characterized by the parameters p, expressed as a percentage, and n, the number of hours in the period. We use the notation $p\%/n$ to describe an attention shift parameter pair. Based on a sensitivity analysis (see Figures 2.6–2.8), we chose $p = 20\%$. We chose $n = 24$ and $n = 168$ (1 week) because these represent typical disconnection periods for many portable users.

A final characteristic of an attention shift is the *age* of the shift, which represents the amount of time that has elapsed since the user last referenced one of the "new" files.[5] The age of an attention shift indicates when the files involved in the shift were last used. We estimated the age by locating the most recently-referenced "new" file (a file included in count m), and subtracting its reference time from the start time of the second period. This approximation tends to understate the age, since it assumes that the most-recently-referenced file is representative of the entire group m of "new" files. For our purposes this choice is the conservative one, since it assumes maximum effectiveness for a simple LRU-style hoarding system.

However, it was not always possible to find a file to use in calculating the age of the shift. This would happen when none of the "new" files had ever appeared before in the trace. In this case, we conservatively assumed that the "new" files had been referenced exactly one second before the beginning of the entire trace. Because of these two assumptions, the attention-shift ages reported in this paper are only a lower bound on the true ages that would be encountered by a predictive hoarding system.

The *bounded locality intervals* discussed in [Majumdar and Bunt 1986] are similar to attention shifts, but are parameterized on working-set sizes rather than on the expected length of a disconnection.

The statistics we report are:

Working-set statistics. For each day and week, we calculated the working set size in files, Mb, and number of accesses. Means and standard deviations were calculated by averaging data across time for each

[4] We eliminated files that were created during the second period because they are not problematical for a hoarding system that must predict which existing files need to be stored.

[5] The ages of successive attention shifts can represent overlapping periods.

	WS Size (Mb)			WS Size (Files)		
Environment	*Mean*	σ	*Max*	*Mean*	σ	*Max*
Productivity	1.0	(2.0)	134.5	39	(80)	3293
Reduced Productivity	0.7	(1.8)	41.1	7	(10)	547
Programming	0.3	(0.4)	18.0	10	(27)	2153
Commercial	18.2	(13.1)	65.0	294	(442)	1643
Reduced Commercial	10.9	(6.0)	33.6	294	(442)	1643

Table 2.1: Daily Working-Set Statistics

UID, and then calculating the mean and standard deviation across the per-UID means.

Attention-shift statistics. For each 1-day and 7-day attention shift, we examined the total size of the working set needed to hold both the old and the new data (in files and Mb). We also computed the per-user attention shift rate per day and per week. Finally, we calculated the age of each shift.

Conflict statistics. For each conflict, we examined the number of users involved in the conflict, and the size of the file on which the conflict occurred. We also calculated the per-user conflict rate per day and per week.

Success in mobile computing depends on small values for all of these statistics. Clearly, the working set must be small enough to fit comfortably on the typical portable's disk. The attention-shift rate should remain low, both so that the longer-period working set remains small and so that it is easier to predict the future working set based on recent behavior. The conflict rate must remain low to allow convenient file updates.

2.4 Analysis

The results of our analysis are very encouraging for an hoarding system. As hoped, working sets are small and attention-shift rates are low. Conflict rates are generally low, and it is clear how one might handle conflicts in the environments that had high conflict rates. However, attention-shift ages tend to be high, indicating that a predictive hoarding system needs to exercise significant intelligence to ensure that a portable computer is prepared for attention shifts.

Each table given below lists the mean for the statistic, followed by the standard deviation (in parentheses) and the maximum. For example, in Table 2.1, the mean daily working set for the productivity environment was 1.0 Mb, with a standard deviation of 2.0 Mb and a maximum of 134.5 Mb.

With the exception of Figures 2.6–2.8, all figures show the variation in a given measure over the duration of the trace. For example, Figure 2.1 shows the daily and weekly working sets for the productivity environment, for each day and each week captured during the trace.[6]

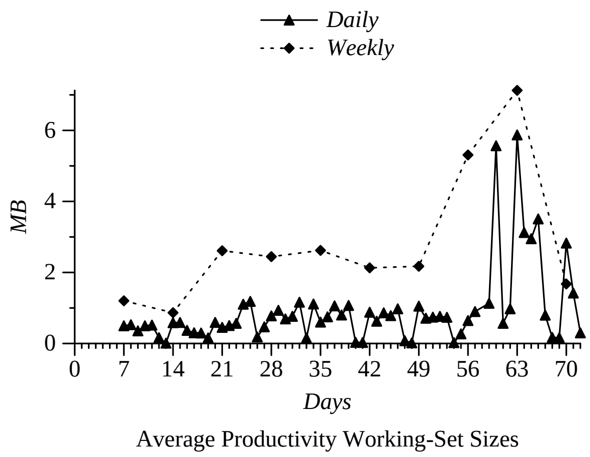
2.4.1 Working Sets

Tables 2.1 and 2.2 summarize the working-set sizes we observed. Figures 2.1–2.4 show the variation in mean and maximal working set sizes with time.

[6] In these and all other graphs, the lines connecting data points are present only to make it easier to see associated points, and are not meaningful in themselves. In particular, although the daily maxima in the right-hand sides of Figures 2.4 and 2.5 appear to exceed the weekly maxima, careful examination shows that only the connecting lines cross, and the actual data points for weekly maxima are always larger than the daily values.

	WS Size (Mb)			WS Size (Files)		
Environment	*Mean*	σ	*Max*	*Mean*	σ	*Max*
Productivity	2.7	(4.7)	148.4	110	(215)	3284
Reduced Productivity	1.4	(2.8)	43.6	19	(31)	548
Programming	0.6	(1.1)	18.3	22	(55)	2170
Commercial	26.8	(16.6)	65.7	374	(553)	1638
Reduced Commercial	16.8	(8.7)	33.8	374	(553)	1638

Table 2.2: Weekly Working-Set Statistics



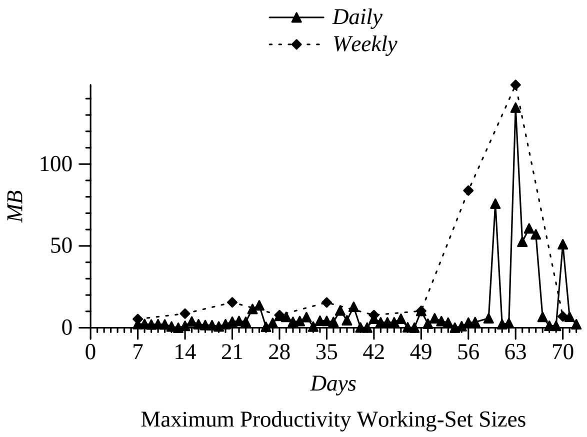
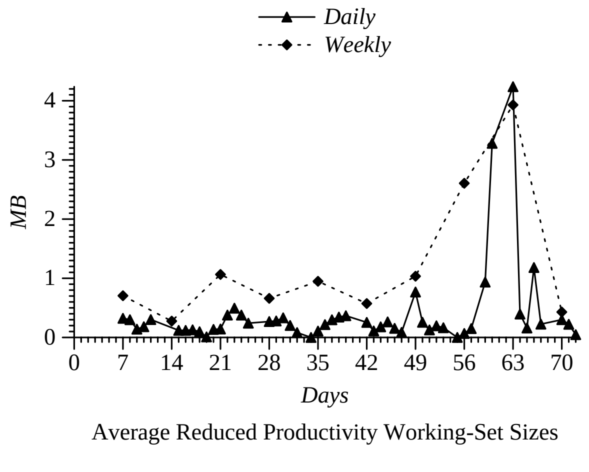


Figure 2.1: Working-Set Sizes for Productivity Environment



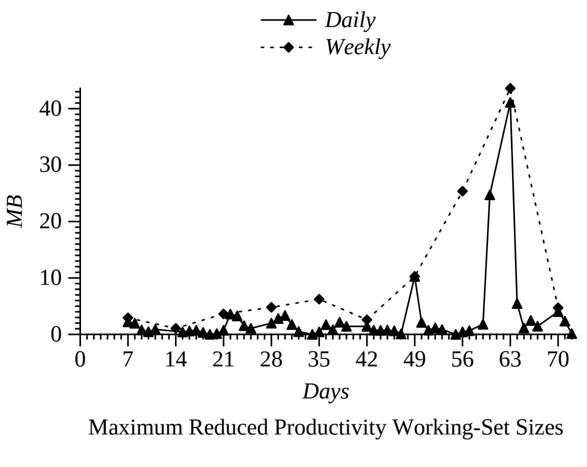


Figure 2.2: Working-Set Sizes for Reduced Productivity Environment

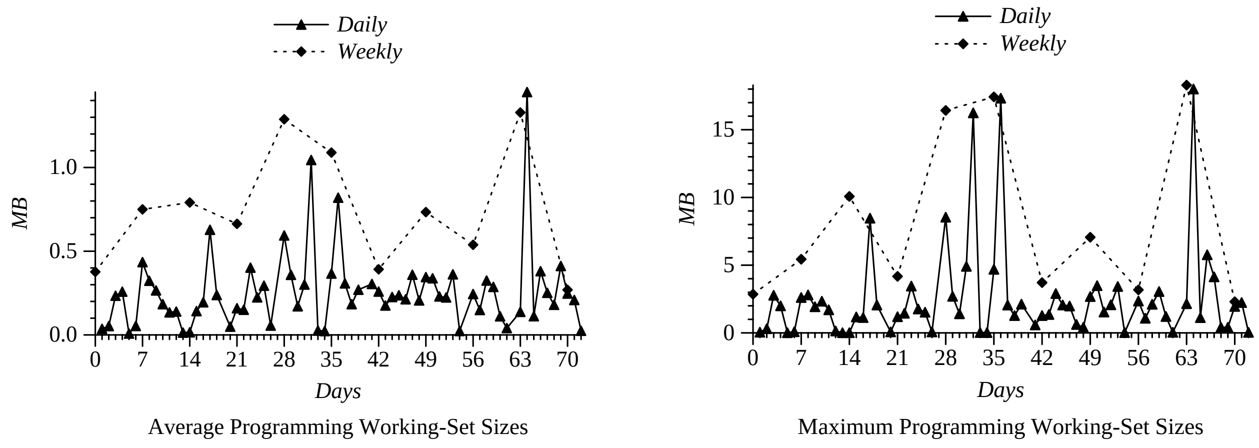


Figure 2.3: Working-Set Sizes for Programming Environment

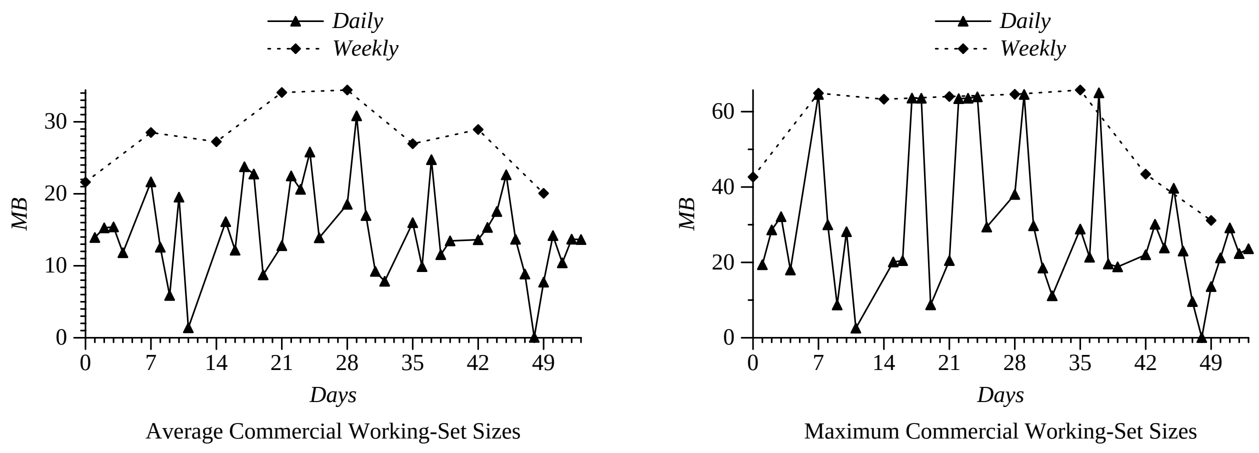
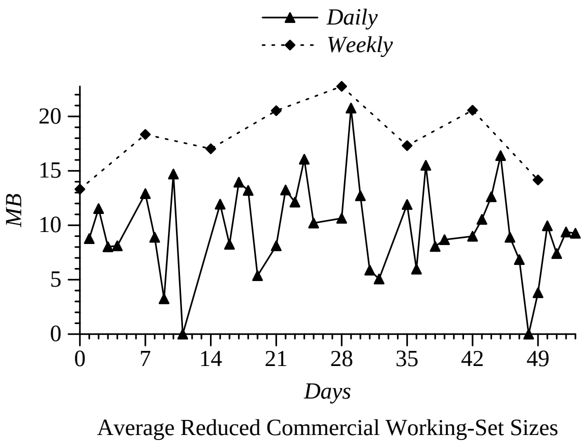


Figure 2.4: Working-Set Sizes for Commercial Environment



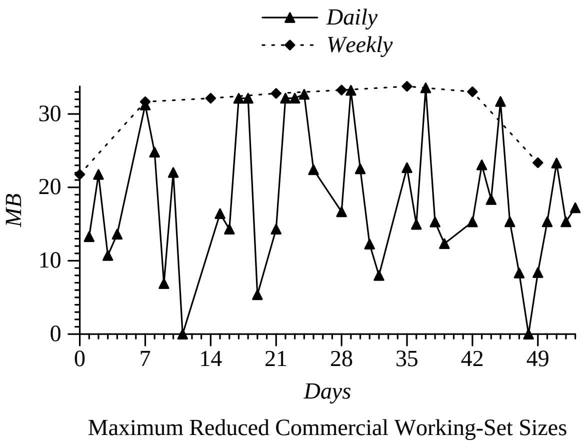


Figure 2.5: Working-Set Sizes for Reduced Commercial Environment

Mean working-set sizes tended to be small in all three environments, with the largest being about 18 Mb per day and 27 Mb per week, in the commercial environment. Maximal working sets were very large (148 Mb per week) only in the personal-productivity environment, apparently due to a single `grep`-style operation that occurred in week 9. This "`grep` phenomenon" is clearly visible in Figure 2.1. Eliminating this single maximum produced a secondary maximum of only 84 Mb. Maximal working sets in the other environments ranged only to 66 Mb.

These working-set figures indicate that it will be easy to store enough files on a portable disk to satisfy the average user,[7] although some software or user behavior may have to change when disconnected. (For example, instead of relying on a large `grep`, a user might use an inverted index to locate the files containing references to a particular string [Manber and Wu 1994].)

2.4.2 Attention Shifts

Tables 2.3 and 2.4 summarize the attention shifts observed. Figures 2.6–2.8 show the sensitivity of attention-shift rates to the parameter p. Except in the commercial environment, the number of attention shifts steadily decreases with increasing p, but the exact shape of the curve is quite inconsistent. In the absence of a clear-cut change in curvature (a knee or cliff), to guide us in the selection of p, we chose $p = 20\%$, which is near enough to the peak of the curves that we will not tend to underestimate the number of attention shifts, yet not so small that we will detect a shift every time a user accesses one or two new files.

Figures 2.9–2.11 show the variations in attention-shift rates with time, for $p = 20\%$. The amount of data involved in attention shifts was generally small (33 Mb or less), though the maxima were large (up to 152 Mb; the maxima follow from the size of the maximal working set and the definition of an attention shift). In all three environments, the number of attention shifts was surprisingly large and consistent, averaging up to 0.6 per user per week. This high rate has serious implications for a predictive hoarding scheme, because it shows that, as explained below, it is not sufficient to simply hoard the least recently used files. Instead, the system must be aware of attention shifts and store only the files that will actually be used.

However, because of the small size of the working sets involved in the average attention shift, a well-designed predictive hoarding system can afford to store both the old and the new set, so that attention shifts need not affect the usability of a mobile computer if the hoarding system is sufficiently accurate.

[7] We expect working-set sizes to change dramatically over the next few years as users move towards multimedia applications, but we also expect that disk sizes will increase sufficiently rapidly for portable computers to keep pace. In some sense, this phenomenon is self-regulating, since users will not tend to use images and sounds extensively if doing so would tax their portable storage capacity.

	Number Per User Per Day			Mb Involved			Files Involved			Age (Days)		
Environment	*Mean*	σ	*Max*	*Mean*	σ	*Max*	*Mean*	σ	*Max*	*Mean*	σ	*Max*
Productivity	0.4	(0.3)	0.8	1.6	(6.5)	135.7	64	(164)	3296	10.1	(15.8)	65.0
Reduced Productivity	0.2	(0.2)	0.5	0.8	(3.2)	41.1	13	(33)	548	26.5	(19.8)	65.0
Programming	0.3	(0.2)	0.5	0.6	(1.6)	20.9	16	(109)	2161	28.7	(21.5)	71.0
Commercial	0.3	(0.3)	0.9	21.8	(13.8)	65.7	217	(398)	1654	3.2	(4.6)	36.0
Reduced Commercial	0.3	(0.3)	0.9	14.6	(8.1)	33.8	217	(398)	1654	3.2	(4.6)	36.0

Table 2.3: 20%/24-Hour Attention Shifts (All Users)

	Number Per User Per Week			Mb Involved			Files Involved			Age (Days)		
Environment	*Mean*	σ	*Max*	*Mean*	σ	*Max*	*Mean*	σ	*Max*	*Mean*	σ	*Max*
Productivity	0.7	(0.2)	0.8	4.7	(12.4)	151.8	177	(376)	3423	15.8	(15.3)	63.0
Reduced Productivity	0.3	(0.1)	0.4	2.0	(5.5)	44.3	37	(71)	553	32.7	(18.9)	63.0
Programming	0.5	(0.1)	0.6	1.7	(3.4)	22.6	55	(215)	2174	30.1	(20.4)	70.0
Commercial	0.5	(0.4)	1.0	33.3	(17.4)	66.8	420	(584)	1661	11.4	(6.2)	35.0
Reduced Commercial	0.5	(0.4)	1.0	21.1	(9.0)	33.8	420	(584)	1661	11.4	(6.2)	35.0

Table 2.4: 20%/168-Hour Attention Shifts

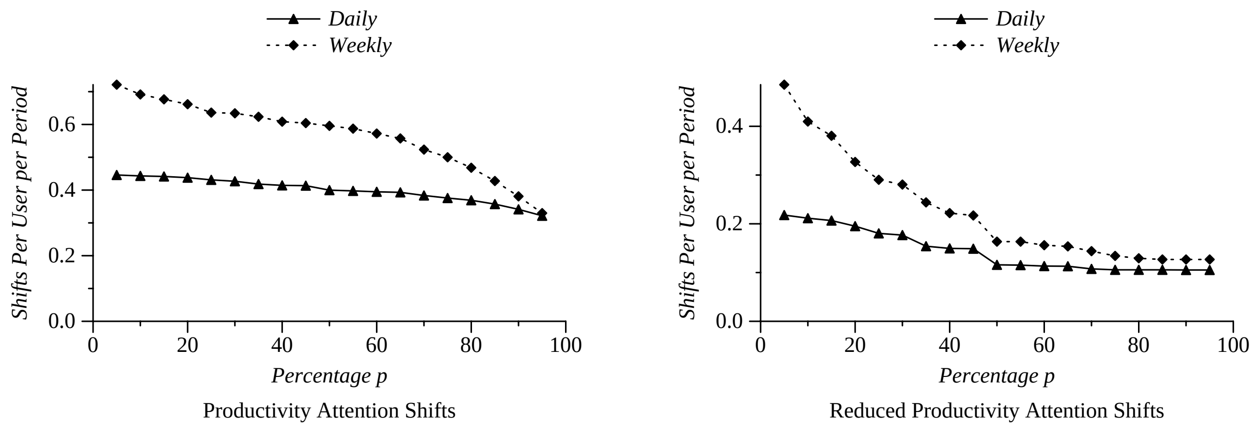


Figure 2.6: Attention-Shift Sensitivity for Productivity Environment

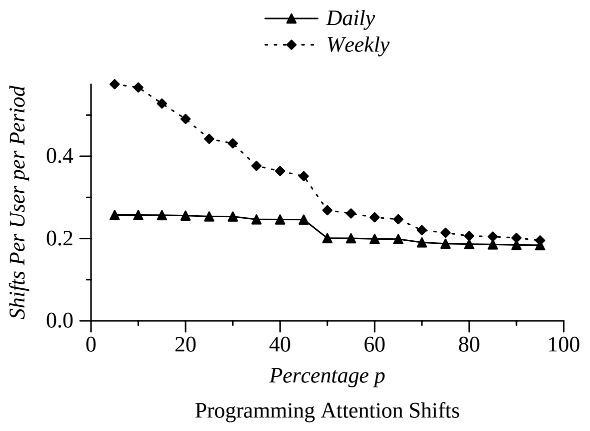


Figure 2.7: Attention-Shift Sensitivity for Programming Environment

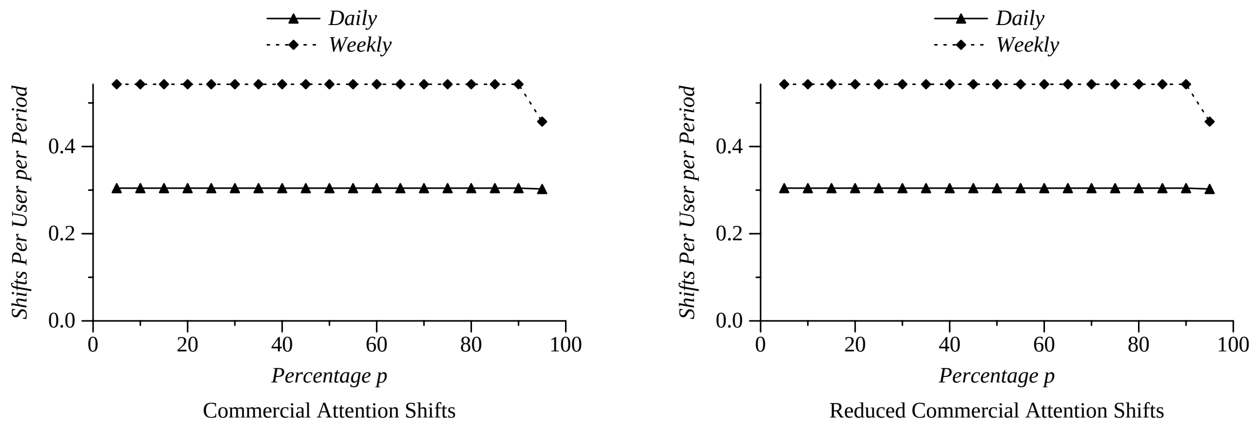
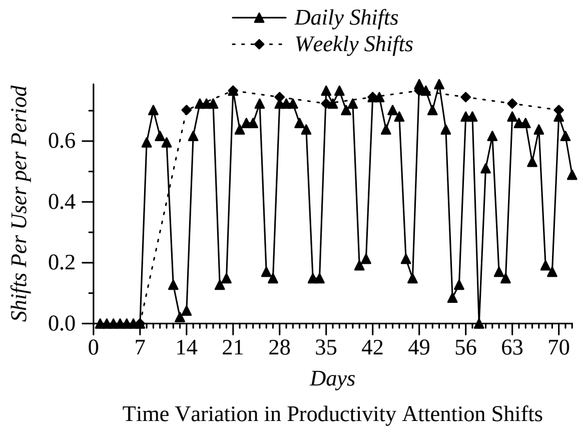


Figure 2.8: Attention-Shift Sensitivity for Commercial Environment



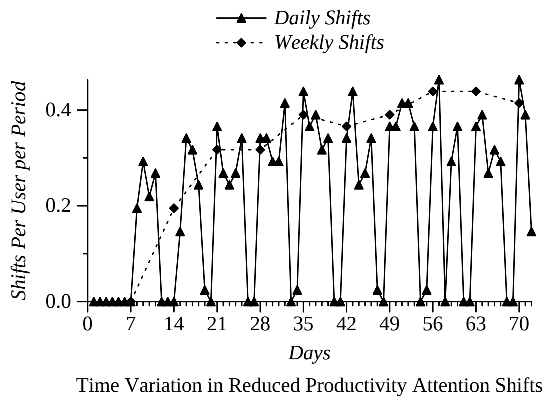


Figure 2.9: 20% Attention-Shift Rates for Productivity Environment

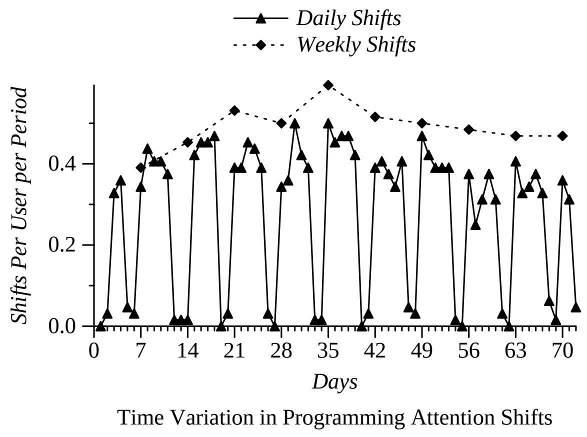
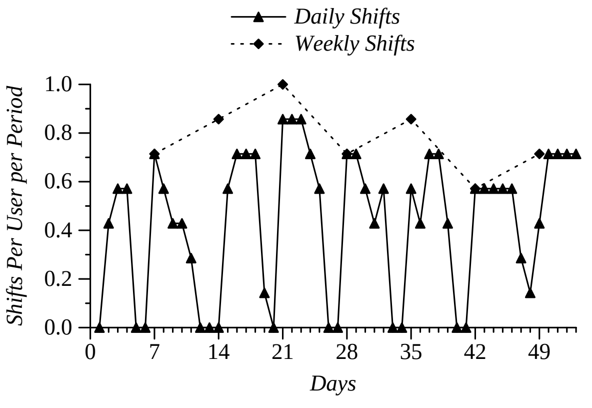
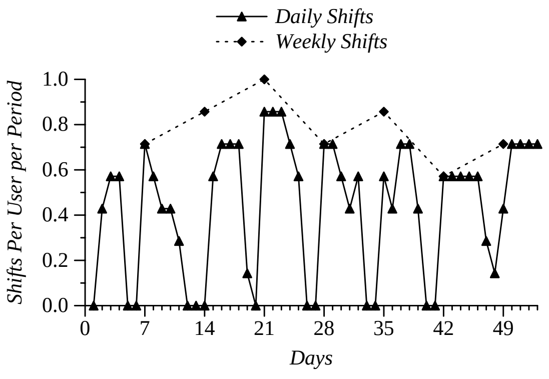


Figure 2.10: 20% Attention-Shift Rates for Programming Environment



Time Variation in Commercial Attention Shifts



Time Variation in Reduced Commercial Attention Shifts

Figure 2.11: 20% Attention-Shift Rates for Commercial Environment

	Daily Conflicts			Weekly Conflicts		
Environment	*Mean*	*σ*	*Max*	*Mean*	*σ*	*Max*
Productivity	1.20	(1.16)	4.28	6.08	(3.27)	10.11
Reduced Productivity	0.00	(0.01)	0.05	0.02	(0.03)	0.07
Programming	0.01	(0.02)	0.06	0.11	(0.09)	0.28
Commercial	4.35	(4.75)	16.29	12.32	(8.57)	24.57
Reduced Commercial	4.35	(4.75)	16.29	12.32	(8.57)	24.57

Table 2.5: Per-User Conflict Rates

Of course, if there is space to store both the old and new working sets, the question arises whether a simple LRU scheme would be sufficient to ensure that both working sets are available. The attention-shift age figures shown in Tables 2.3 and 2.4 belie this notion. For both the programming and the reduced productivity environments, the mean age of an attention shift is over 4 weeks and the maximum is near the length of the trace, indicating that an LRU hoard would very likely have been flushed by transient phenomena before the older files were re-referenced. This hypothesis is strengthened by the observation that the conservative method of estimating the ages of previously-unreferenced files, explained in Section 2.3, would produce a mean age of approximately half the length of the trace (about 5 weeks) if there were absolutely no historical data in the trace. In actuality, the new working set may not have been accessed for many months and thus may have been flushed from even a very lengthy LRU hoard. Other methods will be needed to ensure that a mobile machine will be prepared for an attention shift. The above data merely assures us that there will be room to store both today's and tomorrow's working sets once they have been identified.

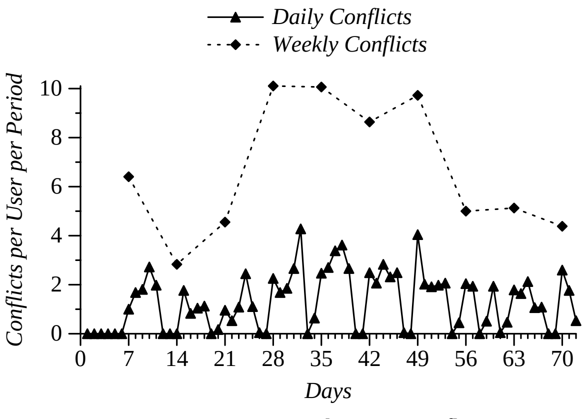
2.4.3 Conflicts

Tables 2.5 and 2.6 show statistics about conflicts and their rate of occurrence, respectively. Figures 2.12–2.14 show the variations in conflict rates with time. Conflicts were very rare in the "programming" environment, averaging 0.01 conflict per user per day, and only 0.11 per week. In nearly every case only two users were involved in a given conflict, although occasionally a third would write to the same file within 24 hours.

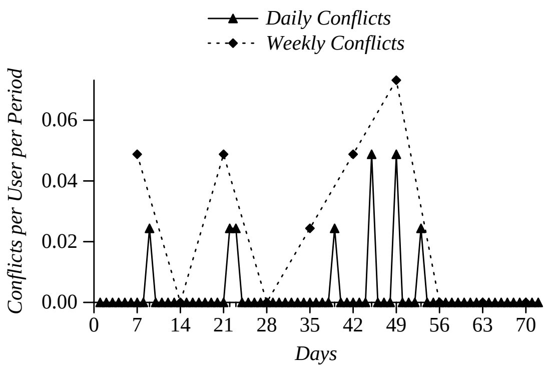
As expected, the 7 users of the "commercial" environment, with its shared accounting database, produced a high conflict rate of up to 25 per user per week, with up to 6 users writing to the same file in a single

Environment	Mb Involved in Daily Conflicts			Users Involved in Daily Conflicts			Mb Involved in Weekly Conflicts			Users Involved in Weekly Conflicts		
	Mean	σ	*Max*	*Mean*	σ	*Max*	*Mean*	σ	*Max*	*Mean*	σ	*Max*
Productivity	0.02	(0.08)	2.36	3.39	(3.06)	22.00	0.02	(0.09)	2.36	3.61	(3.62)	27.00
Reduced Productivity	0.04	(0.04)	0.12	2.00	(0.00)	2.00	0.04	(0.04)	0.12	2.00	(0.00)	2.00
Programming	0.07	(0.16)	1.08	2.02	(0.15)	3.00	0.06	(0.12)	1.08	2.09	(0.29)	3.00
Commercial	0.22	(0.81)	5.37	3.16	(1.10)	6.00	0.27	(0.83)	5.37	3.16	(1.28)	6.00
Reduced Commercial	0.17	(0.81)	5.37	3.16	(1.10)	6.00	0.20	(0.83)	5.37	3.16	(1.28)	6.00

Table 2.6: Conflicts

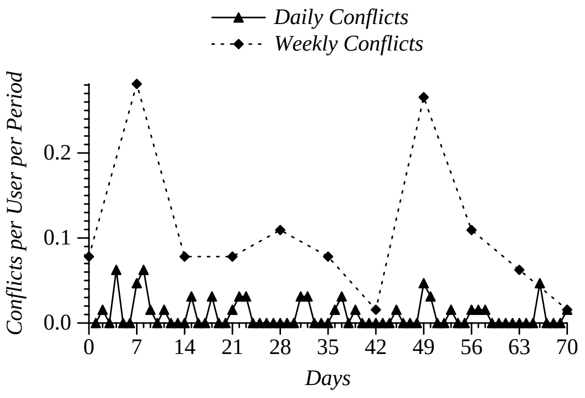


Time Variation in Productivity Conflict Rates



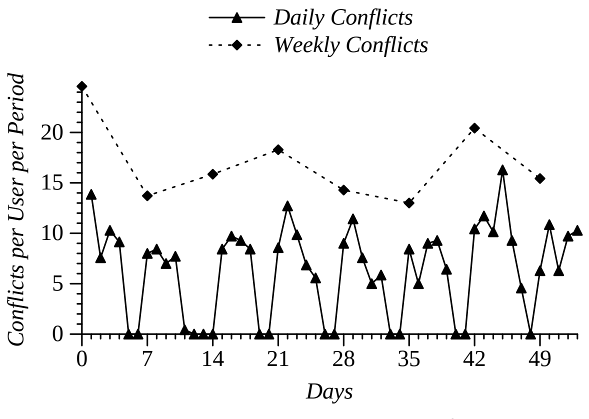
Time Variation in Reduced Productivity Conflict Rates

Figure 2.12: Conflict Rates for Productivity Environment

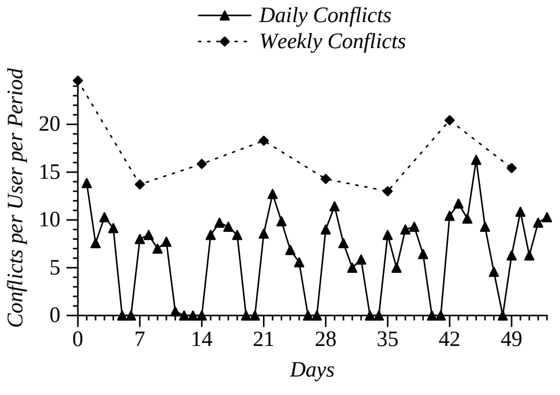


Time Variation in Programming Conflict Rates

Figure 2.13: Conflict Rates for Programming Environment



Time Variation in Commercial Conflict Rates



Time Variation in Reduced Commercial Conflict Rates

Figure 2.14: Conflict Rates for Commercial Environment

day. In a mobile environment, an automated resolver similar to those discussed in [Reiher *et al.* 1994] would be required to handle these numerous conflicts. Since accounting applications typically involve appending records to a transaction database, we expect that such a resolver would be easy to write.

The surprise was the "personal productivity" environment, which produced conflict rates averaging 1.2 per user per day, with up to 22 users writing to the same file in a single 24-hour period. We examined these conflicts in more detail to discover the cause, and found that nearly all of them involved mailboxes or fax-spooling files.

Since both mailbox and spooling files operate in a modified append-only mode (all but one user appends to the end of the file, and a simple locking mechanism prevents update while other file contents are modified), the conflicts on mail files do not present a problem for mobility. In fact, the retry-on-failure queuing algorithm of most mailers would handle mailbox conflicts with no software changes. In view of these observations, we generated the "reduced productivity" trace, which omitted these files from the statistics. With this change, the conflict rate dropped to a mean of only 0.02 per user per week, a number so small that it could conceivably be handled even without the help of automatic resolvers.

2.5 Conclusions

The data gathered and analysis performed in this study strongly indicated that predictive file hoarding would be a feasible approach to automating disconnected operation for mobile computers. However, the data also indicated that simple LRU hoarding would be insufficient. Based on this data, we proceeded with the design of a more complex algorithm, which is described in detail in the remainder of this dissertation.

Chapter 3

Semantic Distance

The preliminary studies described in Chapter 2 indicate that predictive hoarding is possible, but that simple LRU is not an adequate basis for a hoarding system. A successful hoarding system must be able to identify files that are of interest to the user *as a coherent group*. One way of doing so is for the system to be aware of semantic relationships among files.

In this chapter, we introduce the concept of *semantic distance*, a new measure that captures information about file relationships. We develop several variations of the measure, introduce algorithms to calculate it, and prove a number of theorems about the measure and the complexity of the algorithms.

3.1 Semantic Relationships Among Files

As mentioned in Section 1.2.5, p. 4, the SEER system is based on the concept of locating clusters of files that are used together to work on a particular project. To identify these clusters, SEER must discover the relationships among their member files. Since these relationships are defined by the semantics of the files (rather than by their names, size, location, or similar attributes), we call them *semantic relationships*.

Although it is relatively easy for an expert human to look at a set of files and decide whether they are semantically related, automating this task is much more difficult. One approach, which we will discuss further in Section 5.4 (p. 56), is to know something about the format and internal contents of a file. For example, a source file in a programming language often explicitly mentions the names of other files and libraries that it uses. A UNIX `makefile` is little more than an explicit list of the relationships among the various files used to build a program or other object, and hot links in WINDOWS® provide yet another hint about how a user perceives the importance and relationships among various files.

3.1.1 Inferred Semantic Relationships

Although the above techniques are important and useful, they are also *ad hoc* and thus inapplicable in the general case. In such situations, SEER must fall back on other methods. Since we do not wish to burden the user with the task of specification, the only choice is to infer the semantic relationships from other information.

There are a number of choices for such inferences. Directory organization is a powerful hint (see Section 4.2.2, p. 50), but it is often *only* a hint, because many users either overcrowd a directory (*e.g.*, putting all presentations into a single location, as is encouraged by software such as POWERPOINT®), or store complex projects in multiple directories.[1]

One of the few sources of information that is always available to an inferring program is the sequence of the user's references to files. If we could use this sequence to discover semantic interrelationships, we would

[1]As observed in [Tait *et al.* 1995], directory information is more useful for determining what is unrelated than for discovering close relationships.

have a powerful tool for discovering clusters and thus for controlling the hoard contents. SEER uses this clustering approach.

Time Sequences

When people work on a computer, they tend to do one task at a time. This behavior is an inherent side effect of the fact that humans do not have parallel streams of consciousness. Thus, it is natural to consider the sequence of file references from a standpoint of elapsed time. If a user refers to two files in quick succession, it is likely that those files are related, while if the files are accessed several hours apart, the relationship is likely to be more tenuous.

Although this idea is attractive, it also suffers from a number of drawbacks. One is the inconsistent nature of human time. If we are in the midst of developing a document and are interrupted by lunch or a lengthy telephone call, the files we access after the interruption are nevertheless related to the ones we used before. Similarly, the gap between quitting time and the next morning's start time is often relatively inconsequential from a semantic standpoint.

A second drawback is the difficulty of distinguishing the human from the computer time scale. Consider, for example, a compiler and an editor. The compiler accesses a number of source files within less than a second, while the editor, limited by human typing speed, may only refer to one or two files within an hour. Yet from the user's point of view, those one or two files may be as closely related as the many source files touched by the compiler. An automated system cannot easily discern that one program is behaving on its own while another is limited by human behavior, and thus it is very difficult to choose an appropriate time base to use when evaluating the relationships among references.[2]

A final difficulty is the inherent irregularity of time-based measures. The rate at which a compiler reads source files is affected by the system load and the size and complexity of those files, yet this rate has no bearing on the true relationships among them.

Reference Sequences

However, all is not lost if we reject time as a basis for inference. We may still be able to glean information from the ordered sequence of references. By eliminating all interval information, while keeping the sequencing, we can obviate most of the difficulties discussed above yet take advantage of the important insight that two files that are accessed together are more closely related than two accessed separately.

In the remainder of this chapter, we will elaborate the concept of a sequence of references, developing several different ways to infer inter-file relationships from the sequence, and will introduce practical algorithms that can be used to implement those inferences.

3.2 Reference Sequences and Derived Data

We begin by defining basic notation for the reference stream observed by SEER and for the fundamental semantic relationships implied by this stream.

3.2.1 Reference Sequences and Local Reference Distances

The most fundamental objects handled by SEER are files and references to them. Let $\mathcal{F} = \{f_1, f_2, \ldots f_F\}$ be the set of files in the system. Let the user's stream of references to files be $\mathcal{R} = \{r_1, r_2, \ldots : r_i \in \mathcal{F}\}$.[3] Each r_i is the name of a file, *e.g.*, f_i. (A given file f_a may be referenced multiple times by elements of $\mathcal{R}$.) Although the sequence $\mathcal{R}$ is unbounded, we will often consider a finite-time subset $\mathcal{R}_T = \{r_1, r_2, \ldots r_T\}$.

[2]It would be possible to use different time bases for different programs, but it is not clear how to specify the time base without human intervention, nor how one would deal with irregularities in human behavior, such as coffee breaks.

[3]On a multi-user system, this stream may be intermingled with references belonging to other users, or other tasks performed by a single user. We assume that a separate mechanism will separate these various references, so that our subsequent discussion only needs to deal with a single user's activity. Appendix B presents a method for separating references.

The *basic local reference distance* d_{ij} between two references r_i and r_j to distinct files is defined as:

$$d_{ij} = \begin{cases} j - i & \text{if } j > i \wedge r_i \neq r_j, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Example 3.2.1 Let

$$\mathcal{F} = \{f_1, f_2, f_3\}$$

and

$$\mathcal{R}_6 = \{f_1, f_2, f_1, f_3, f_2, f_3\}$$

Then the defined d_{ij} are:

$$d_{ij} = \begin{pmatrix} - & 1 & - & 3 & 4 & 5 \\ - & - & 1 & 2 & - & 4 \\ - & - & - & 1 & 2 & 3 \\ - & - & - & - & 1 & - \\ - & - & - & - & - & 1 \\ - & - & - & - & - & - \end{pmatrix}$$

3.2.2 Basic Distance Multisets

There is a local reference distance defined between every pair of references in $\mathcal{R}$. From the local reference distances, we select those related to a particular pair of files. We can further choose a subset of a file pair's distances, based on the positions of the various references. Each such choice will have a different effect on our final definition of semantic distance, giving greater or lesser weight to certain types of relationships.

For every distinct pair of files f_a, f_b where $f_a \neq f_b$, we define four multisets based on the local reference distance. First, the *basic all-pairs distance multiset* is defined as:

$$\mathcal{D}^*_{f_a,f_b} = \{d_{ij} : r_i = f_a \wedge r_j = f_b \wedge j > i\}$$

This captures all local reference distances, regardless of what separates them.

Example 3.2.2 Using the local reference distances from Example 3.2.1,

$\mathcal{D}^* =$

	To		
From	f_1	f_2	f_3
f_1		$\{d_{12}, d_{15}, d_{35}\}$	$\{d_{14}, d_{16}, d_{34}, d_{36}\}$
f_2	$\{d_{23}\}$		$\{d_{24}, d_{26}, d_{56}\}$
f_3	$\emptyset$	$\{d_{45}\}$	

$=$

	To		
From	f_1	f_2	f_3
f_1		$\{1, 4, 2\}$	$\{3, 5, 1, 3\}$
f_2	$\{1\}$		$\{2, 4, 1\}$
f_3	$\emptyset$	$\{1\}$	

Second is the *basic pairwise distance multiset*, which is restricted to the r_j most closely following each r_i:

$$\mathcal{D}^{\leftrightarrow}_{f_a,f_b} = \{d_{ij} : d_{ij} \in \mathcal{D}^*_{f_a,f_b} \wedge \forall k, i < k < j, r_k \neq f_a \wedge r_k \neq f_b\}$$

Example 3.2.3 Using the local reference distances from Example 3.2.1,

$$\mathcal{D}^{\leftrightarrow} = \begin{array}{r|ccc} & & \text{To} & \\ \text{From} & f_1 & f_2 & f_3 \\ \hline f_1 & & \{d_{12}, d_{35}\} & \{d_{34}\} \\ f_2 & \{d_{23}\} & & \{d_{24}, d_{56}\} \\ f_3 & \emptyset & \{d_{45}\} & \end{array}$$

$$= \begin{array}{r|ccc} & & \text{To} & \\ \text{From} & f_1 & f_2 & f_3 \\ \hline f_1 & & \{1,2\} & \{1\} \\ f_2 & \{1\} & & \{2,1\} \\ f_3 & \emptyset & \{1\} & \end{array}$$

Third is the *basic reference-to distance multiset* that collects all d_{ij} following each reference to f_a:

$$\mathcal{D}^{\rightarrow}_{f_a,f_b} = \{d_{ij} : d_{ij} \in \mathcal{D}^{*}_{f_a,f_b} \wedge \forall k, i < k < j, r_k \neq f_a\}$$

Example 3.2.4 Using the local reference distances from Example 3.2.1,

$$\mathcal{D}^{\rightarrow} = \begin{array}{r|ccc} & & \text{To} & \\ \text{From} & f_1 & f_2 & f_3 \\ \hline f_1 & & \{d_{12}, d_{35}\} & \{d_{34}, d_{36}\} \\ f_2 & \{d_{23}\} & & \{d_{24}, d_{56}\} \\ f_3 & \emptyset & \{d_{45}\} & \end{array}$$

$$= \begin{array}{r|ccc} & & \text{To} & \\ \text{From} & f_1 & f_2 & f_3 \\ \hline f_1 & & \{1,2\} & \{1,3\} \\ f_2 & \{1\} & & \{2,1\} \\ f_3 & \emptyset & \{1\} & \end{array}$$

The final multiset is the complementary *basic referenced-by distance multiset* that collects all d_{ij} preceding each reference to f_b:

$$\mathcal{D}^{\leftarrow}_{f_a,f_b} = \{d_{ij} : d_{ij} \in \mathcal{D}^{*}_{f_a,f_b} \wedge \forall k, i < k < j, r_k \neq f_b\}$$

Example 3.2.5 Using the local reference distances from Example 3.2.1,

$\mathcal{D}^{\leftarrow}$ =

	To		
From	f_1	f_2	f_3
f_1		$\{d_{12}, d_{15}, d_{35}\}$	$\{d_{14}, d_{34}\}$
f_2	$\{d_{23}\}$		$\{d_{24}, d_{56}\}$
f_3	$\emptyset$	$\{d_{45}\}$	

=

	To		
From	f_1	f_2	f_3
f_1		$\{1, 4, 2\}$	$\{3, 1\}$
f_2	$\{1\}$		$\{2, 1\}$
f_3	$\emptyset$	$\{1\}$	

For convenience in subsequent development, we will usually drop the subscripts f_a, f_b and simply use $\mathcal{D}^*$, etc. Unless otherwise specified, it is to be understood that all discussions refer to the distance multisets for a particular pair of files designated f_a and f_b. We use $\mathcal{D}$ to refer to an unspecified member of $\mathcal{D}^*, \mathcal{D}^{\leftrightarrow}, \mathcal{D}^{\rightarrow}, \mathcal{D}^{\leftarrow}$.

Clearly, $\mathcal{D}^{\leftrightarrow} \subseteq \mathcal{D}^{\rightarrow} \subseteq \mathcal{D}^*$ and $\mathcal{D}^{\leftrightarrow} \subseteq \mathcal{D}^{\leftarrow} \subseteq \mathcal{D}^*$.

Theorem 3.2.1 $\mathcal{D}^{\leftrightarrow} = \mathcal{D}^{\rightarrow} \cap \mathcal{D}^{\leftarrow}$.

Proof

$$\begin{aligned}
\mathcal{D}^{\rightarrow} \cap \mathcal{D}^{\leftarrow} &= \{d_{ij} : d_{ij} \in \mathcal{D}^* \wedge \forall k, i < k < j, r_k \neq f_a\} \\
&\quad \cap \{d_{ij} : d_{ij} \in \mathcal{D}^* \wedge \forall k, i < k < j, r_k \neq f_b\} \\
&= \{d_{ij} : d_{ij} \in \mathcal{D}^* \wedge \forall k, i < k < j, r_k \neq f_a \\
&\quad \wedge \forall k, i < k < j, r_k \neq f_b\} \\
&= \mathcal{D}^{\leftrightarrow}. \blacksquare
\end{aligned}$$

Theorem 3.2.2 $\mathcal{D}^* = \mathcal{D}^{\rightarrow} \cup \mathcal{D}^{\leftarrow}$.

Proof

$$\begin{aligned}
\mathcal{D}^{\rightarrow} \cup \mathcal{D}^{\leftarrow} &= \{d_{ij} : d_{ij} \in \mathcal{D}^* \wedge \forall k, i < k < j, r_k \neq f_a\} \\
&\quad \cup \{d_{ij} : d_{ij} \in \mathcal{D}^* \wedge \forall k, i < k < j, r_k \neq f_b\} \\
&= \{d_{ij} : d_{ij} \in \mathcal{D}^* \wedge (\forall k, i < k < j, r_k \neq f_a \\
&\qquad \vee \forall k, i < k < j, r_k \neq f_b)\} \\
&= \{d_{ij} : d_{ij} \in \mathcal{D}^* \\
&\quad \wedge (\forall k, i < k < j, r_k \neq f_a \vee r_k \neq f_b)\}
\end{aligned}$$

Since it is always true that either $r_k \neq f_a$ or $r_k \neq f_b$, the final condition is a tautology, and thus $\mathcal{D}^{\rightarrow} \cup \mathcal{D}^{\leftarrow} = \{d_{ij} : d_{ij} \in \mathcal{D}^*\} = \mathcal{D}^*$. ■

For each distance multiset, we define a shorthand for the cardinality:

$$N^* = |\mathcal{D}^*|, N^{\leftrightarrow} = |\mathcal{D}^{\leftrightarrow}|, N^{\rightarrow} = |\mathcal{D}^{\rightarrow}|, N^{\leftarrow} = |\mathcal{D}^{\leftarrow}|$$

We use N to refer to the cardinality of an unspecified multiset $\mathcal{D}$.

3.2.3 Lengthy References

In an actual system, file references do not take place at a single moment, but rather span a period of time. For example, the executable binary of a program is referenced only at the beginning of execution, yet the program may potentially remain active for days or even years. Files referenced by a program near the end of its lifetime are probably as closely related to it as those referenced near the beginning. Similarly, if we look at file opens and closes (as opposed to reads and writes, which for our purposes *are* instantaneous), a file has a finite lifetime during which many closely-related files may be referenced. This observation brings up the possibility of defining a more complex reference distance based on the lifetime of a file.

We will consider a new reference sequence $\mathcal{S}$ in which file opens and closes are considered as separate events, denoted s^o and s^c respectively, with s being used to denote a reference that can be either an open or a close. The elements of $\mathcal{S}$ will be numbered sequentially, as $s_1, s_2, \ldots$ We will write $s_i = s_j$ if s_i and s_j refer to the same file. In particular, $s^o_i = s^c_j$ if s^o_i and s^c_j refer to the same file, even though one is an open and one is a close. The number of elements of a particular type in a subsequence from s_i to s_j, inclusive, will be denoted N^o_{ij} or N^c_{ij}. Formally, these values are defined as $N^o_{ij} = |\{s^o_k : i \leq k \leq j\}|$ and $N^c_{ij} = |\{s^c_k : i \leq k \leq j\}|$. Of course, $N^o_{ij} + N^c_{ij} = j - i + 1$.

The number of open or close references to a particular file f_k in a subsequence from s_i to s_j, inclusive, is denoted $N^o_{ij;k}$ or $N^c_{ij;k}$, respectively. Formally, $N^o_{ij;k} = |\{s^o_m : i \leq m \leq j \wedge s^o_m = f_k\}|$, and similarly for $N^c_{ij;k}$. Note that $N^o_{1j;k} \geq N^c_{j;k}$ for all j.

Definition 3.2.1 The *lifetime local reference distance* between two opens of distinct files $s^o_i = f_a$ and $s^o_j = f_b$ is defined as:

$$\ell_{ij} = \begin{cases} 0 & \text{if } j > i \wedge N^o_{1j;a} > N^c_{1j;a} \wedge \nexists k : i < k < j \wedge N^o_{1k;a} = N^c_{1k;a} \\ j - i - N^c_{ij} & \text{if } j > i \wedge \exists k : i < k < j \wedge N^o_{1k;a} = N^c_{1k;a} \\ \text{undefined} & \text{otherwise} \end{cases}$$

In other words, the distance is zero if f_a is opened while f_a remains continuously open, and equal to the number of intervening open references (including the open of f_b) otherwise.

The four lifetime distance multisets are then defined as:

$$\begin{aligned} \mathcal{L}^*_{f_a,f_b} &= \{\ell_{ij} : s_i = f_a \wedge s_j = f_b \wedge j > i\} \\ \mathcal{L}^{\leftrightarrow}_{f_a,f_b} &= \{\ell_{ij} : \ell_{ij} \in \mathcal{D}^* \wedge \forall k, i < k < j, s_k \neq f_a \wedge s_k \neq f_b\} \\ \mathcal{L}^{\rightarrow}_{f_a,f_b} &= \{\ell_{ij} : \ell_{ij} \in \mathcal{D}^* \wedge \forall k, i < k < j, s_k \neq f_a\} \\ \mathcal{L}^{\leftarrow}_{f_a,f_b} &= \{\ell_{ij} : \ell_{ij} \in \mathcal{D}^* \wedge \forall k, i < k < j, s_k \neq f_b\} \end{aligned}$$

Much of our subsequent development will be concerned with $\mathcal{L}^{\leftrightarrow}$.

3.3 The Distance Histogram

Since the members of the distance multisets $\mathcal{D}$ and $\mathcal{L}$ are integers and may appear more than once, we can define a three-dimensional matrix $\mathcal{H}$ (resp. $\mathcal{J}$) that summarizes the important information in $\mathcal{D}$ (resp. $\mathcal{L}$). An element h_{ijk} of $\mathcal{H}$ is equal to the number of times $d_{ij} = k$ in $\mathcal{D}$, and similarly for $\mathcal{J}$. Formally, $h_{ijk} = |d_{ij}| : d_{ij} = k$. We call the matrices $\mathcal{H}$ and $\mathcal{J}$ the *basic distance histogram* and *lifetime distance histogram*, respectively. Of course, there are multiple versions of these matrices, corresponding to the multiple ways of calculating $\mathcal{D}$ and $\mathcal{L}$.

To avoid unnecessary clutter, we will refer only to the basic distance histogram $\mathcal{H}$ in this section, with the understanding that the same development applies equally well to $\mathcal{J}$ unless otherwise specified.

The distance histogram is useful because it provides a unified method for understanding various approaches to analyzing reference streams. For example, the cardinality of a distance multiset is equal to the sum of the appropriate elements of $\mathcal{H}$ along the third dimension:

$$N_{ij} = \sum_k h_{ijk}$$

Since the distance histogram $\mathcal{H}$ is very large, methods for compressing and summarizing the information it contains are very attractive. We will review a previous approach to summarizing the information in $\mathcal{H}$ before presenting our own methods.

3.3.1 Appleton's Probabilities

Appleton [Griffioen and Appleton 1994] has defined a set of probabilities characterizing which file is likely to be accessed soon, where "soon" is a parameter of the algorithm. In terms of the pairwise distance histogram $\mathcal{H}^{\leftrightarrow}$, Appleton's probabilities are defined as follows:

$$p_{ij} = \frac{\sum_{k \leq L} h^{\leftrightarrow}_{ijk}}{\sum_{j} \sum_{k \leq L} h^{\leftrightarrow}_{ijk}}$$

where L is the *lookahead* of the algorithm, representing the distance into the future for which predictions are desired. The value p_{ij} represents an estimate of the *local* probability that file j will be referenced within L accesses of file i. Since Appleton does not store zero values of p_{ij}, and since the value of L necessarily limits the number of p_{ij} that must be stored, Appleton's method requires much less space than that required to store $\mathcal{H}$ itself. This space efficiency is a great strength of Appleton's method.

A simple extension of Appleton's method would be to keep more than one plane of $\mathcal{H}^{\leftrightarrow}$, so that p_{ij} could be easily calculated for different values of L. If all planes up to a given L_{MAX} were stored, then the storage required would be only a constant multiple of that needed to store local probabilities for $L = L_{\mathrm{MAX}}$.

3.3.2 Arithmetic Mean Reference Distances

A simple way to summarize the elements of $\mathcal{H}$ is to take the arithmetic mean along the third dimension.

The *arithmetic mean* {*all-pairs, pairwise, reference-to, referenced-by*} *reference distance* between two files is denoted by $\{{}^{a}D^{*}, {}^{a}D^{\leftrightarrow}, {}^{a}D^{\rightarrow}, {}^{a}D^{\leftarrow}\}$ and is calculated as the arithmetic mean of the appropriate multiset:

$$\begin{aligned}
{}^{a}D^{*} &= \frac{1}{N^{*}} \sum_{d \in \mathcal{D}^{*}} d \\
{}^{a}D^{\leftrightarrow} &= \frac{1}{N^{\leftrightarrow}} \sum_{d \in \mathcal{D}^{\leftrightarrow}} d \\
{}^{a}D^{\rightarrow} &= \frac{1}{N^{\rightarrow}} \sum_{d \in \mathcal{D}^{\rightarrow}} d \\
{}^{a}D^{\leftarrow} &= \frac{1}{N^{\leftarrow}} \sum_{d \in \mathcal{D}^{\leftarrow}} d
\end{aligned}$$

We use ${}^{a}D$ to refer to an unspecified (arithmetic) mean reference distance, and we also define a similar measures ${}^{a}L$ for the lifetime distance multiset $\mathcal{L}$.

Example 3.3.1 Using the multisets from Examples 3.2.2 through 3.2.5, it is easy to calculate:

${}^{a}D^{*}$	To		
From	f_1	f_2	f_3
f_1		$\frac{7}{3}$	3
f_2	1		$\frac{7}{3}$
f_3		1	

${}^{a}D^{\leftrightarrow}$	To		
From	f_1	f_2	f_3
f_1		$\frac{3}{2}$	1
f_2	1		$\frac{3}{2}$
f_3		1	

${}^{a}D^{\rightarrow}$	To		
From	f_1	f_2	f_3
f_1		$\frac{3}{2}$	2
f_2	1		$\frac{3}{2}$
f_3		1	

${}^{a}D^{\leftarrow}$	To		
From	f_1	f_2	f_3
f_1		$\frac{7}{3}$	2
f_2	1		$\frac{3}{2}$
f_3		1	

In terms of the distance histogram $\mathcal{H}$, these distances can be calculated as a weighted arithmetic mean:

$$^{a}D_{ij} = \frac{\sum_k k h_{ijk}}{\sum_k h_{ijk}}$$

and similarly for the variant measures.

3.3.3 Geometric Mean Reference Distances

There is nothing special about the summarizing with the arithmetic mean; other mathematical summary methods are also possible. In fact, the arithmetic mean has certain drawbacks for our purposes. For example, consider two pairwise multisets, $\mathcal{D}^{\leftrightarrow}_{f_a,f_b} = \{1, 1, 1498\}$ and $\mathcal{D}^{\leftrightarrow}_{f_c,f_d} = \{500, 500, 500\}$. Using the arithmetic mean, $^{a}D^{\leftrightarrow}_{f_a f_b} = {}^{a}D^{\leftrightarrow}_{f_c f_d} = 500$, yet intuitively we would say that f_a and f_b are much more likely to be related than f_c and f_d.[4]

A simple solution to this difficulty is to use the geometric mean, which gives greater weight to small values than large ones. For example, the geometric mean of the first multiset above is 11.44, while the second is 500.

The *geometric mean* {*all-pairs, pairwise, reference-to, referenced-by*} *reference distance* between two files is denoted by $\{^{g}D^{*}, {}^{g}D^{\leftrightarrow}, {}^{g}D^{\rightarrow}, {}^{g}D^{\leftarrow}\}$ and is calculated as the geometric mean of the appropriate multiset.

$$\begin{aligned}
^{g}D^{*} &= \sqrt[N^{*}]{\prod_{d \in \mathcal{D}^{*}} d} \\
^{g}D^{\leftrightarrow} &= \sqrt[N^{\leftrightarrow}]{\prod_{d \in \mathcal{D}^{\leftrightarrow}} d} \\
^{g}D^{\rightarrow} &= \sqrt[N^{\rightarrow}]{\prod_{d \in \mathcal{D}^{\rightarrow}} d} \\
^{g}D^{\leftarrow} &= \sqrt[N^{\leftarrow}]{\prod_{d \in \mathcal{D}^{\leftarrow}} d}
\end{aligned}$$

For most purposes, the geometric mean can be replaced by the arithmetic mean of the logarithms of the distances, since we normally use the mean distances only in comparison with each other, and never make direct use of the individual values.

We use ^{g}D to refer to an unspecified (geometric) mean reference distance. For the lifetime distance multiset $\mathcal{L}$, the measure ^{g}L must be defined in a slightly different manner, because ℓ can be zero, and a single zero ℓ would make the geometric mean zero as well. To avoid multiplying by zero, we increment ℓ in defining the geometric mean lifetime distance:[5]

$$\begin{aligned}
^{g}L^{*} &= \sqrt[N^{*}]{\prod_{\ell \in \mathcal{D}^{*}} (1+\ell)} \\
^{g}L^{\leftrightarrow} &= \sqrt[N^{\leftrightarrow}]{\prod_{\ell \in \mathcal{D}^{\leftrightarrow}} (1+\ell)} \\
^{g}L^{\rightarrow} &= \sqrt[N^{\rightarrow}]{\prod_{\ell \in \mathcal{D}^{\rightarrow}} (1+\ell)}
\end{aligned}$$

[4]This example was first suggested by Michial Gunter [Gunter 1995].

[5]This modification does not affect the validity of the measure, since for our purposes we are interested in relative, rather than absolute, values.

$$ {}^{g}L^{\leftarrow} = \sqrt[N^{\leftarrow}]{\prod_{\ell \in \mathcal{D}^{\leftarrow}} (1+\ell)} $$

In terms of the distance histogram $\mathcal{H}$, these distances can be calculated as a weighted geometric mean:

$$ {}^{g}D_{ij} = \sqrt[K]{\prod_{k} (h_{ijk})^{k}} $$

where $K = \sum_k h_{ijk}$, and similarly for the variant measures.

3.3.4 Algorithms

Observation 3.3.1 *Any algorithm that calculates any of the distance measures ${}^{a}D$, ${}^{a}L$, ${}^{g}D$, or ${}^{g}L$ for all pairs of files requires $\Omega(F^2)$ space, where F is the number of files. Any algorithm that calculates a distance measure between a single file and all other files must use at least $\Omega(F)$ space. Any algorithm that calculates a distance measure between a single pair of files requires $\Omega(1)$ space. All of these lower bounds hold regardless of the method of calculation, because that much space is needed to hold the results. Algorithm 3.3.1 will demonstrate that these are also upper bounds for calculating all arithmetic mean distances and the two geometric mean distances ${}^{g}D^{\leftrightarrow}$ and ${}^{g}D^{\rightarrow}$ for the basic distance multiset, and Algorithm 3.3.3 will develop the same result for the lifetime distance measure.*

Observation 3.3.2 *Any algorithm that calculates any of the distance measures ${}^{a}D$, ${}^{a}L$, ${}^{g}D$, or ${}^{g}L$ requires $\Omega(T)$ time.*

Algorithm 3.3.1 ***Simultaneous calculation of the measures ${}^{a}D^{\leftrightarrow}$, ${}^{a}D^{\rightarrow}$, ${}^{a}D^{\leftarrow}$, ${}^{a}D^{*}$, ${}^{g}D^{\leftrightarrow}$, and ${}^{g}D^{\rightarrow}$.*** *(Equivalent C-like pseudocode is given in Algorithm 3.3.2.)*

The algorithm is constructed as two nested loops. The outer loop iterates once for each referenced file, referred to as f_2. The inner loop is executed once for each known file, which will be called f_1 in the exposition.

The algorithm makes use of a triple for each known file, and a 12-tuple for each pair of files. The per-file triple is denoted $\Gamma = <t, n_r, \sigma_r>$, where t is the index of the most recent reference to the file, n_r is the total number of references to the file, and σ_r is the total of all past values of t. The first value is used to decide when a reference should be included in $\mathcal{D}^{\leftrightarrow}$. The latter two quantities are used in the calculation of ${}^{a}D^{\leftarrow}$ and ${}^{a}D^{*}$.

The per-pair 12-tuple is denoted

$$ \Theta = <n^{*}, \sigma^{*}, \pi^{*}, n^{\leftrightarrow}, \sigma^{\leftrightarrow}, \pi^{\leftrightarrow}, n^{\rightarrow}, \sigma^{\rightarrow}, n^{\leftarrow}, \sigma^{\leftarrow}, n^{f_1}, \sigma^{f_1}> $$

where the "n" elements are used to count the number of elements in the corresponding distance multiset, the "σ" elements are used to sum the elements, and the "π" elements are used to accumulate the product of the distance multiset. The final two elements (those superscripted f_1) are used in calculating $\mathcal{D}^{\leftarrow}$. n^{f_1} counts the total number of references to the first file of the pair since the occurrence of the most recent reference to the second file. σ^{f_1} sums the times of reference (subscript of r) to the first file.

At any time, the relevant arithmetic mean distances may be calculated as the quotient of a "σ" element divided by the corresponding "n" element, and the geometric mean distances may be calculated by taking the appropriate "n"th root of a "π" element.

All elements of all tuples are initialized to zero, except "π" elements, which are initialized to 1.

For convenience in exposition, we will abbreviate identification of members of the tuples. Unless otherwise specified, all references to elements of Γ will be understood to be elements of Γ_{f_2} (not Γ_{f_1}), and all references to elements of Θ will be understood to be elements of Θ_{f_1,f_2}. Thus, for example, n^r refers to $n^r \in \Gamma_{f_2}$ and σ^{*} is $\sigma^{*} \in \Theta_{f_1,f_2}$, but t_{f_1} refers to $t \in \Gamma_{f_1}$.

The algorithm:

```
for $r_i \in \mathcal{R}$
    $f_2 := r_i$
    for $f_1 \in \mathcal{F} \wedge f_1 \neq f_2$
        $d := i - t_{f_1}$
        if $t < t_{f_1}$
            $n^{\leftrightarrow} := n^{\leftrightarrow} + 1$
            $\sigma^{\leftrightarrow} := \sigma^{\leftrightarrow} + d$
            $\pi^{\leftrightarrow} := d\pi^{\leftrightarrow}$
        $n^{\rightarrow} := n^{\rightarrow} + 1$
        $\sigma^{\rightarrow} := \sigma^{\rightarrow} + d$
        $\pi^{\rightarrow} := d\pi^{\rightarrow}$
        $n^{*} := n^{*} + n^{r}$
        $\sigma^{*} := \sigma^{*} + n^{r} i - \sigma^{r}$
        $n^{\leftarrow} := n^{\leftarrow} + n^{f_1}$
        $\sigma^{\leftarrow} := \sigma^{\leftarrow} + n^{f_1} i - \sigma^{f_1}$
        $n^{f_1} := 0$
        $\sigma^{f_1} := 0$
    for $f_1 \in \mathcal{F} \wedge f_1 \neq f_2$
        $n^{f_1} := n^{f_1} + 1$
        $\sigma^{f_1} := \sigma^{f_1} + i$
    $n^{r} := n^{r} + 1$
    $\sigma^{r} := \sigma^{r} + i$
    $t := i$
```

Algorithm 3.3.2 ***Simultaneous calculation of the measures $^{a}D^{\leftrightarrow}$, $^{a}D^{\rightarrow}$, $^{a}D^{\leftarrow}$, $^{a}D^{*}$, $^{g}D^{\leftrightarrow}$, and $^{g}D^{\rightarrow}$.*** *(A formal version is given in Algorithm 3.3.1.)*

```
/*
 * C-like pseudocode to calculate distance multisets.  For
 * simplicity of exposition, we ignore initialization and the code
 * needed to deal with not-yet-referenced files.
 *
 * To avoid exceeding floating-point ranges, we calculate the
 * product values as sums of logarithms.  At any time, the
 * arithmetic mean distances can be calculated as the quotient of
 * the appropriate 'Sum' variable divided by the corresponding 'n'
 * variable; the logarithms of the geometric means can be
 * calculated by dividing the appropriate 'Prod' variable by the
 * corresponding 'n' variable.  The actual geometric mean can be
 * determined by exponentiation if desired.
 */
struct PerFile {
  int lastRefTime;  /* Last time referenced */
  int nRefs;        /* Total number of references */
  int sumRefs;      /* Sum of all reference times */
} perFile[NFILES];
struct PerPair {
  int nPairwise;    /* Number of pairwise distances */
  int pairwiseSum;  /* Sum of pairwise distances */
  double pairwiseProd; /* Log product of pairwise distances */
  int nRefTo;       /* Number of reference-to distances */
  int refToSum;     /* Sum of reference-to distances */
  double refToProd; /* Log product of ref-to distances */
  int nRefBy;       /* Number of reference-by distances */
  int refBySum;     /* Sum of reference-by distances */
  int nAllPairs;    /* Number of all-pairs distances */
  int allPairsSum;  /* Sum of all-pairs distances */
  int nF1Refs;      /* Number of references to f1 since f2 */
  int f1Sum;        /* Sum of times for nF1Refs */
} perPair[NFILES][NFILES];
  for (<r = every reference>) {
    f2 = file referenced
    reftime = time of reference /* (index) */
    for (<f1 = every file>) {
      /* Note that f1 is the earlier-referenced file */
      if (f1 == f2)
        continue;
      refdist = reftime - perFile[f1].lastRefTime;
      /*
       * Calculating pairwise.  We only want to
       * contribute to the mean if this is the first
       * reference to f2 since the last reference to f1.
       */
      if (perFile[f2].lastRefTime < perFile[f1].lastRefTime) {
        perPair[f1][f2].nPairwise++;
        perPair[f1][f2].pairwiseSum += refdist;
        perPair[f1][f2].pairwiseProd += log(refdist);
      }
      /*
       * Calculating refto.  Just sum things up.  This
       * will automatically catch every reference to f2
       * after the most recent reference to f1.
       */
      perPair[f1][f2].nRefTo++;
      perPair[f1][f2].refToSum += refdist;
      perPair[f1][f2].refToProd += log(refdist);
      /*
       * Calculating allpairs.  For every reference time
       * to f1, t, from the beginning of time, we want
       * to add in (reftime - t).  We do this by
       * distributing the multiplication.
       */
      perPair[f1][f2].nAllPairs += perFile[f1].nRefs;
      perPair[f1][f2].allPairsSum += perFile[f1].nRefs * reftime;
      perPair[f1][f2].allPairsSum -= perFile[f1].sumRefs;
      /*
       * Calculating refby.  For every reference time t
       * from the last time f2 was referenced, we want
       * to add in (reftime - t).  We do this by
       * distributing the multiplication.
       */
      perPair[f1][f2].nRefBy += perPair[f1][f2].nF1Refs;
      perPair[f1][f2].refBySum +=
        perPair[f1][f2].nF1Refs * reftime - perPair[f1][f2].f1sum;
      perPair[f1][f2].nF1Refs = 0;
      perPair[f1][f2].f1Sum = 0;
    }
    /*
     * Update the sums used in calculating refby and allpairs.
     */
    for (<f1 = every file>) {
      if (f1 == f2)
        continue;
      perPair[f2][f1].nF1Refs++;
      perPair[f2][f1].f1Sum += reftime;
    }
    perFile[f2].nRefs++;
    perFile[f2].sumRefs += reftime;
    perFile[f2].lastRefTime = reftime;
  }
```

Algorithm 3.3.3 ***Simultaneous calculation of*** $^aL^{\leftrightarrow}$***,*** $^aL^{\rightarrow}$***,*** $^aL^{\leftarrow}$***,*** $^aL^{*}$***,*** $^gL^{\leftrightarrow}$***, and*** $^gL^{\rightarrow}$***.** *Modify Algorithm 3.3.1 as follows: add to* Γ *a new element,* o*, which is incremented on each open reference and decremented on each close.*[6] *In the calculation of* d *(the reference distance between two files), set* d *to zero if the open count for* f_1 *is nonzero. Also, when calculating* $\pi^{\leftrightarrow}$ *and* $\pi^{\rightarrow}$*, multiply by* $d+1$ *rather than* d *(to protect against multiplication by zero). The remainder of the algorithm is the same.*

Theorem 3.3.1 *The distance measures* $^aD^{\leftrightarrow}$, $^aD^{\rightarrow}$, $^aD^{\leftarrow}$, $^aD^{*}$, $^gD^{\leftrightarrow}$, $^gD^{\rightarrow}$, $^aL^{\leftrightarrow}$, $^aL^{\rightarrow}$, $^aL^{\leftarrow}$, $^aL^{*}$, $^gL^{\leftrightarrow}$, *and* $^gL^{\rightarrow}$ *can be calculated in* $\Theta(F^2)$ *space.*

Proof Immediate from Observation 3.3.1 and Algorithms 3.3.1 and 3.3.3. ■

Observation 3.3.3 *Assuming* F *files and* T *references, and with* $T > F$*, the time complexity of Algorithms 3.3.1 and 3.3.3 is* $O(FT)$*.*

Observation 3.3.4 *The best algorithms we have found for calculating* $^gD^{*}$, $^gD^{\leftarrow}$, $^gL^{*}$, *and* $^gL^{\leftarrow}$ *require* $O(F^2T)$ *space and* $O(T^2)$ *time. It is an open question whether better algorithms might be devised for calculating these distance measures.*

Observation 3.3.5 *Since these are lower bounds, the only way to save space (and time) is to approximate the distance measures by ignoring some file pairs.*

Observation 3.3.6 *Since our primary purpose is to identify semantic clusters, we are only interested in calculating distance measures in those cases where they will turn out to have a small value. However, we do not know* a priori *which pairs of files will satisfy this condition.*

Observation 3.3.7 *If we could predict* a priori *which file pairs were relevant, we could calculate the distance measures accurately for those pairs while discarding accuracy for others.*

3.3.5 Choice of Distance Metrics

In Sections 3.2.2 3.2.3 we have outlined two different methods of calculating semantic distance (basic and lifetime), and for each method we have defined four different distance types. The choice between the resulting

[6]The remaining operations could be performed for all references, rather than only for opens, at the designer's option, without affecting the complexity of the algorithm.

eight options depends on a number of factors, the most important of which are relevance, computability, and efficiency.

After analyzing and experimenting with the options, we felt that lifetime distance represents file relationships better than basic distance, because it better captures the behavior of programs that keep files open for long periods. We chose the pairwise distance over the other options both for efficiency and because it is nearby relationships, rather than historical ones, that best identify the semantic closeness of files to one another. Thus, SEER uses $\mathcal{L}^{\leftrightarrow}$ to calculate inter-file distances.

3.4 Distance Sequences

The sequence of open/close references $\mathcal{S}$ develops dynamically over time. Since $\mathcal{L}$ is derived from $\mathcal{S}$, $\mathcal{L}$ also develops over time. Each reference s_i generates up to $F-1$ distances, which we can consider as being created simultaneously in an unspecified order. In this section we will confine our attention to those distances that are members of $\mathcal{L}^{\leftrightarrow}$, unless otherwise specified. We use f_i^o and f_i^c to refer an open or close, respectively, of f_i.

All following theorems also assume that the underlying system allows at most O_{MAX} files to be open at the same time, unless otherwise specified.

3.4.1 Restrictions on Distance Multisets

Although any reference sequence is possible, the same is not true of distance sequences. The following theorems will demonstrate this fact.

Theorem 3.4.1 *There exists an open/close reference sequence $\mathcal{S}$ such that $\forall a, b : a, b \leq O_{MAX} : \mathcal{L}^{\leftrightarrow}_{f_a,f_b} = \{0, \ldots 0\}$, where the cardinality $N^{\leftrightarrow}$ of $\mathcal{L}^{\leftrightarrow}$ is 1 if $b < a$ and 2 otherwise.*

Proof The sequence

$$\{f_1^o, f_2^o, \ldots f_{O_{\text{MAX}}}^o, f_1^c, f_1^o, f_2^c, f_2^o, \ldots f_{O_{\text{MAX}}}^c, \ldots f_2^c, f_1^c\}$$

satisfies the requirements. ■

Theorem 3.4.2 *For $F > 2$, no basic reference sequence $\mathcal{R}$ can produce $\mathcal{D}^{\leftrightarrow} = \{1\}$ for all files.*

Proof Without loss of generality, we consider only sequences in which the first reference to each file appears in file-number order and in which repeated adjacent references to the same file are elided. Assume there is a sequence that produces $\mathcal{D}^{\leftrightarrow}_{f_i,f_j} = \{1\}$ for all i, j. By assumption the first two elements of $\mathcal{R}$ are f_1 and f_2, respectively.

Now consider the position of f_3. By assumption, it must appear after the first appearance of f_2 in $\mathcal{R}$. If f_3 immediately follows the first appearance of f_2, the local pairwise distance between f_1 and f_3 will be 2, and the sequence $\mathcal{R}$ will not satisfy the requirements of the theorem. Since files must first appear in file-number order, the only other choice is for f_1 to reappear at least once following the first appearance of f_2 but before the first appearance of f_3. In this case, the local pairwise distance between f_2 and f_3 is 2 or more, and again $\mathcal{R}$ does not satisfy the requirements. Since there are only two possibilities for placing f_3, there is no sequence $\mathcal{R}$ that satisfies the requirements of the theorem. ■

Theorem 3.4.3 *No open/close reference sequence can produce $\mathcal{L}^{\leftrightarrow}_{f_af_b} = \{0, \ldots 0\}$ for more than O_{MAX} file pairs f_a, f_b.*

Proof Without loss of generality, we consider only sequences in which the first reference to each file appears in file-number order. Assume that we have a sequence, not necessarily that used in Theorem 3.4.1, which generates $\mathcal{L}^{\leftrightarrow} = \{0, \ldots 0\}$ for all O_{MAX} files referenced in the sequence. We would like to extend this sequence by inserting a new reference, $s^o_{O_{\text{MAX}}+1}$, which will cause the distance from all other files to $f_{O_{\text{MAX}}+1}$ to be zero.

Because of the restriction on the order of initial references, the new reference must be inserted following the first reference to $f_{O_{\mathrm{MAX}}}$. Since only O_{MAX} files may be open at one time, the location of this insertion means that some other file f_i must have been both opened and closed at least once before the appearance of the new reference, and that this file is still closed. Then, by Definition 3.2.1, $\ell_{f_i f_{O_{\mathrm{MAX}}+1}}$ must necessarily be nonzero. ■

Theorem 3.4.4 *In any basic reference sequence $\mathcal{R}$ that refers to F files, there will be at least one local distance $d_{ij} \geq F-1$.*

Proof Without loss of generality, we consider only sequences in which the first reference to each file appears in file-number order. Consider the first reference to file f_F. By hypothesis, every other file in $\mathcal{F}$ has already appeared at least once in $\mathcal{R}$. When f_F appears, some other file f_i must have appeared least recently. Therefore, there must have been intervening references to all of the other $F-2$ files (excluding f_i and f_F) between the last reference to f_i and the first reference to f_F. The existence of these intervening references requires that the distance between f_i and f_F be $F-1$ or greater. ■

Corollary 3.4.1 *In a basic reference sequence $\mathcal{R}$ referring to F files, there is at least one distance d^1_{ij} such that $d^1_{ij} \geq F-1$, another d^2_{ij} such that $d^2_{ij} \geq F-2$, etc., down to 1.*

Proof Apply the proof of Theorem 3.4.4 to files f_F, f_{F-1}, f_{F-2}, etc. ■

Theorem 3.4.5 *In any open/close reference sequence $\mathcal{S}$ that refers to $F > O_{MAX}$ files, there will be at least one local lifetime distance $\ell_{ij} \geq F - O_{MAX} - 1$.*

Proof Without loss of generality, we consider only sequences in which the first reference to each file appears in file-number order. Consider the first open of file f_F. By hypothesis, an open reference to every other file in $\mathcal{F}$ has already appeared at least once in $\mathcal{S}$. When f_F appears, some other file f_i must have been closed least recently. Following the last open of f_i (which must have preceded the close), there must have been closes of all of the other $F-2$ files. Each of these closes must have had a corresponding open, and all but O_{MAX} of those opens must have followed the last open of f_i. Thus, the distance between f_i and f_F must be $F - O_{\mathrm{MAX}} - 1$ or greater. ■

3.4.2 Pathological Distance Multisets

The above development shows that some distance multisets are impossible. We now demonstrate that despite this fact, there are other achievable distance sequences that can be troublesome to some approximation algorithms that will be introduced in Section 3.5.3.

Theorem 3.4.6 *For any number of files F, there is at least one open/close reference sequence $\mathcal{S}$ that generates $N^{\leftrightarrow}_{ij} = 1$ for all file pairs.*

Proof Construct the sequence

$$\mathcal{S} = \{s^o_1, s^c_1, s^o_2, s^c_2, \ldots s^o_F, s^c_F, s^o_{F-1}, s^c_{F-1}, \ldots s^o_1, s^c_1\}$$

The ascending portion of the sequence generates exactly one $\ell^{\leftrightarrow}_{ij}$ for each file pair for which $j > i$. The descending portion does the same for pairs where $j < i$. Because the definition of $\mathcal{L}^{\leftrightarrow}$ uses only the nearest member of $\mathcal{S}$ for each file, these distances will be the only members generated. ■

Corollary 3.4.2 *There is also a basic reference sequence $\mathcal{R}$ that generates $N^{\leftrightarrow}_{ij} = 1$ for all file pairs.*

Proof Modify the sequence used in Theorem 3.4.6 by dropping closes and converting opens into simple references. ■

Theorem 3.4.7 *For a given open/close reference sequence $\mathcal{S}_T$, the arithmetic-mean pairwise lifetime distance ${}^{a}L^{\leftrightarrow}_{f_a,f_b}$ between two files f_a and f_b can be made arbitrarily close to 0 by appending an appropriate sequence.*

Proof In the sequence $\mathcal{S}_T$ there are $N^{\leftrightarrow}_{f_a,f_b}$ local distances ℓ_{ij} with a sum of $\sum_{\ell \in \mathcal{L}^{\leftrightarrow}} \ell$, so that ${}^{a}L^{\leftrightarrow}_{f_a,f_b} = \frac{1}{N^{\leftrightarrow}} \sum_{\ell \in \mathcal{L}^{\leftrightarrow}} \ell$. To modify this sequence so that ${}^{a}L^{\leftrightarrow}_{f_a,f_b} = \epsilon$ for some $\epsilon > 0$, we append n repetitions of $\{f_a^o, f_b^o, f_a^c, f_b^c\}$, where n will be chosen later. Each of these repetitions adds a $\ell_{ij} = 0$ to $\mathcal{S}_T$. Then the new ${}^{a}L^{\leftrightarrow}_{f_a,f_b}$ is given by:

$${}^{a}L^{\leftrightarrow}_{f_a,f_b} = \frac{\sum_{\ell \in \mathcal{L}^{\leftrightarrow}} \ell}{N^{\leftrightarrow} + n}$$

We wish to choose n such that the desired ϵ is achieved:

$$\begin{aligned} \epsilon &= \frac{\sum_{\ell \in \mathcal{L}^{\leftrightarrow}} \ell}{N^{\leftrightarrow} + n} \\ N^{\leftrightarrow} + n &= \frac{\sum_{\ell \in \mathcal{L}^{\leftrightarrow}} \ell}{\epsilon} \\ n &= \frac{\sum_{\ell \in \mathcal{L}^{\leftrightarrow}} \ell}{\epsilon} - N^{\leftrightarrow} \end{aligned}$$

Thus, if we choose $n \geq \frac{\sum_{\ell \in \mathcal{L}^{\leftrightarrow}} \ell}{\epsilon} - N^{\leftrightarrow}$, we will achieve the desired ϵ. ■

Observation 3.4.1 *The references appended to $\mathcal{S}$ in the proof of Theorem 3.4.7 may instead be inserted in arbitrary locations, so long as each insertion consists of one or more pairs $\{f_a^o, f_b^o, f_a^c, f_b^c\}$. The proof of the theorem requires only that each pair produce a $\ell^{\leftrightarrow}_{ij} = 1$ and that the total number of pairs be sufficiently large.*

Theorem 3.4.8 *For a given open/close reference sequence $\mathcal{S}_T$, the geometric-mean lifetime pairwise distance ${}^{g}L^{\leftrightarrow}_{f_a,f_b}$ between two files f_a and f_b can be made arbitrarily close to 1 by appending an appropriate sequence.*

Proof In the sequence $\mathcal{S}_T$ there are $N^{\leftrightarrow}_{f_a,f_b}$ local distances ℓ_{ij} with a product of $\prod_{\ell \in \mathcal{L}^{\leftrightarrow}} (1 + \ell)$, so that ${}^{g}L^{\leftrightarrow}_{f_a,f_b} = \sqrt[N^{\leftrightarrow}]{\prod_{\ell \in \mathcal{L}^{\leftrightarrow}} (1 + \ell)}$. To modify this sequence so that ${}^{g}L^{\leftrightarrow}_{f_a,f_b} = \epsilon$ for some $\epsilon > 1$, we append n repetitions of $\{f_a^o, f_b^o, f_a^c, f_b^c\}$, where n will be chosen later. Each of these repetitions adds a $\ell_{ij} = 0$ to $\mathcal{S}_T$. Then the new ${}^{g}L^{\leftrightarrow}_{f_a,f_b}$ is given by:

$${}^{g}L^{\leftrightarrow}_{f_a,f_b} = \sqrt[K]{\prod_{\ell \in \mathcal{L}^{\leftrightarrow}} (1 + \ell)}$$

where $K = N^{\leftrightarrow} + n$. We wish to choose n such that the desired ϵ is achieved:

$$\begin{aligned} \epsilon &= \sqrt[K]{\prod_{\ell \in \mathcal{L}^{\leftrightarrow}} (1 + \ell)} \\ \epsilon^K &= \prod_{\ell \in \mathcal{L}^{\leftrightarrow}} (1 + \ell) \end{aligned}$$

$$\begin{aligned}
\epsilon^{N^{\leftrightarrow}+n} &= \prod_{\ell\in\mathcal{L}^{\leftrightarrow}}(1+\ell) \\
\epsilon^{N^{\leftrightarrow}}\epsilon^{n} &= \prod_{\ell\in\mathcal{L}^{\leftrightarrow}}(1+\ell) \\
\epsilon^{n} &= \frac{\prod_{\ell\in\mathcal{L}^{\leftrightarrow}}(1+\ell)}{\epsilon^{N^{\leftrightarrow}}} \\
n &= \log_{\epsilon}\left(\frac{\prod_{\ell\in\mathcal{L}^{\leftrightarrow}}(1+\ell)}{\epsilon^{N^{\leftrightarrow}}}\right)
\end{aligned}$$

If we choose n greater than or equal to the value given above, we will achieve the desired ϵ. ■

Theorem 3.4.9 *If k new references are added to ${}^{g}L^{\leftrightarrow}_{f_a,f_b}$ at distances*

$$\{\ell_1,\ell_2,\ldots\ell_k\}$$

the ratio of the new to the old distance, denoted I, is given by:

$$I=\sqrt[K]{\frac{\prod_{1}^{k}\ell_i^{N^{\leftrightarrow}}}{({}^{g}L^{\leftrightarrow})^{kN^{\leftrightarrow}}}}$$

where $K=N^{\leftrightarrow}(N^{\leftrightarrow}+k)$.

Proof Before the k new references are added, ${}^{g}L^{\leftrightarrow}=\sqrt[N^{\leftrightarrow}]{\prod_{\ell\in\mathcal{L}^{\leftrightarrow}}(1+\ell)}$. When the new references are included, ${}^{g}L^{\leftrightarrow}=\sqrt[N]{\prod_{\ell\in\mathcal{L}^{\leftrightarrow}}(1+\ell)\prod_{1}^{k}\ell_k}$, where $N=N^{\leftrightarrow}+k$. I is the ratio of the new to the old value. Taking the logarithm of I,

$$\begin{aligned}
\ln I &= \frac{\sum_{\ell\in\mathcal{L}^{\leftrightarrow}}\ln\ell+\sum_{1}^{k}\ln\ell_i}{N^{\leftrightarrow}+k}-\frac{\sum_{\ell\in\mathcal{L}^{\leftrightarrow}}\ln\ell}{N^{\leftrightarrow}} \\
&= \frac{N^{\leftrightarrow}\left(\sum_{\ell\in\mathcal{L}^{\leftrightarrow}}\ln\ell+\sum_{1}^{k}\ln\ell_i\right)-(N^{\leftrightarrow}+k)\left(\sum_{\ell\in\mathcal{L}^{\leftrightarrow}}\ln\ell\right)}{N^{\leftrightarrow}(N^{\leftrightarrow}+k)} \\
&= \frac{N^{\leftrightarrow}\sum_{1}^{k}\ln\ell_i-k\sum_{\ell\in\mathcal{L}^{\leftrightarrow}}\ln\ell}{N^{\leftrightarrow}(N^{\leftrightarrow}+k)}
\end{aligned}$$

Letting $K=N^{\leftrightarrow}(N^{\leftrightarrow}+k)$ and exponentiating, we get

$$\begin{aligned}
I &= \sqrt[K]{\frac{(\prod_{1}^{k}\ell_i)^{N^{\leftrightarrow}}}{(\prod_{\ell\in\mathcal{L}^{\leftrightarrow}}\ell)^{k}}} \\
&= \sqrt[K]{\frac{\prod_{1}^{k}\ell_i^{N^{\leftrightarrow}}}{({}^{g}L^{\leftrightarrow})^{kN^{\leftrightarrow}}}}
\end{aligned}$$

■

Theorem 3.4.10 *There exists a finite basic reference sequence $\mathcal{R}$ in which*

$$\forall i,j : i \neq j : {}^{g}D^{\leftrightarrow}_{ij} \to 1 \text{ as } T \to \infty.$$

Proof[7] Given F files, consider each of the pairs $\{f_i, f_j\}$, where $1 \leq i < j \leq F$. There are $F(F-1)/2$ such pairs. For some arbitrary integer k, concatenate each pair with itself k times, forming a sequence containing k repetitions of each file pair, for a total length of $kF(F-1)$. We concatenate the subsequences for $k = 1, k = 2, \ldots$ to produce an infinite string. For any prefix of this infinite sequence, we will determine an upper bound on the distance between any pair of files. We will then show that this bound approaches 1 as the prefix lengthens.

First, consider the contribution to the distance ${}^{g}D^{\leftrightarrow}_{ij}$ from $\{f_i, f_j\}$ pairs that terminate in the last portion of the prefix, the subsequence that contains each pair repeated k times. Call this subsequence the *subject sequence*. If $i < j$, there are k instances of $\{f_i, f_j\}$, while if $i > j$, there are $k-1$ instances. In either case, there are at least $k-1$ instances of $\{f_i, f_j\}$.

There are also instances of longer distances in the subject sequence. Any distance that terminates in the subject sequence must terminate in an occurrence of f_j. In $F-1$ subsequences of the subject sequence, f_j appears as part of a repeated pair; in each of these subsequences, at most only the first repetition of f_j can terminate a distance that is greater than 1, since the remaining repetitions are "shielded" from earlier instances of f_i by the first instance of f_j. The length of the subject sequence is $kF(F-1)$, and the previous sequence (all pairs for $k-1$) is obviously shorter. So any distance terminating in the subject sequence is less than $2kF(F-1)$. We have seen that there are at most $F-1$ such distances.

Now consider successively longer prefixes of the infinite sequence, starting at $k=1$. There are at least 0 distances of 1 in the first sequence, 1 distance equal to 1 in the second, 2 in the third, etc. Adding these values up to the $(k-1)^{\text{st}}$ sequence, there are at least $k(k-1)/2$ distances of length 1. (We deliberately exclude the final sequence in the prefix, so that our proof demonstrates an upper bound for the distance even if the infinitely long string is truncated in the middle of a sequence.)

We have seen that there are at most $F-1$ instances of distances greater than 1 that end in the last sequence, and all are less than $2kF(F-1)$. So there are at most $k(F-1)$ such instances anywhere in the prefix, and all of them are shorter than $2kF(F-1)$. (Note that we include the final sequence in this upper limit, so as to be generous in our estimate of these contributions that would increase the upper bound.)

Given at least $k(k-1)/2$ distances of length 1 and at most $k(F-1)$ distances of length less than $2kF(F-1)$, the geometric mean of the distances must clearly be less than

$$(2kF(F-1))^{\frac{k(F-1)}{k(k-1)/2}}$$

The numerator of the exponent represents the maximum number of distances greater than 1, and the denominator represents the minimum number of distances of any length. In particular, we have been pessimistic by counting only the distances of length 1 in the denominator, but that is sufficient for the proof.

The exponent simplifies to $2(F-1)/(k-1)$, yielding

$$(2kF(F-1))^{\frac{2(F-1)}{k-1}}$$

Taking the logarithm, we have

$$\frac{\ln(2kF(F-1))}{2kF(F-1)}$$

Applying L'Hôpital's rule, we find the first derivative of the numerator and denominator with respect to k:

$$\frac{1/(2kF(F-1))}{2F(F-1)}$$

[7] This proof was kindly provided by Eric Postpischil.

and find the limit as $k \to \infty$, which is clearly zero. Since the limit of the logarithm of the distance is 0, the limit of the distance itself is 1. ■

Corollary 3.4.3 *There exists an open/close reference sequence $\mathcal{S}$ in which*

$$\forall i, j : i \neq j : {}^gL_{ij}^{\leftrightarrow} \to 1 \textit{ as } T \to \infty.$$

Proof Perform the same construction as in the proof of Theorem 3.4.10, using the sequence $\{f_a^o, f_b^o, f_a^c, f_a^o, f_b^c, f_a^c\}$ in place of the pair $\{f_a, f_b\}$, and so forth. The quantity $2kF(F-1)$ becomes $4kF(F-1)$, but this change does not affect the limit. ■

3.5 Estimates of Pairwise Reference Distance

Taking Observation 3.3.5 (the only way to save space and time is to ignore some file pairs) as our guide, let us consider the possibility of approximating a mean distance ${}^aL$ or ${}^gL$ through some unspecified calculation involving a chosen subset of the corresponding multiset $\mathcal{L}$. Let $\widetilde{{}^aL}$ and $\widetilde{{}^gL}$ be the approximations to ${}^aL$ and ${}^gL$, respectively. Let $\mathcal{L}^+$ be the subset of $\mathcal{L}$ that is included in the approximation, and $\mathcal{L}^-$ be the excluded subset, so that

$$\begin{aligned} \mathcal{L} &= \mathcal{L}^+ \cup \mathcal{L}^- \\ \mathcal{L}^+ \cap \mathcal{L}^- &= \emptyset \end{aligned}$$

(Following our usual convention, the cardinalities of $\mathcal{L}^+$ and $\mathcal{L}^-$ will be denoted N^+ and N^-, respectively.) Then

$${}^aL = \frac{1}{N}\left(\sum_{\ell \in \mathcal{L}^+} \ell + \sum_{\ell \in \mathcal{L}^-} \ell\right)$$

The relative error ϵ in $\widetilde{{}^aL}$ is

$$\epsilon = \frac{\widetilde{{}^aL} - {}^aL}{{}^aL}$$

which can easily be seen to have a range of $(-1, \widetilde{{}^aL} - 1)$ when $1 < \widetilde{{}^aL} < \infty \wedge 1 < {}^aL < \infty$. Similarly,

$${}^gL = \sqrt[N]{\Big(\prod_{\ell \in \mathcal{L}^+} \ell\Big)\Big(\prod_{\ell \in \mathcal{L}^-} \ell\Big)}$$

and the relative error is

$$\epsilon = \frac{\widetilde{{}^gL} - {}^gL}{{}^gL}$$

with a similar range of $(-1, \widetilde{{}^gL} - 1)$.

We will define several approximations to ${}^aL$ and ${}^gL$, identified by subscripts as $\widetilde{{}^aL}_1, \widetilde{{}^aL}_2, \ldots$ and $\widetilde{{}^gL}_1, \widetilde{{}^gL}_2, \ldots$ and with errors denoted $\epsilon_1, \epsilon_2, \ldots$

Some of the following theorems refer to the *control* we have over the membership of a multiset. By this term we mean that we can select certain values to be included in that multiset according to some criterion. For example, if we insist that all members of $\mathcal{L}^+$ be less than 10, the membership of $\mathcal{L}^+$ is controlled. Either of $\mathcal{L}^+$ and $\mathcal{L}^-$ may be controlled, but it is not possible to control the membership of both multisets simultaneously because doing so would require controlling the membership of $\mathcal{L}$, which would imply a chosen reference stream. (Having both $\mathcal{L}^+$ and $\mathcal{L}^-$ be uncontrolled is equivalent to selecting their members at random; we will not consider this case.)

3.5.1 Mean of a Subset

The simplest way to calculate $\widetilde{{}^aL}$ or $\widetilde{{}^gL}$ is as the appropriate mean of $\mathcal{L}^+$:

$$\widetilde{{}^aL_1} = \frac{1}{N^+} \sum_{\ell \in \mathcal{L}^+} \ell$$

and

$$\widetilde{{}^gL_2} = \sqrt[N^+]{\prod_{\ell \in \mathcal{L}^+} \ell}$$

Observation 3.5.1 *If the membership of $\mathcal{L}^-$ is not controlled according to value, the error in $\widetilde{{}^aL_1}$ and in $\widetilde{{}^gL_2}$ is also uncontrolled, and thus has a range of*

$$-1 \leq \epsilon_1 \leq \widetilde{{}^aL} - 1 \tag{3.1}$$

or

$$-1 \leq \epsilon_2 \leq \widetilde{{}^gL} - 1 \tag{3.2}$$

respectively.

Observation 3.5.2 *One way to attempt to reduce the absolute value of ϵ_1 or ϵ_2 would be to apply some sort of correction based on the value of N. Unfortunately, it is not possible to do so, since recording N for all file pairs would require $O(F^2)$ storage, which is exactly what we are trying to avoid.*

Observation 3.5.3 *Another way to reduce ϵ_1 or ϵ_2 is to control the membership of $\mathcal{L}^-$ such that excluded members do not make a large contribution to ϵ_1 or ϵ_2, respectively.*

3.5.2 Fixed-Space Algorithms

Since the difficulty we are fighting is the $O(F^2)$ storage requirement, an obvious choice is to assign a fixed amount of space to storing and calculating $\mathcal{L}^+$.

For simplicity of exposition, our subsequent development will be limited to algorithms for estimating ${}^gL$. All of these algorithms can be modified to estimate ${}^aL$ as well, if desired.

Keep Smallest k Distances

The simplest variation on Algorithm 3.3.1 is to store only the smallest k distances, for some k. Since the identity of the smallest distance varies depending on whether the arithmetic or geometric mean is used, we will restrict ourselves to discussion of the latter, with the understanding that much of the development is the same for arithmetic means.

At each step of the algorithm, the distances from f_1 to f_2 are sorted according to the value of $\sqrt[n]{\pi}$, and all but k of these distances are discarded. Since each step can add at most one distance to the k already stored, the computation for each step reduces to simply calculating the distance $\widetilde{{}^gL}$, inserting it into the proper place in the list of k distances, and discarding the $k+1$st. We will call this algorithm *keep-smallest-k* (KSK) and the approximation generated by it $\widetilde{{}^gL_3}$. The results for $\widetilde{{}^gL_2}$ still apply, since

$$\widetilde{{}^gL_3} = \sqrt[N^+]{\prod_{\ell \in \mathcal{L}^+} \ell}$$

We have only changed the subscript to highlight the fact that we are now using a specific algorithm to select the members of $\mathcal{L}^+$.

An important factor in this algorithm is the treatment of ties. If the new $\widetilde{{}^gL}$ being inserted is equal to the current largest $\widetilde{{}^gL}$, either may be legitimately kept. This condition arises frequently in practice when both $\widetilde{{}^gL}$ and N^+ are small. The complete algorithm for handling ties is discussed later, in Section 5.3.2 (p. 54). However, ties do not affect the following development.

Theorem 3.5.1 *For distances calculated by KSK, $-1 < \epsilon_3 < \widetilde{{}^gL_3} - 1$.*

Proof By Observation 3.5.1, it suffices to prove that we do not have control over the membership of $\mathcal{L}^-$. By Corollary 3.4.3, it is possible to have more than k files, each of which have equal and minimum distance from the file in question, which we will call f_i. Necessarily, then, there will be some file that is omitted from the list of k closest files. Call this file f_j. All references to this file will generate distances that are members of $\mathcal{L}^-$.

Now modify the continuing reference sequence so that all of the k closest files grow in distance. Then insert a single reference pair, $\{f_i, f_j\}$. f_j will now enter the list of closest k files. However, the previous membership of $\mathcal{L}^-$ for ℓ_{ij} is completely uncontrolled, so the relative error is given by Equation 3.2. ■

Theorem 3.5.2 *The time complexity of KSK is $O(FT)$.*

Proof Each new reference to a file f_a in $\mathcal{S}$ updates the distances to all of the other $F-1$ files. It is necessary to examine each of these files to decide whether the distance between it and f_a is currently being stored and thus must be updated, or if the distance is smaller than some current distance and thus must be inserted. Since there are T references in $\mathcal{S}$, $O(FT)$ operations are necessary to perform the updates. ■

Theorem 3.5.3 *The space complexity of KSK is $O(F)$.*

Proof Immediate from the definition of the algorithm. ■

3.5.3 Variable-Space Algorithms

The algorithms in Section 3.5.2 use a fixed and predictable amount of storage for each file, resulting in $O(F)$ space complexity. If we are willing to accept a worst case of $O(F^2)$ and depend on statistical behavior to lessen the storage requirement, then we can use variable-space algorithms.

Mean Threshold

One way to reduce the space requirements is the *mean threshold* method (MT), in which we calculate running means and discard all mean distances that are currently greater than or equal to a threshold $\theta > 1$. We will call the approximation calculated by this method $\widetilde{{}^gL_4}$.

Theorem 3.5.4 *For distances calculated by the mean threshold algorithm, $-1 < \epsilon_4 \leq 0$.*

Proof By definition, distances kept by $\widetilde{{}^gL_4}$ are less than or equal to θ. If the true distance ${}^gL$ is greater than θ, the error will be negative. It is easy to see that the lower bound can be achieved by setting the distance sequence to $\{\mathcal{L}^-, \mathcal{L}^+\}$ where the members of $\mathcal{L}^-$ are arbitrarily large and $\mathcal{L}^+ = \{1\}$.

The only way a positive error could be achieved is to have $\widetilde{{}^gL_4} > {}^gL$. This condition is only possible if some members of $\mathcal{L}^-$ are less than θ. But the only way these values could have been discarded is if they were averaged with other values sufficiently greater than θ to cause the resulting mean to exceed θ. There might have been multiple instances of discarding, in which $\widetilde{{}^gL_4}$ was first less than θ and then grew to become larger. For each such sequence, $\sqrt[N]{\prod_{\ell \in \mathcal{L}^\leftrightarrow} (1+\ell)} > \theta$, so $\prod_{\ell \in \mathcal{L}^\leftrightarrow} (1+\ell) > \theta^N$. The geometric mean of these sequences is equal to the $(\sum N)$-th root of their product. It is easy to see that this value would exceed θ, and thus that the geometric mean of $\mathcal{L}^-$ would therefore be greater than θ. Thus, it is not possible to have $\widetilde{{}^gL_4} > {}^gL$. ■

Theorem 3.5.5 *The worst-case space requirement of MT is $O(F^2)$.*

Proof Immediate from Corollary 3.4.3. ■

Theorem 3.5.6 *The worst-case time complexity of MT is $O(FT)$.*

Proof By Corollary 3.4.3, it is possible to generate reference sequences in which all pairwise distances are less than θ. Thus, each new reference can potentially update the distances to all F files, without causing any of these distances to exceed θ. Since there are T references in $\mathcal{S}$, these updates require $O(FT)$ operations. ■

Threshold

A variation of MT is the simple *threshold* (T) algorithm, in which we define

$$\mathcal{L}^- = \{\ell_{ij} : \ell_{ij} > \theta\}$$

where $\theta \geq 1$, and thus

$$\mathcal{L}^+ = \{\ell_{ij} : \ell_{ij} \leq \theta\}$$

We will call this approximation $\widetilde{{}^gL_5}$. The algorithm can be easily implemented by keeping a history of the last θ files referenced, and updating only the distances from those files to the currently referenced file; this approach is attractive because the running time is proportional only to T.

Theorem 3.5.7 $-1 < \epsilon_5 \leq 0$.

Proof Since every member of $\mathcal{L}^+$ is less than or equal to θ, it is clear that $\sqrt[N^+]{\prod_{\ell \in \mathcal{L}^+} \ell} \leq \theta$. Further, since every member of $\mathcal{L}^-$ is greater than θ, it is also clear that $\sqrt[N^+]{\prod_{\ell \in \mathcal{L}^+} \ell} \leq \sqrt[N]{\prod_{\ell \in \mathcal{L}} \ell}$ and thus

$$\begin{aligned}
\epsilon_5 &= \frac{\sqrt[N^+]{\prod_{\ell \in \mathcal{L}^+} \ell} - \sqrt[N]{\prod_{\ell \in \mathcal{L}} \ell}}{{}^gL} \\
\epsilon_5 &\leq \frac{\sqrt[N]{\prod_{\ell \in \mathcal{L}} \ell} - \sqrt[N]{\prod_{\ell \in \mathcal{L}} \ell}}{{}^gL} \\
\epsilon_5 &\leq 0
\end{aligned}$$

By Observation 3.5.1, the lower bound is -1. ■

Theorem 3.5.8 *The worst-case space requirement of T is $O(F^2)$.*

Proof The construction used in the proof of Corollary 3.4.3 generates an unbounded number of distances equal to 1 for each pair of files in $\mathcal{F}$. Thus, for any $\theta > 1$, a distance must be stored for each pair of files. ■

Theorem 3.5.9 *The time complexity of T is $O(T)$.*

Proof Since the $\ell_{ij}^{\leftrightarrow}$ are monotonically increasing, any member of $\mathcal{S}$ can only change the distance to the θ most recently-referenced files. Updating these distances requires $O(\theta) = O(1)$ time. Since there are T members of $\mathcal{S}$, the total complexity is $O(T)$. ■

Adjusted Threshold

In the *adjusted threshold* (AT) algorithm, we improve the error behavior of T by counting the size of $\mathcal{L}^-$. Since every member of $\mathcal{L}^-$ is greater than θ, it is certain that the product of $\mathcal{L}^-$ is at least $(\theta+1)^{N^-}$. This fact can be used to improve the accuracy of our approximation, which we will call $\widetilde{{}^gL_6}$:

$$\widetilde{{}^gL_6} = \sqrt[N]{(\theta+1)^{N^-} \prod_{\ell \in \mathcal{L}^+} \ell}$$

Theorem 3.5.10 $\epsilon_5 < \epsilon_6 \leq 0$.

Proof Since AT estimates $\prod_{\ell \in \mathcal{L}^-} \ell$ as $(\theta + 1)^{N^-}$, the estimate of $\prod_{\ell \in \mathcal{L}^-} \ell$ can never exceed the true value. Thus, $\widetilde{{}^gL_6} \leq {}^gL$, proving the upper bound.

For the lower bound, by definition $\widetilde{{}^gL_5} \leq \widetilde{{}^gL_6}$. Furthermore, equality is achieved only when $N^- = 0$, and in that case $\epsilon_6 = 0$ as well. So ϵ_5 must be strictly less than ϵ_6. ■

Theorem 3.5.11 *The space complexity of AT is $O(F^2)$.*

Proof The count N^- must be maintained for every file pair. ■

Theorem 3.5.12 *The time complexity of AT is $O(FT)$.*

Proof Whenever a distance is discarded, the counts for all files that have excluded that distance must be updated. The number of files that exclude a particular count will be approximately $F - \theta$, which is $O(F)$. Thus, the time complexity to process a single reference is $O(F)$; since there are T references, the total time complexity of AT is $O(FT)$. ■

The time and space complexity of AT are no better than Algorithm 3.3.1. The importance of AT is not in superior performance but in its contribution to our final algorithm, to be outlined in Section 3.5.4.

3.5.4 Combination Algorithms

Noting that some of our approximations feature linear space complexity, while others have linear time behavior, we might combine appropriate features from multiple algorithms to try to achieve linear time and space.

Threshold-Limited Keep Smallest k Distances

The obvious candidates for combination are threshold (T) and keep-smallest-k (KSK). In this algorithm, we discard all $\ell_{ij}^{\leftrightarrow} > \theta$, and also apply the sort-and-discard method used in KSK to limit the space requirements. We will call this approximation $\widetilde{{}^gL_7}$, and refer to the algorithm as *T-KSK*.

Theorem 3.5.13 $-1 < \epsilon_7 < \widetilde{{}^gL_7} - 1$.

Proof Since T-KSK discards all distances discarded by either T or KSK, it has control no greater than that of those two algorithms, and thus its relative error cannot be better than the worst of T and KSK. KSK has the wider error bounds, and thus controls the maximum error of T-KSK.

Theorem 3.5.14 *The space complexity of T-KSK is $O(F)$.*

Proof Immediate from the definition of the algorithm. ■

Theorem 3.5.15 *The time complexity of T-KSK is $O(T)$.*

Proof Same as the proof of Theorem 3.5.9. ■

Adjusted Threshold-Limited Keep Smallest k Distances

A transformation similar to that which converted T into AT can be applied to T-KSK, producing the *Adjusted Threshold-Limited Keep Smallest k Distances* (AT-KSK) algorithm. In this modification, a count is kept of the membership of $\mathcal{L}^-$, but the count is maintained only for those files that are members of the list of k smallest files. This change reduces the space requirements to $O(F)$, at the expense of a slightly wider error bound. We denote the distance calculated by this algorithm $\widetilde{{}^gL_8}$.

To count the members of $\mathcal{L}^-$, we must keep track of the occurrence of each member. Since we are interested in online algorithms, we must be able to calculate the semantic distance between two files at any point in the reference sequence. For example, consider the following two similar sequences (where each f_i represents an open/close pair):

$$\{f_1, f_2, f_1, \ldots f_1, f_2\}$$

and

$$\{f_1, f_2, f_1, \ldots f_2\}$$

where the omitted references do not include either f_1 or f_2, and there are at least θ omitted references.

In the first case, there are two pairwise local distances $\ell^{\leftrightarrow}_{f_1 f_2}$, each with a value of 1, and T-KSK would detect and record both. In the second case, T-KSK would see a single local distance $\ell^{\leftrightarrow}_{f_1 f_2}$ of 1. The second $\ell^{\leftrightarrow}_{f_1 f_2}$, which has an unknown value greater than θ, would not be recognized or even counted, because f_1 has been dropped from the history before f_2 is referenced.

In AT-KSK, we use an *adjustment flag*, which indicates that f_1 was the last of the two files to appear. This flag is set when f_1 is dropped from the history of θ most recently-seen files, but only if f_1 has the most recent reference time at that point. One adjustment flag is kept for each mean distance ${}^gL^{\leftrightarrow}_{f_1 f_2}$.

At the next reference to either f_1 or f_2, we apply a conditional correction to any flagged ${}^gL^{\leftrightarrow}_{f_1 f_2}$, clearing the flag in the process. If the reference is to f_1, there is no correction, because the first case above applies and a future reference to f_2 will generate a local distance $\ell_{f_1 f_2}$ that will update ${}^gL^{\leftrightarrow}_{f_1 f_2}$.

However, if the reference is to f_2, then the second case applies. We know that a distance has appeared that is a member of $\mathcal{L}^-$, and without knowing its precise value, we can estimate it as $\theta+1$ and use this value to update ${}^gL^{\leftrightarrow}_{f_1 f_2}$.

As described, AT-KSK is expensive to implement, because on every reference to f_2, all known mean distances ${}^gL^{\leftrightarrow} f_i f_2$ must be located and adjusted. However, we can improve the performance by noting that the adjustment can be done only when ${}^gL^{\leftrightarrow} f_1 f_2$ is examined for some other purpose, providing that we can decide (a) whether an adjustment is needed, and (b) which of f_1 and f_2 was referenced first after the adjustment flag was set.

A convenient time to make this decision is the next time f_1 is referenced, since at that point all ${}^gL^{\leftrightarrow} f_1 f_j$ are readily accessible. The first condition is then true if the adjustment flag is set. The second decision can be made by comparing the last reference times of f_1 and f_2. If f_2 has been referenced since the last reference to f_1, then the second case applies and an adjustment is needed.

Theorem 3.5.16 $\epsilon_7 \leq \epsilon_8 \leq 0$.

Proof Since AT-KSK estimates $\prod_{\ell \in \mathcal{L}^-} \ell$ as $(\theta+1)^{\widetilde{N^-}}$, where $\widetilde{N^-} \leq N^-$, the estimate of $\prod_{\ell \in \mathcal{L}^-} \ell$ can never exceed the true value. Thus, $\widetilde{{}^gL_6} \leq {}^gL$, proving the upper bound.

For the lower bound, by definition $\widetilde{{}^gL_7} \leq \widetilde{{}^gL_8}$. However, unlike AT, equality can be achieved if $\widetilde{N^-}$ is zero. This situation can occur for a pair of files f_i and f_j if f_j is not added to f_i's list of k nearest files until after all members of $\mathcal{L}^-$ have appeared. ■

Theorem 3.5.17 *The time complexity of AT-KSK is $O(T)$, and the space complexity is $O(F)$.*

Proof AT-KSK is the same as T-KSK except for the adjustment phase. The information necessary to perform the adjustments can be stored with each file in constant space (only the adjustment flag is needed).

The adjustment can be performed in constant time when f_1 is next referenced. Thus, AT-KSK has the same time and space complexity as T-KSK, or $O(T)$ and $O(F)$, respectively.

AT-KSK is the algorithm used by SEER. The complete implementation is described in Section B.2.3 (p. 142).

Chapter 4

Clustering Methods

One of the major contributions of SEER is the use of cluster analysis to infer information about the user's projects. This chapter discusses the clustering process and the underlying algorithms.

4.1 Prior Art

The field of cluster analysis has a long and rich history. Cluster analysis is used for many applications, in both business and science. The problem has been widely studied, and there have been numerous algorithms proposed to attack it. Summaries of the field can be found in [Bock 1974, Duran and Odell 1974, Hartigan 1975, Sokal 1977, Späth 1980, Zupan 1982].

The difficulty of cluster analysis is twofold:

1. Before clusters can be formed, a numerical metric must be found that can usefully distinguish related from unrelated objects (a "distance metric"). For many algorithms, a second metric is required that can be used to evaluate the quality of a cluster.

2. Cluster analysis is compute-intensive. If a cluster-quality metric is available, optimal clustering (assigning objects to an arbitrary set of clusters such that the metric is maximized) has been shown to be NP-hard [Garey and Johnson 1979, p. 281].

Most of the research in clustering algorithms has concentrated on solving the second problem through efficient heuristics, although there has been a limited amount of investigation into the question of measuring the distance between objects.

4.1.1 Classes of Clustering Methods

Traditional clustering methods can be roughly classified into three types of algorithms:

Iterative Objects are assigned to clusters through some initial method (often randomly). The initial assignment is then perturbed according to some algorithm in an attempt to improve the overall quality of the assignment.

Agglomerative Objects are assigned to clusters through some initial method (usually one cluster per object). The original clusters are then combined into larger clusters based on an inter-cluster distance measure. Sometimes multiple iterations are performed, so that a hierarchy of nested clusters is produced.

Divisive Objects are first assigned to clusters through some initial method (usually one cluster for all objects). The initial clusters are then divided into smaller clusters. Sometimes, as in agglomerative clustering, the process is repeated to produce a hierarchy.

Iterative methods are probably the most popular of the three. One advantage of iterative methods is that it is at least theoretically possible to achieve an optimal clustering. Another advantage, for the purposes of Seer, is that a previous clustering could be used as a starting point for a future run, which could potentially have a significant impact on long-term efficiency. The major disadvantages are that iterative methods are expensive, sensitive to initial conditions, and sensitive to the accuracy of the distance and quality metrics.

Agglomerative methods are popular in situations where a hierarchical clustering is useful. Agglomerative methods can also be fast. Since a cluster can often be represented by a single point (frequently the centroid), combining two clusters reduces the total amount of future processing, and it is usually easy to compute the new representative point. Since the aggregation operation is simple, the overall algorithm is usually efficient. Agglomerative algorithms are not normally sensitive to initial conditions. However, agglomerative methods can be unstable, producing very different clusters when certain parameters are varied slightly.

Divisive methods are the converse of agglomerative methods. Theoretically, any agglomerative algorithm can be converted into a divisive one, and vice versa. However, it is usually more efficient to combine two clusters than to divide them. (For example, a combined centroid can be computed knowing two centroids and the number of members in each cluster, while finding the new centroids of a cluster divided in two requires examining every member.) For this reason, divisive methods are rarely used.

4.1.2 The Algorithm of Jarvis and Patrick

The algorithm used by Seer is based on one developed by Jarvis and Patrick [Jarvis and Patrick 1973]. Because the Jarvis and Patrick algorithm is both simpler and more generalized than that used in Seer, we summarize it here before describing our modification.

The algorithm operates in two phases: neighbor finding and clustering. The first phase locates and records the n nearest neighbors of each point; the second uses these nearest-neighbor tables to compute a distance measure that drives the clustering process. One major advantage of the Jarvis and Patrick algorithm is that the clustering phase is single-pass; however, it does not produce a hierarchical clustering.

In the first phase, each point A is compared to every other point B and a distance (usually Euclidean) is calculated. A running table of the k nearest neighbors to A is kept,[1] and B is inserted into this table if appropriate, dropping the farthest neighbor to make room. The same updating process can be applied to B at the same time.

Since each point needs only a list of k neighbors, where k is a parameter of the algorithm, the space complexity of this phase is $O(N)$. Since each pair of points must be compared, the time complexity is $O(N^2)$.

The second phase begins by assigning each point to its own cluster. For each point A, the k nearest neighbors are examined. For each such neighbor, B, a distance metric is calculated as follows: if A is listed in B's nearest-neighbor list, and A and B have at least n other nearest neighbors in common (where $n < k$ is a parameter of the algorithm), then the two points are considered "close;" otherwise they are "far". Whenever two points A and B are close, their containing clusters are combined. Since combination is done based only on neighbor lists, which do not change, the algorithm is order-independent, idempotent, and single-pass.

There are a number of ways of representing cluster membership. One simple method is to maintain a linked list of members for each cluster. This data structure requires $O(N)$ space regardless of the number of clusters. Combining clusters can then be done in constant time. Since each point must be examined only once, and processing a single point requires constant time, the time complexity of the second phase is $O(N)$. Thus, the complete algorithm requires $O(N)$ space and $O(N^2)$ time.

An interesting characteristic of this algorithm is that it does not incorporate assumptions about the shape of a cluster. So long as there is an unbroken chain of nearby pairs of points, a cluster can be as irregular or snakelike as one might imagine. This tolerance of odd shapes is unusual in the field of clustering, and turns out to be very useful for Seer.

[1]In the original paper, the roles of k and n are interchanged. The notation used here is chosen for consistency with the algorithms given in Chapter 3.

4.2 Clustering in Seer

4.2.1 Requirements

As discussed previously, Seer forms clusters of files as a fundamental part of the hoarding process. The primary distance measure is semantic distance as defined in Chapter 3, although other information from external investigators (Section 5.4, p. 56) must also be incorporable. The number of files monitored by Seer is typically in the tens of thousands. The chosen clustering algorithm must then satisfy the following requirements:

Efficiency. Because of the number of points (files) involved, algorithms that require $O(N^2)$ space or time are unacceptable. (The restriction on time complexity might be relaxed).

Limited Information. The number of points implies that it is not practical to calculate a semantic distance between every pair of files, since that would require at least $O(N^2)$ time and space. A clustering algorithm appropriate for Seer must therefore be able to operate without a complete distance matrix.

Overlapping Clusters. Some files, such as utility programs, must be members of more than one cluster (see Section 6.4, so the clustering algorithm must be able to generate partially disjoint clusters with common members.

Non-metric distance measure. Semantic distance does not satisfy the triangle inequality, precluding algorithms that depend on Euclidean spaces.

Multiple distance measures. Information from external investigators, usually expressed as distances between some pairs of files, must be usable. (This requirement cannot be satisfied by the traditional method of incorporating multiple measurements, which is to express each value as a position along an axis in multidimensional space and then reduce these values using a Euclidean or similar distance metric, because external investigators do not necessarily provide distances between every pair of points.)

The algorithm we have developed, which is described below, satisfies all of these requirements.

4.2.2 The Seer Algorithm

Basic Algorithm

The fundamental clustering algorithm used in Seer is a slight variation on the method of Jarvis and Patrick. Rather than making an $O(N^2)$ pass through all pairs of points to calculate a nearest-neighbor list, we use the list of the k closest files calculated by Algorithm AT-KSK in Section 3.5.4 as a substitute. This design reduces the time and space complexity to $O(N)$.

A minor but important difference from the original Jarvis and Patrick algorithm is that when we compare two points A and B, we do not require that B include A in its nearest-neighbor list. This relaxation of the design is required because the semantic-distance measure is not symmetric (it is possible for the semantic distance from A to B to be small while the distance from B to A is large.)

Because we found that distance pairs observed only a few times tend to be unreliable, we insist that a reference occur several times before we consider it as a shared neighbor. The threshold is user-settable, with a default of 3 occurrences. Rejecting small sample sizes helps to suppress the poor clusterings that would arise as a result of chance relationships.

Overlapping Clusters

To generate overlapping clusters, we run the clustering algorithm twice, using two parameters, n_1 and n_2, with $n_2 < n_1$. The first pass operates as given in Jarvis and Patrick, with the modifications mentioned in Section 4.2.2. The second pass, with the smaller (and thus looser) threshold, does not combine nearby clusters into larger ones. Instead, each of the two files involved in the comparison is inserted into the other's cluster. Thus, the files at the boundaries of the two clusters tend to become members of each, without causing the clusters to join into one unmanageably large group.

However, this design can produce identical and subset clusters to be produced. For example, consider a two-file cluster, $\{A, B\}$ and a single-file cluster, $\{C\}$. If A is loosely related to C, each will be added to the other's cluster, resulting in the cluster $\{A, B, C\}$ and $\{A, C\}$. If B is also loosely related to C, adding each to the other's cluster will result in two identical clusters.

Identical clusters are a problem because they interfere with the hoard-priority calculation described in Section 5.3.4 (p. 56), so SEER detects and suppresses identical clusters before computing hoard priorities.

Subset clusters are potentially a useful phenomenon, because the system has the option of storing either the smaller or the larger cluster, depending on hoard space. However, subset clusters add some complexity to the process of calculating hoard priority; this process is discussed in detail in Section 5.3.4.

Incorporating Other Measures

As discussed in Section 4.2.1, other distance measures besides semantic distance need to be incorporated besides semantic distance. One of these measures is the directory distance, which is zero for files in the same directory and positive for files in different directories, with larger values as the files become farther apart in the tree. Directory distance is the only measure that can be calculated between any pair of files in the database; SEER calculates it on demand for files that have some other relationship recorded so that it does not have an $O(N^2)$ efficiency impact. We follow Tait [Tait *et al.* 1995] in assuming that being in a different directory implies that files are unrelated, rather than assuming that being in the same directory implies a relationship.

There may also be distances provided by external investigators; these distances are provided only for some pairs of files at the option of the particular investigators.

The extra distance measures must be incorporated into the clustering calculation in some way. One obvious method would be to use a standard distance-reduction method, such as Euclidean or Mahalanobis [Späth 1980] distance, to combine these distances with the semantic distance. These modified semantic distances would then be used to form a new nearest-neighbor list for the clustering algorithm.

However, combining the investigated distances directly with the semantic distance introduces a difficulty: the need to calculate a new nearest-neighbor list. Since the semantic distance is only known for the files in the current neighbor list, there is no way to insert new files that have become closer than other neighbors as a result of the incorporation of the additional information.

An alternative to applying the distance measures to the semantic distance is to combine them directory with the count of shared nearest neighbors. For example, suppose a pair of files A and B have i neighbors in common, as determined by examining the list of semantically nearest files maintained by Algorithm AT-KSK in Section 3.5.4. We modify i by considering it to be one axis in a multidimensional space, with the other axes corresponding to the directory-distance measure and the various externally investigated values. By careful definition of the various measures, we can safely assume that an unspecified value is equal to zero. However, it is still important to consider the signs of various measures (i is a "higher is better" metric in the sense that larger values will make clustering more likely; external investigations have been defined so that a larger value means that a pair of files is more closely related, while directory distance should be treated in the opposite sense because higher values mean that a pair of files is more distant from one another).

We originally considered using Euclidean distance as a method for combining investigator data, but we decided against Euclidean distance both for simplicity and so that investigators could have the option of decreasing the existing values (Euclidean distance can only produce an increase). Our current implementation uses Manhattan distance: the directory distance and all external values are each multiplied by a measure-specific constant (currently unity for all measures) so that it is easy to experiment with different weightings of the various measures.

Chapter 5

System Design

SEER is divided into several major parts:

- A small *kernel hook* into the operating system that allows system calls of interest to be traced. The hook comprises about 2000 lines of C code, which contain about 400 semicolons.

- An *observer* that collects system calls, selects those that are of importance to a particular user, and converts them into a generalized file-reference trace. The `observer` is about 4600 lines of C^{++}, containing about 1400 semicolons. The observer also uses library routines totaling about 1200 lines, 850 semicolons.

- A *correlator* that receives file references from the `observer`, deduces semantic relationships, forms clusters, and makes hoarding decisions. The `correlator` is about 27,000 lines of C^{++}, containing about 6500 semicolons. It also uses 4300 lines of libraries totaling about 1200 semicolons; these libraries are the same as those used by the `observer`.

- A *controller* for sending and receiving commands and information to and from the `correlator`. The `controller` is about 2000 lines of C^{++}, with about 500 semicolons.

- Several *external investigators*[1] that evaluate information embedded in the file system and to pass it to the `correlator` (via the `controller`), for use in hoarding decisions. The existing investigators represent about 500 lines of scripts and associated control files.

- An underlying *replication substrate* that handles the details of moving files between the portable and fixed computers, and of keeping hoarded information consistent with other copies of the same files. Our prototype replication substrate, CHEAP RUMOR (described in Appendix A), is about 5400 lines of PERL [Wall and Schwartz 1991] code.

- A *hoarding interface* that converts decisions by SEER into commands to the replication substrate. The hoarding interface for CHEAP RUMOR is a single script of about 400 lines.

This chapter discusses these components, their interactions, and the rationale behind the choices made in their design.

We also developed several auxiliary programs and scripts for dumping data structures, controlling simulations, and so forth. These utilities are not described here.

All told, the complete SEER system, including miscellaneous utilities but excluding CHEAP RUMOR, totals about 7000 lines of scripts, 500 lines of makefiles, and 40,000 lines of C and C^{++} code that contain slightly over 10,000 semicolons.

[1]This term was suggested by Peter Reiher [Reiher 1995].

5.1 Underlying Assumptions

The SEER system was designed to satisfy the following assumptions and goals:

- SEER is not responsible for replication or for file consistency.
- The user's primary computing device will be the portable machine, but other machines may be used as well.
- Each user will have only one portable, disconnectable computer.
- The underlying replication system will provide either a global namespace or one that allows easy conversion of local file names into their remote equivalent.
- The portable computer will have modern disk, memory, and CPU capacities, but will nevertheless place a premium on disk space.
- Disconnected or weakly-connected operation will be common.
- Fully-connected operation will be available from time to time.
- Efficient operation of the SEER subsystem is of only secondary interest.

The last assumption (efficiency) requires some justification. We do not argue that the efficiency of a predictive hoarding system is unimportant in a production environment. To the contrary, we believe that the overhead of the current system is higher than desirable and can be reduced. However, when we first designed the system, we made a deliberate and conscious choice to concentrate on functionality and research flexibility, discarding efficiency whenever it was clearly achievable but would add significantly to development effort. This allowed us to apply our limited resources to solving the significant fundamental problems of predictive hoarding. We plan to address efficiency as an important issue in the future.

5.2 Observing User Behavior

To make inferences from user behavior, one must be able to observe that behavior. Since the behavior of interest to SEER is embodied by system calls, our implementation must be able to observe or infer what those calls are. This information must then be collected and integrated to deduce file relationships for hoarding purposes.

In SEER, the information is collected through a small modification to the operating system kernel. The kernel hook builds trace packets, each of which contains a summary of a selected system call,[2] and makes the packets available to the `observer`, which performs the following functions:

- Selects packets that are relevant to one or more chosen users,
- Tracks process creations and terminations,
- Tracks process working directories,
- Classifies references according to type,
- Converts relative pathnames into absolute ones (under LINUX only; the conversion is done by the kernel in SUNOS),
- Converts file handles into file names when necessary,

[2]Not all calls are traced, and the traced calls tend to be infrequent, so the performance impact is minimal. See Section 8.3.1, p. 101, and Section B.1.1, p. 132, for more information.

- Passes chosen trace packets to one or more `correlator`s in a standard form,
- Optionally writes incoming packets to a save file for debugging or replay.

The `observer` is capable of watching the activities of more than one user, and of communicating with more than one `correlator`. In the initial design, we expected to support shared workstations, so that an `observer` might need to track the activities of more than one user. Since there can only be one `correlator` per user, tracking multiple users required multiple output connections. As it turned out, we have never found it necessary to this feature.

A closely related feature, also unused, allows the `correlator` to run on a different machine from the `observer`. This flexibility was intended to support the common situation where a user maintains active windows on several computers. The idea was that the user would have an `observer` on each of these machines, each feeding information to a single `correlator` on the primary laptop. Limitations in the underlying replication systems have precluded this mode of operation, but we have not found its lack to be a significant problem.

5.3 Correlating References

The heart of SEER is the `correlator`, and naturally it comprises the majority of the code. As outlined in Section 5.2 and described in detail in Section B.1.2 (p. 133), the `observer` collects system trace packets, turns them into standardized reference types with full pathnames, and passes them to the `correlator`. The `correlator` in turn is responsible for:

- Communicating with the `observer` and `controller`,
- Tracking file references,
- Calculating semantic distance,
- Maintaining the database of known files, and
- Making hoarding decisions

Each of these activities is itself a complex process, discussed further in Section B.2, p. 134.

5.3.1 Managers

Because it is a research tool, the SEER `correlator` was designed to support multiple *hoard managers*. For each file known to the system, a manager makes a decision whether that file should or should not be hoarded (stored locally). Via the `controller`, the system can select a manager to be used and discover the hoarding decisions that must be executed by the replication substrate.

There are currently five hoarding managers:

LRU The LRU manager fills the hoard on a simple LRU basis. This hoarding algorithm is the same as that used by LITTLE WORK [Huston and Honeyman 1993].

Bounded LRU The bounded LRU manager modifies plain LRU by allowing the user to provide an instruction file that specifies bounds on the perceived LRU age of particular files. Files older than their bounds are treated by the manager as if the bound were their true age. This manager has not been extensively used due to the difficulty of specifying appropriate bounds for large numbers of files.

Weighted LRU The weighted LRU manager modifies plain LRU by allowing the user to provide an instruction file that specifies weights to be applied to the perceived LRU age of particular files. The weight is a real-valued multiplier on the age, so that different files will age at different rates. No existing system uses this scheme, but we believe that it would be more powerful and easier for users to understand than the calculation used in CODA. As with the bounded LRU method, we have been unable to convince any users to exert the effort needed to exercise this manager fully.

Linear LRU The linear LRU manager combines the ideas of bounding, weighting, and CODA-style offsetting into a single bounded linear transformation. For each file, the user may specify an offset (y intercept), weight (slope), and upper bound to be applied to the file's LRU age. By appropriate selection of these parameters, any of the other LRU-style managers, including the CODA method, can be supported. Again, however, users have not proven receptive to amount of work required to use this system. In addition, we have found it difficult to express the linear transformation in intuitively simple terms.

Clustering The clustering manager uses an entirely new approach to making hoarding decisions, first described in [Kuenning 1994]. Semantic distance and information from external investigators are integrated by a clustering algorithm, which generates overlapping clusters that represent the user's projects. Hoarding decisions are then made based an LRU age that is representative of the entire cluster. Using the same age for the entire cluster ensures that all files needed to work on a project are simultaneously available. It also allows the user to signal an attention shift by simply referencing a single file within the project.

There are also three pseudo-managers; these are described in Section B.2.2, p. 136.

5.3.2 Calculating Semantic Distance

When a file reference is recorded by the `correlator`, it is classified according to type. If it is a type that is involved in the semantic-distance calculation, a process-specific reference history (see Section B.2.2, p. 141) is scanned for previously referenced files. For each nonempty entry in the reference history, the semantic distance is potentially updated. When all entries have been processed, the oldest entry is discarded, and a record of the just-referenced file is inserted at the head of the history.

In the following discussion, *referenced file* refers to the file that has just been referenced. *History entry* refers to the entry in the history table that is being examined or updated, and *history file* refers to the file mentioned in that entry.

To update a history entry, it is first checked to see whether it refers to the referenced file. If so, the older entry is discarded and the history-update loop is exited (since all previous entries were updated as a result of the earlier reference). Otherwise, the local distance from the history entry to the current file is equal to the distance from current entry to the end of the history table, and this value is used to update the semantic-distance list of the file associated with the history entry.

Updating the semantic distance from a file A (the history file) to file B (the referenced file) involves two steps. First, an entry for B must be created in A's semantic-distance list. If this step is successful, the distance itself must be updated. The latter operation is quite simple: just increment the number of references, and add either the latest distance (for arithmetic means) or the logarithm of one plus the distance (for geometric means) to the total distance.

The first step, locating or creating an entry for B, is complicated by the need to replace an entry if the list of related files is already at the limit of k. The list is first searched to see if an entry already exists for B. If so, the entry can simply be updated. If not, and the entry count is below k, a new entry can be created and added to the list. Otherwise, an existing entry must be replaced, according to the following prioritized heuristics:

1. Replace an entry whose target is in the `FORGOTTEN`, `FORGETTING`, or `MISSING` state (see Section B.2.2, p. 140), with priority in that order.

2. Search for the entry whose average distance is largest, breaking ties randomly, and replace that entry *if* the new semantic distance is less than the average distance of the replaced entry.

3. Replace an obsolete entry, as defined below.

The reasoning behind replacing obsolete entries is that it is possible for a file pair to be referenced a few times, at a very small distance, and then never appear again. If these entries aren't replaced, they will take up space in the semantic-distance list that could be better used for more frequently occurring entries. If the target file of a relationship hasn't been referenced within a certain number of references, SEER considers the relationship "old" and makes it a candidate for replacement as discussed above. The exact threshold is computed by adding a constant (currently 25,000 references) to a factor based on the average distance. This factor is calculated by multiplying the average distance by a weight (currently 1.2) and by the number of times this reference pair has been seen. The reasoning behind adding the average distance is that the first file of the pair has already been seen, so if the additive constant were zero, SEER could expect the second file to be referenced at approximately that interval. The first weighting factor allows for the uncertainty in this prediction; the second modifies the prediction by increasing the threshold for reference pairs that have been seen more often, under the assumption that they are more likely to recur.

The last remaining task in maintaining the semantic distance is to apply a *threshold adjustment*.[3] The adjustment is implemented exactly as described in Section 3.5.4, except that the numerical adjustment is made when the adjustment flag is set, and then backed out later if necessary.[4]

Special Cases of Semantic Distance

Most file references can be handled as simple semantic-distance updates. A few require more specialized treatment; these cases are discussed in detail in Section 6.9 (p. 68).

5.3.3 Clustering

Clustering is performed only when the clustering manager is asked to run (*i.e.*, fill the hoard). Clusters are always recreated from scratch, using the algorithm described in Section 4.2 (p. 49).

During clustering, a threshold test is applied to the neighbors of a file. We found rarely occurring reference pairs to be unreliable, leading to poor clustering choices, so SEER insists that a pair appear at least a minimum number of times (by default, 3 times) before it is counted as a nearest neighbor.

5.3.4 Hoard Filling

All managers use a similar algorithm for filling the hoard. A table of *hoard units* is constructed and then sorted according to priority. The table is then walked from the beginning, adding units to the hoard until it reaches the required size. If a hoard unit will not fit, but there is still room in the hoard, that unit is skipped and a lower-priority but smaller one will be chosen instead.

For all managers except the clustering manager, a hoard unit is the same as a file. For the clustering manager, a hoard unit is a cluster. (Since a file might appear in more than one cluster, only the sizes of previously-unhoarded files in the cluster are considered during the post-sort hoarding walk.)

A minor but important point is that files open at the time of hoarding have an "official" reference time equal to the time of the open, but for hoarding purposes they are treated as though their last reference time were equal to the current time.

Hoard Priority in LRU Managers

The priority of a hoard unit depends on the particular manager. For the LRU manager, it is simply the LRU age of the file (the older the file, the lower its priority). The bounded LRU manager modifies this algorithm by placing a file-specific bound on the LRU age; files with ages greater than their bound will have the age forced equal to the bound for sorting purposes. The weighted LRU manager multiplies the LRU age by a file-specific weight, so that some files grow old more quickly than others, before sorting. Finally,

[3] For historical reasons, the commentary in the code refers to threshold adjustment as "`NPREV` correction" (`NPREV` is the constant defining the value of the threshold θ.)

[4] There is no particular reason for doing it one way or the other; the choice is arbitrary.

the linear LRU manager integrates both of these options with Coda's offset method by applying a linear transformation (slope and intercept) to the LRU age, and then applying an upper bound to the result.

Hoard Priority in the Clustering Manager

Hoarding priority in the clustering manager is considerably more complex to calculate. Our first approach was to simply use the highest priority (youngest LRU age) of any file in the cluster. This method would cause the entire cluster to be hoarded if any file in it was recently referenced, so that a user would only need to reference one file to bring in a project. However, we learned that it is not always desirable to use the most recent reference as the cluster priority. For example, many clusters will contain the user's favorite editor, and the editor will always have been referenced recently. In general, not all of these clusters should be hoarded just because the editor has been referenced.

To deal with this phenomenon, we introduced a modification of the simple algorithm. First, the cluster is scanned for "single-use" files. A single-use file is defined as follows:

- A file that is a member of no other cluster, or
- A file that is a member of other clusters, but for which all other containing clusters are subsets or supersets of this one.

If the cluster contains one or more single-use files, its hoard priority is set based on the most recently referenced of those files. Otherwise, the cluster contains only multiple-use files, so the hoard priority is set based on the LRU age of the *oldest* file in the cluster. This approach ensures that an accidental cluster of utility programs will not be brought into the hoard unless all of the members are being used actively. (Since multiple-use files are by definition members of several clusters, there will nearly always be a cluster that will cause an important multiple-use file to be hoarded. As we show in Chapter 8, this algorithm works well in practice.)

Communicating Hoarding Decisions

In an integrated production environment, Seer would communicate hoarding decisions directly to the underlying replication system. To facilitate research development, however, the current implementation uses a slightly different method. When the user wishes to force hoarding decisions to be made, he runs a simple interface script that is designed for his replication substrate. The script uses the `controller` to force hoarding and copy the relevant hoarding decisions into a pipe. The information is then post-processed by the script and turned into commands to the underlying replication system, which executes the actual storage changes.

5.4 External Investigators

While semantic distance provides a very powerful method for discovering relationships among files, it is a low-level inferred mechanism. Often, much more precise information is available at a higher level. An example is is the well-known Unix `Makefile`, which lists explicit dependencies among the files needed to build a software package. In general, there is a rich variety of information available, the usefulness of which will vary depending on the application and the user.

To support this richness, we included a facility to support arbitrary *external investigators*. The investigators examine the file system, infer relationships, and provide that information to the `correlator`. This information is then incorporated into the clustering process as described in Section 4.2.2 (p. 50).

5.4.1 Interface to Investigators

Seer supports two types of external investigation. *Relation* investigators provide pairwise relationship information between files, which is then incorporated into the automated clustering algorithm. *Cluster*

```
12.0 /u/geoff/lib++/memleaktest.cc /u/geoff/lib++/memleak.hh
12.0 /u/geoff/dmalloc/heap.c /u/geoff/dmalloc/dmalloc.h
12.0 /u/geoff/dmalloc/heap.c /u/geoff/dmalloc/conf.h
```

Figure 5.1: Output of a Relation Investigator

```
/u/geoff/xcron/fifotest /u/geoff/xcron/fifotest
    /u/geoff/xcron/Makefile /u/geoff/xcron/fifotest.o
    /u/geoff/xcron/fifo.o
```

Figure 5.2: Output of a Cluster Investigator (folded to fit page)

investigators locate entire clusters, which are then provided to the `correlator` as a complete unit. These clusters are internally static, and are not further modified by the `correlator`. However, when hoarding decisions are made, the investigated clusters participate on an equal basis with the automatically generated ones.

The investigator interface is provided by the `controller`. An investigator invokes the `controller`, giving the name (chosen arbitrarily) and type (relation or cluster) of the investigated information. The `controller` then reads lines from standard input, parses them to extract the investigated information, and passes them on to the `correlator`. Outdated investigation information can be updated or deleted by reusing an existing name.

Investigated relations are unidirectional, just like semantic distance. The description line for a relation investigator contains a floating-point weight (higher values indicate a stronger relation, with zero being reserved for deleting existing information) followed by a source and one or more targets. A relation of the specified weight is created from the source file to each of the named targets.

Investigated clusters are unordered and unweighted. Each cluster is described by a single line of unbounded length, consisting of a name to be given to the cluster and a list of member files. The name is arbitrarily chosen by the investigator to distinguish that particular cluster so that it can be updated later.

Figure 5.1 shows an extract of output from a relation investigator, which would normally be fed to the `controller` to load the relations into SEER. In this case all of the relations have been given the same weight, 12.0. The first line indicates a relation between the source file for `memleaktest` and a header file it uses. The second and third line record a similar relationship between a single source file and two header files (note that the latter relationships could have been specified on a single line if desired).

Figure 5.2 shows the output of a cluster investigator, which again would be fed to the `controller`. The first field is a name given to the cluster; in this case it happens to be the same as the first file in the cluster. The remaining fields name the cluster members. All of these files are needed to work on the `fifotest` project together.

5.4.2 Relation Investigators

To demonstrate the concept of external investigation, we have implemented three relation investigators and a cluster investigator. The relation investigators are:

Name. An investigator that locates relationships based on common naming conventions, such as source and object files, source and revision-history files, and C^{++} header and implementation files. Files are considered related if they differ only in their suffix, or if they follow well-known relationships such as SCCS or RCS [Tichy 1982] files.

```
fifotest: fifotest.o fifo.o
          $(CC) $(CFLAGS) -o fifotest fifotest.o fifo.o
```

Figure 5.3: `Makefile` Input to a Cluster Investigator

C includes. An investigator that reads C and C^{++} source files and determines relationships among source and header files. Unlike the name investigator, this program is capable of locating relationships in different directories and among files with very different names.

TEX includes. An investigator that reads TEX documents to discover required style files, option files, and subsidiary documents.

The name investigator is implemented as a special-purpose `awk` script, while the latter two are based on a generalized pattern-driven PERL [Wall and Schwartz 1991] program written by Andrew Louie.

5.4.3 Cluster Investigators

We also implemented a cluster investigator that examines `Makefile`s to discover clusters implied by the dependencies. Figure 5.3 shows a typical `Makefile` dependency entry. The investigator invokes `make` in a no-action mode[5] so that it does not have to duplicate the complex processing of that program, captures the output, and massages it to generate the appropriate clusters. The clusters generated by the above `Makefile` line are shown in Figure 5.2. The cluster is given a name equal to the target program as a convenient way to uniquely identify it. Note that the `Makefile` itself is included in the cluster, since compilation could not proceed if the `Makefile` were not hoarded.

5.5 Replication Substrate

To be usable, a hoarding system (whether predictive or not) must be able to move files between the laptop and other networked machines, track and propagate updates, and provide the user with the illusion of a single, complete file system. Previous systems have devoted a major part of their implementation effort to solving this single problem [Kistler 1993, Huston and Honeyman 1993].

However, with the advent of generalized replication systems such as FICUS, CODA, and RUMOR, SEER is able to separate the problem of choosing hoard contents from the problem of managing a selectively replicated file system. SEER depends upon an underlying replication substrate to handle the latter problem. At a minimum, SEER requires that the substrate provide the following features:

Selective file storage. Arbitrary choice of remote or local file storage, manageable at the level of individual files.

File tracking. Internal tracking of the current location of each file.

Redundant command filtering. Ability either to query the current location of each file so that redundant location-change commands can be suppressed, or to ignore redundant requests.

Update propagation. Automatic transfer of updates (including deletions) from the computer on which they occurred to any and all other copies of the file, regardless of where the updates were initiated.

[5] Using the `-p` and `-n` switches.

Note that the above list does not include replication. If the substrate wishes, files may be merely moved back and forth between the mobile machine and some other (networked) storage site, rather than being replicated across several sites. So long as it is possible for SEER to inform the substrate that a file must be added to or removed from the portable computer, it does not care how many copies of the file exist, what its update history has been, or similar issues that cause difficulties for replication systems. Even the requirement for internal tracking is a minor artifact of the current implementation that could be easily dispensed with (the existing code finds it convenient to make hoarding requests without considering the current storage state of a file, letting the replication substrate discard redundant commands).

Although the above is a minimal list, SEER can work with much more powerful replication substrates. All of the following features can be supported or tolerated:

Multiple replicas. Multiple copies (replicas) of files, with automated propagation of updates, so long as files can be added to or removed from the hoard on an individual basis.

Seer-independent hoard changes. Hoarding actions taken by the user or automated programs without or in opposition to SEER's advice (a trivial interface script may be required to inform SEER of the action, so that it doesn't work at cross purposes to the user).

Remote access. Transparent networked access to files not kept in the hoard, so that the user sees a consistent name space even if some files are not stored locally.[6]

Security. Arbitrary security and authentication mechanisms.

Multiple substrates. Multiple, incompatible replication substrates managing different portions of the name space.

Finally, the following features are useful if available:

Conflict detection. Automatic detection and reporting of conflicting updates to multiple copies of a file.

Conflict resolution. Automatic resolution of conflicts (*e.g.* by intelligent merging of conflicting updates).

Status queries. Ability to determine the names and sizes of all unstored files, at least while connected. This feature is useful when replaying saved trace files in a development environment. Note that it can usually be implemented by running a recursive directory listing on a replica that stores copies of all files.

Location queries. Ability to discover the replication status of all files in the volume while connected. This feature allows the interface between SEER and the replication substrate to display only the changes in file status (similar to a file difference listing).

Command batching. Ability to "batch" requests to the replication substrate for efficiency.

Disconnected queries. A feature that allows the user to determine the names or attributes of files that are not locally stored. Users have found this feature convenient for distinguishing between hoard misses and simple errors in recollection of file names.[7]

Section 5.3.4 explains how SEER communicates its hoarding decisions to the chosen replication substrate(s). Appendix A describes the simple substrate we built for testing SEER.

[6] SEER can handle non-transparent name spaces, so long as there is a simple rule for detecting remote names and transforming them into local ones. The automounter support discussed in Section 6.11 (p. 70) is an example of a non-transparent space.

[7] In some cases, this feature is more than just convenient. For example, if an RCS history file [Tichy 1982] is not present, the `ci` command will create one, causing a future conflict with the correct history file. This situation illustrates one of the pitfalls of the concept of hoarding, albeit one that can be solved by a sufficiently powerful replication substrate.

5.5.1 Other Replication Substrates

Some replication substrates (*e.g.*, CODA) do not provide all of the necessary features listed above. Nevertheless, such systems can sometimes be supported by loosening SEER's control. For example, SEER can be configured to produce `.hoardrc` files [Kistler 1993, Section 5.3.1.4] that can then be loaded into CODA's hoard manager. Although CODA does not allow SEER to directly control the hoard contents, careful selection of hoard priorities can still produce hoard contents that will be satisfactory to the end user.

5.6 User Interface

SEER was designed to operate transparently, without user intervention. This goal has largely been achieved; most of the current users do not tinker with or tune the system. However, as with most research systems, it is often desirable to provide a control interface to allow parameters to be modified and debugging information to be extracted.

The basic interface to SEER is the `controller`. Among other options, this program allows the researcher and the end user to query and set internal `correlator` parameters, force management, and preload size information into the file table. More details are give in Section B.3, p. 145.

During the development of SEER, two graphical interfaces were created to sugar-coat the UNIX command-line nature of the `controller` and to display manager output in a friendlier form. The first was written in the TCL [Ousterhout 1990] scripting language, while the second was written using HTML forms and CGI scripts, so that a Web browser would serve as a convenient interface. However, neither of these interfaces are now actively maintained, because day-to-day use of SEER is so simple that no GUI is required.

5.6.1 Master Instruction File

General control information for the `correlator` is supplied in a special master instruction file. This file is created by the system administrator, and is used to inform the `correlator` of important system information such as critical directories required to bootstrap the computer.

Originally, SEER was intended to operate entirely automatically, without any human control whatsoever. We have nearly achieved that goal, but have found that there are a few situations that either are fundamentally intractable, or are too complex, subtle, or important for SEER to be trusted with.

An example of the former is files that are used in the early part of the bootstrap process. A modern operating system must perform a certain amount of setup before it is ready to start a complex program like SEER; usually this setup requires special programs and control files. For example, UNIX-like systems must generally verify the integrity of the file system, which requires that the `fsck` program be available. This step must be performed before SEER can be activated, which means that SEER will never observe `fsck` in action. It would be disastrous if SEER decided that `fsck` did not need to be in the hoard since it never appeared to be used.

An example of the latter is the X Windowing System. Although X is a user program that does not execute until after SEER is running, it is complex and is critical to the operation of the computer. Because it is consistently used and so important, there is little point in allowing SEER to control the hoarding of the major components, since they must always be present anyway. Furthermore, the "suspend" modes available on all modern laptop computers mean that a machine might be used for months at a time without X ever being restarted. In such an environment, SEER might decide that critical startup files can be dropped from the hoard, since they haven't been used in a very long time, with obvious problems when the inevitable reboot arrives.[8]

To provide for these and a few other situations, SEER supports an instruction file, which is generally created by a system administrator and transparent to users. Most entries in the instruction file specify a pathname; if the pathname refers to a directory, the instruction usually refers to everything in the subtree

[8]Note that SEER can still control less critical files such as fonts, without risking the usability of the system.

rooted at that point. There are also features allowing an instruction to apply to either a symbolic link or its expansion.

The specific features supported by the instruction file are described in Section B.2.4, p. 143.

5.6.2 Manager-Specific Instruction Files

Certain hoard managers (currently Bounded LRU, Weighted LRU, and Linear LRU) require the user to specify hoarding parameters for each individual file. This capability is supported through manager-specific instruction files, which are read when referenced by a command in the master instruction file. The format of a manager-specific instruction file is the domain of the particular manager, but all current managers use a similar format. Each instruction file consists of a series of lines, one per file or directory in the system. The first field in the line is a pathname; the remaining fields contain the parameter values to apply to that path (*e.g.*, the weight for the weighted LRU manager). If the pathname is a directory, the values apply to the subtree rooted at that point (unless overridden by a later entry in the same instruction file).

Since the `controller` provides a way to reread the master instruction file at any time, a user can change the hoarding parameters by writing them into the appropriate manager-specific file and than asking that the master file be reread. A mechanism similar to CODA's merging of `.hoardrc` files could be implemented using simple wrapper scripts, if desired. However, as mentioned before, we have found that users do not want to spend their time tinkering with arcane numerical parameters, nor to waste it with reloading specification files every time they turn to a new project. Thus, the capability to dynamically reload instruction files has gone unused to date.

5.6.3 Interface to the Replication Substrate

We have already discussed, in Section 5.3.4, how hoarding decisions are communicated to the replication substrate. There is also a need for communication in the other direction, so that SEER can be aware of changes in replication. In particular, if the replication system removes files from the local hoard through the normal file-deletion process (*i.e.*, the `unlink` system call), SEER must first be informed of the action so that it understands that the deletion is a change in hoard status, rather than the destruction of the file. This potential confusion is not a problem if the decision to remove the file from the hoard was initiated by SEER, but it is critical if the decision was initiated independently by the user.

The substrate can inform SEER of a change in storage status with the help of the `controller`, which provides a facility for just this purpose (see Section B.2.1). The substrate provides a list of files to be hoarded or removed from the hoard, and they are passed on to the `correlator`. If the substrate was not designed to work with SEER, a simple "wrapper" script or program can generate the file list. The only requirement is that the list be provided to the `controller` *before* the files are removed from the hoard, rather than after, so that the `correlator` will be aware of the impending action before it is taken.

Chapter 6

Difficulties With the Real World

The previous chapters have presented a generally elegant framework for building an automated hoarding system. Unfortunately, the realities of an actual operating system are not so clean. During the development of SEER, we repeatedly encountered real-world behavior that made the system operate incorrectly. This section reviews the most important of those practical intrusions. Although SEER currently runs under the LINUX operating system, we have concentrated on difficulties that are common to most, if not all, software platforms.

6.1 Meaningless Activities

Perhaps the most troublesome problem that arose during the development of SEER is the existence of processes and programs that engage in "meaningless" activity that provides no information about semantic relationships. One of the best examples of this type of activity is the UNIX program `find`, which searches the disk looking for a file with certain specified characteristics (most modern operating systems have a similar function). Because `find` opens every directory and looks at every file in sequence, the accesses it makes do not give any hint about what files are of interest to the user, or about inter-file relationships (other than directory relationships, which are easy to discover anyway). In addition, because `find` accesses every file, it destroys any LRU history that might have been useful in hoarding decisions.

As we gained experience with SEER, we learned that there were many programs with behavior similar to `find`, and we spent a considerable amount of time searching for the best solution to the problem. Approaches we considered or experimented with included:

1. List programs such as `find` as special cases in a control file, and ignore the accesses generated by such programs (by flagging them as "meaningless").

2. Ignore all `PEEK` accesses, which are the primary cause of the problem, and then use the instruction file to list as special cases the few programs, such as `make`, whose peeks are actually meaningful.

3. Detect processes that access a very large number of files, and assume that these processes are meaningless.

4. Detect processes that access files at a high rate (relative to real time), and assume that they are meaningless.

5. Detect that a process has opened a directory for reading (which is a typical behavior of meaningless programs) and use this fact to automatically mark it as meaningless for the rest of its lifetime.

6. Detect directory opens, but mark a process meaningless only while the directory is open.

7. Apply a threshold-based heuristic to compare the number of files a process might know about (from reading directories) with the number of files it actually touches, marking it meaningless if it touches the majority of files it has learned about.

The first approach, hand-specifying meaningless programs, is attractive due to simplicity of implementation, but places a heavy burden on the person responsible for creating and maintaining the control file. The second, ignoring `PEEK`s and specifying meaningful programs by hand, suffers from the same problem, and also is more dangerous to the end-user because a failure in the control file will cause important files to fail to be hoarded, rather than overfilling the hoard with unimportant ones. (However, it would still be possible to implement a separate "this process is meaningful even though it is peek-intensive" instruction. To date, we have not found it necessary to have such a capability.)

The third solution, using the number of files accessed, assumes that meaningless programs open more files than meaningful ones. This assumption turns out to be untrue; for example, compilers often access tens or even hundreds of header files, while a search program such as `grep` may only open a few files. If the threshold is too low, long-running editors like `emacs` will be considered meaningless, while if it is too high, many badly behaving programs will be missed.

The fourth approach (using access rates) seems plausible, but it is difficult in practice to define a threshold rate. The proper value would depend on the speed of the individual machine, and would also vary depending on system load. It might be possible to integrate scheduler information, such as the amount of CPU time or the number of time slices accumulated, to infer a rate, but this would require more extensive intrusion into the kernel.

The fifth approach, marking as meaningless any process that opens a directory, is almost as simple as the first. We experimented with this method, but it failed in practice because many meaningful programs read directories (for example, some text editors do so to implement filename completion).

The sixth solution (marking a process meaningless while a directory is open) is based on the assumption that a meaningless program such as `find` will keep at least one directory open while it descends the directory tree. Unfortunately, some implementations choose to open a directory, read the contents into an internal buffer, and then close the directory before processing the buffer, so that this solution, too, fails in practice.

The seventh method, comparing potential to actual accesses, though more complex, was much more successful. Each time a process opens a directory, SEER counts the number of files in the directory, which represents the total number of files the process could potentially access. Actual accesses are then recorded in a second counter. SEER tracks the historical behavior of a particular program and compares the relative values of the counters to a threshold, based on that history. For example, `find` will tend to have a history of accessing every possible file, and thus would be marked as being a meaningless process, while an editor will access far fewer than the maximum and will remain meaningful.[1]

Although option 7 worked for nearly all programs, we found a few cases where it broke down. For this reason, we also implemented the sixth option, considering all activity by a process to be meaningless while it has a directory open. A process is considered meaningless if either condition is satisfied; this works well because the failure mode of both methods is to not detect meaninglessness in certain situations.

There remains one more difficulty, however, which is the UNIX `getcwd` library routine. `getcwd` deduces the full pathname of a process' working directory by climbing the directory tree and locating the individual components of the path. Doing so requires opening and reading directories in a fashion that is very similar to the behavior of `find`, so that the potential-access counter approach would mark as meaningless any process that asked for the name of its own working directory. To address that problem, we installed another heuristic that detects the pattern of behavior of `getcwd` (opening the directory named "`..`" for read) and temporarily marks the process as being inside this function.[2] During this period, all file references are ignored (even for purposes of inferring meaninglessness).

[1] Two important details are that a process is never considered meaningless if the the potential-access counter is zero, indicating that it has never read a directory, and that when the potential-access counter first goes nonzero, the actual-access counter is reset to zero. Both of these features were added in response to observations of the behavior of real programs.

[2] An alternative would have been to modify the `getcwd` routine itself to notify SEER of its actions. We chose the pattern-detection approach as being less invasive and simpler to implement.

These heuristics have made it possible for SEER to make the right decision about the relevance of a process' references in most cases. Nevertheless, we have retained the ability to hand-specify a few processes as being meaningless.[3] As in information retrieval, it is necessary to filter out certain irrelevant relationships, and as in that field, the current mechanisms could undoubtedly benefit from further refinement.

6.2 Shared Libraries

Certain files on a modern computer are so fundamental that they are used by nearly every program. The most common example of this phenomenon, though hardly the only one, is the shared library. In many UNIX systems, the first several files opened by a program are shared libraries of various sorts.

The use of shared libraries presents a serious problem for a system that tries to infer relationships among files from the sequence of opens. That every program accesses a particular library is not an issue in itself, for it merely means (in terms of semantic distance) that those programs cannot run without it. The problem lies in the following file access: the SEER system, seeing an access to the shared library and then an open of some other file A, incorrectly concludes that the shared library is closely related to A. In fact, if it could store enough relationships, it would eventually decide that the shared library is closely related to every file on the computer. A clustering algorithm would then decide that everything belonged in a single large cluster.

SEER's solution is to apply a simple but effective heuristic. If a particular file represents more than a given percentage (currently 1%[4]) of all observed accesses, it is designated a "frequently referenced" file and is eliminated from the calculation of semantic distances and file relationships. Since such a file is obviously of critical importance to the user, it is always included in the hoard, regardless of its last reference time or clustering status.

An unexpected result of this heuristic is that when a user is working intensively on a single project, a "normal" file may become frequently-referenced. For example, during the development of SEER, one of its C^{++} header files became a frequently-referenced file. However, the marking of a "normal" file as frequent causes few problems for the user, since the file in question will still be hoarded. During the time that the file is considered frequently-referenced, it will not participate in semantic-distance calculations, but when the user changes to a new project, it will return to "normalcy" and previously collected relationships will again become active.

6.3 Critical Files

Every system has some files that are essential to system operation. Many UNIX systems cannot boot without programs such as `/etc/init`. Similarly, MS-DOS requires `CONFIG.SYS` and `AUTOEXEC.BAT` to operate correctly. Other files may not be necessary to boot but may be of critical importance to the user: the average UNIX user is nearly helpless without his `.cshrc` or `.profile` file.

Because modern laptops often support a suspend/resume mode that allows power to be conserved without rebooting or repeatedly logging in and out, SEER may observe that these startup files are rarely used, and incorrectly assume that the user can do without them. This rare access to critical files is a fundamental problem with any completely automated hoarding system.

SEER addresses the problem in three ways. First, the system control file discussed later in Section 6.6 can be used to specify especially critical system files or directories (such as `/etc` in UNIX) that should be left outside SEER's control.

Second, a UNIX-specific heuristic is used to apply a similar exclusion to any file whose name begins with a period (*e.g.*, `.login`). We have found that such files tend to be small, and that they tend to contain important control and configuration information that the user cannot do without.

[3] The current list is limited to `xargs`, `rdist`, the replication substrate, and the external investigators.

[4] This value was chosen by plotting the files in a sample trace in order of their access counts, and then choosing a value slightly above the "knee" of the curve.

Finally, the user may specify a personal control file listing any other files that he considers critical to successful operation. Generally, the only files that need to be listed are scripts that are run at login time, and only "power" users tend to use such scripts, so the average user is not burdened by this requirement. Nevertheless, we are unhappy with the necessity for explicit specification and plan to seek alternatives in our future research.

6.4 Utility Programs

Many, if not most, programs executed in day-to-day computing are utilities that can be used on many different projects. The use of applications for multiple tasks can present difficulties for a clustering-based hoard manager, because the program should properly be a member of several different projects. For example, a text editor might simultaneously be a member of a software development project and a document-writing project. SEER deals with this issue by allowing overlapping clusters, which are generated as described in Section 4.2.2, p. 49.

The presence of overlapping clusters introduces a secondary problem: how should a cluster be prioritized when filling the hoard? Recalling that the ideal of SEER is that a user should only have to reference a single file for a project to be hoarded, the hoard priority of any cluster should be equal to the priority of the most recently referenced file. However, this simple criterion is incorrect for utility programs, which are members of many clusters. It should not be the case that a reference to an editor will cause hoarding of every cluster that involves any sort of editing. Instead, only "target" files should be able to control hoarding. But it is still important to ensure that the editor is hoarded as part of the project.

SEER's solution to this problem is to set the hoard priority of a cluster based on the reference times of files that are members of only that cluster. Only if a cluster is comprised entirely of members shared with other clusters will its priority be set based on the reference times of shared members. The complete algorithm is described in Section 5.3.4, p. 56.

6.5 Detecting Hoard Misses

When the user wishes to access a file that SEER has decided to omit from the hoard, it is necessary to detect and record a hoard miss. This capability is important for two reasons. First, SEER should be informed of the miss so that it can add the file (and all other members of its project) to the hoard for future use. Second, hoard misses provide statistics for measuring the success of SEER (see Section 7.1.3, p. 78) and tuning the algorithms.

Depending on the underlying replication system, automatically detecting a hoard miss can range from trivial to impossible. For example, FICUS supports so-called *remote access*, where an access to a non-local object is automatically converted to an access to a remote one. However, the success of this remote access depends on the availability of the remote replica(s) of the object. If the access succeeds, SEER will be able to identify it as remote (because its internal tables will list the file as unstored), and can mark the file to be hoarded later. If the access fails, however, and returns an error code to the user, it is difficult or impossible (depending on the replication system and the error code returned) to distinguish this case from an attempt to access a completely nonexistent file. Unfortunately, accesses to nonexistent files are common in many programs, so that it is neither meaningful nor efficient to assume that any failed access represents a hoard miss.

A further difficulty arises because some hoard misses occur without a direct attempt to access the file. For example, a user might ask for a directory listing (perhaps to verify the exact file name), note that the file is missing, and never attempt to open it directly.[5] Of course, some replication systems (*e.g.*, FICUS) maintain a complete list of directory members so that this problem does not arise, but others, such as RUMOR, will simply fail.

[5]Some programs, for example TEX, behave in a similar fashion, reading the contents of a directory to determine what files can be opened.

Because of these problems, we have created a separate mechanism for tracking hoard misses when the replication system is unable to support this function. Whenever the user is unable to access a file, he can run a simple program to record the miss in a control file. For statistical purposes, the program also records the time of the miss and a user-specified severity code, as follows:

0. The entire computer is unusable because of the missing file (*e.g.*, a critical startup file is unavailable).[6]
1. The current task will change because of the missing file (*e.g.*, the user can log in but the primary source file for a program or document isn't hoarded).
2. The task will remain the same, but activity within the task will be modified (*e.g.*, an informational file is missing but work can proceed on another part of the same task).
3. The lack of the file will cause little or no trouble.
4. The file isn't actually needed right now, but the hoard should be pre-loaded so that the file will be available in the future.

6.6 Temporary Files and Directories

Many programs create temporary files to hold transient results or to pass information to closely related programs. For example, compilers often generate symbolic output and pass this symbolic form to an assembler to produce the actual object file. Because these files are transient, semantic relationships between them and other more permanent files are not useful to an automated hoarding system, yet the nature of how they are created causes them to have a very small semantic distance, displacing other files from the short list of n closely-related files kept by SEER. If these files could be detected as being temporary, they could be ignored as if they had never existed, which would allow more effective analysis of other relationships.

We first considered attempting to automatically detect temporary files based on their brief lifetimes. However, this approach turned out to be impractical because other relationships have already been affected when the temporary file is deleted, marking the end of its life. The implementation would need a method of "backing out" any relationships created during the lifetime of the file, which is not possible in the context of our current algorithms.

Instead, the implementation of SEER allows certain directories to be marked as transient in a control file (normally set up by a system administrator, rather than a user). Files created in these directories are completely ignored by SEER.

In some operating systems, such as WINDOWS, temporary files are distinguished by a naming pattern (*e.g.*, `*.TMP`) rather than by their location in a directory. On such a system, SEER would have to modified to allow such patterns to be recognized, but the principle of ignoring temporary files would remain the same.

6.7 Non-Files

The UNIX filesystem supports a number of objects besides plain files, including directories, symbolic links, and more exotic objects such as device files and pseudo-filesystems like `/proc`. Many of these objects are critically necessary for system operation (for example, the lack of a device file for the console will probably render it impossible to log in.

With the exception of directories and possibly some pseudo-filesystems, these objects take almost no disk space. Because of their importance and minimal space requirements, SEER always includes them in the hoard. Most such objects are also omitted from semantic-distance and clustering calculations, since they tend to be either transient (*i.e.*, members of `/proc`) or vary depending on extraneous factors (*e.g.*, `/dev/tty`*xx*).

[6] In this case the miss cannot be recorded until the network connection is re-established.

Two object types that deserve special attention are directories and symbolic links. Directories are the only non-file objects that may be moderately expensive to store. The need to store a directory both when connected and when disconnected depends on the underlying replication system: some may require all directories to be present to support remote access to files not hoarded on the portable computer, and most require that all directories leading to a hoarded file be present [Kistler 1993, Ratner 1995] so that the file itself can be reached.

The directory structure can also convey valuable information to the user. If a file is missing (not hoarded) but the directory is present, the user can be more certain that he is attempting to access the correct pathname than if the entire directory subtree is not hoarded. This capability can be important if, for example, the user would like to insert a cross-reference into a document even though the referenced file is currently missing.

Finally, a directory occupies space roughly proportional to the number of files within it. Thus, it can be relatively cheap to hoard a directory if the contained files are not hoarded (as noted above, if there are any contained files, the directory must be stored anyway).

There are a number of choices that could be made regarding the hoarding of directories. In SEER, we hoard all directories in managed subtrees, even if the contained files are not hoarded, and even though this decision has an impact (albeit small) on the total hoard size.

The situation with symbolic links is similar. A symbolic link is even cheaper to store than a directory, and provides similar information to the user even if the target of the link is not hoarded (actually, the information content is greater, because it identifies the missing file without possibility of typographical errors). Thus, symbolic links are also hoarded automatically. (An alternative, which was included in our original design, would be to hoard a symbolic link whenever its target was hoarded. We discarded this option because it does not save a significant amount of space, and always hoarding links simplifies the implementation.)

6.8 Simultaneous Accesses

The formulations of semantic distance given in Chapter 3 assume that the user is generating only a single stream of references. In a modern multi-tasking operating system, however, a typical user often simultaneously generates multiple independent reference streams. For example, it is common to use the time needed to compile a large program more productively by editing a document, reading e-mail, or even playing a game. The independent streams from these activities are intermixed when observed by SEER, and create incorrect and spurious file relationships if not properly handled.

We had originally hypothesized [Kuenning 1994] that the data reductions discussed in Section 3.3 (p. 28) would provide a noise-filtering mechanism adequate to eliminate the effects of these spurious relationships. Unfortunately, experience proved this hypothesis incorrect: although noise was reduced, it was not eliminated, and the resulting spurious relationships tended to cause poor hoarding decisions.

To address the problem, we found it necessary to separate the reference streams on a per-process basis in a manner similar to that used by Tait *et al.*'s SPY UTILITY [Tait *et al.* 1995]. To do so, SEER follows every program execution and termination, together with the creation of child processes (the `fork` system call in UNIX and similar systems). A separate reference-history list is maintained for each process, and semantic distances are calculated on a process-local basis. The file-open test mentioned in Definition 3.2.1 is also performed on a per-process basis.

When a process is created, its reference history is initialized from that of its parent, and at process termination, the two history lists are merged (in reference-time order). This approach allows SEER to detect extended relationships between files referenced by a process and by its parent.

6.9 Non-Open References

There is a variety of ways in which a real program can refer to a file. Besides being opened and closed in various ways (*e.g.*, for reading or writing), a file may be executed as a process, deleted, created as a special filesystem object (*e.g.*, a directory), and have its attributes examined or modified. Under some systems, alternative names for a file may also be created and used.

Many of these situations can be treated as a point-in-time reference, similar to an open immediately followed by a close. Others require more complex treatment:

Process Execution and Termination. Execution of a new process is treated as if it were an open, and process exit is treated as a close.[7] Executions are also recorded as point-in-time read references in the parent process; this extra pseudo-reference is needed so that the child executable will be recorded as closely related to the parent that invoked it.

Descriptor Duplication. Since the `correlator` maintains an open count for each file, duplications of file descriptors (`dup` system call) must increment the open count, so that the corresponding `close` can be tracked properly. However, duplications are not otherwise recorded as references.

Process Forking. Process forking is handled entirely in the `observer`. As a side effect, falsified DUP records are generated to reflect the file descriptor duplication that is a side effect of forking. Under UNIX, falsifying DUPs also requires tracking the close-on-exec flag.

File Deletion. Deleted files are removed from SEER's internal tables. However, because many programs delete files immediately prior to recreating them, SEER delays the actual removal for a short period so that valuable clustering information won't be lost if the file is immediately recreated.

Attribute Examination. Many programs examine file attributes, perhaps to whether a file exists or to discover whether it can be written. In such cases, if the file is actually of utility to the user, it will be subsequently opened, and the actual examination can be ignored because the open will be seen as a reference. However, other programs, such as `make`, base important decisions on the values of the attributes, and the examination may indicate a close relationship between the examined file and another that is actually opened. SEER uses several heuristics to deal with this problem. In general, examination of an attribute is treated as if it were a simultaneous open/close pair. However, if the examination is immediately followed by an open or another examination of the same file, the first examination is discarded as being an insignificant reference. In addition, certain more complex heuristics, discussed in Section 6.1 (p. 63), are applied in some cases.

Renaming. When a file is renamed, there are several options for dealing with accumulated file relationships. There are two fundamentally different uses for renames, which unfortunately have conflicting needs with respect to SEER: permanently changing the name of an object, and replacing an existing object with a newly-created version.

In the first case, the file relationships refer to the underlying object, and the name is incidental. In the second, the relationships are associated with the name, and the specific identity of the object is superficial.

SEER handles this problem with an adaptive heuristic. If the target of a rename does not currently exist, we assume that the first case applies, and arrange to move the relationship data from the old name to the new. However, if the target exists (or is marked for deletion from the file table), we assume the second case, and keep the relationship data that was associated with the target name.

This heuristic works well in practice, but still does not handle certain cases properly. In particular, some editors create backup files by renaming the edited file. After the second edit, the backup file will exist, and so its relationships will take priority. The important relationships of the source file will then be lost. This behavior has not caused problems in our usage to date, but we plan to investigate alternative solutions in the future.

[7]Executions may also require recording a close of the previously executing file in the same process.

6.10 Script Interpreters

Most modern UNIX systems provide kernel support for interpreted programs, such as shell and PERL scripts. This support is activated when an `exec` is issued for the script. A unique character sequence on the first line marks the file as an interpreted program, and also identifies the path to the interpreter. The kernel then executes the interpreter, rather than the named script, and passes the name of the script as an argument to the interpreter.

This feature poses a problem for SEER, because the `observer` only sees the argument to `exec`, which is the name of the executed script. The result is that no relation will be recorded between the script and the interpreter needed to execute it, and thus they may not be hoarded together.

One solution to the problem would be to further modify the kernel so that it generates an INTERPRET record (see Section B.2.1, p. 134) whenever it performs a special interpreter execution. This design would be desirable for efficiency reasons, but implementing it turns out to be problematic. The first difficulty is simply that it requires yet more hooks to be put into the kernel, making porting more difficult. The second is that some kernels implement the interpreter execution by making *ad hoc* changes in the parameters passed to the `exec` system call, and it is relatively difficult to capture the changes into an observer trace record.

Because of the difficulty of tracing script interpreter executions in the kernel, we chose a less efficient but much more flexible solution. Whenever the `observer` sees an execution of a file, it examines the first few bytes to see if it is an interpreted script. If so, it internally generates a supplementary INTERPRET packet to record the relationship between the script and its interpreter.

6.11 Automounted Pathnames

UNIX offers a feature called the *automounter*, which is designed to simplify management of large networked disk systems. The basic idea is that the automounter watches for references to certain special pathnames that represent remote disks. When such a reference is seen, the remote disk is NFS-mounted if necessary, and the referenced pathname is rewritten (using symbolic links) to point to the mounted location.

For example, suppose an automounter were configured to mount the root directories of remote machines under the directory `/root`, giving them the machine names. If the user were to refer to `/root/norgay/foo`, the automounter would NFS-mount Norgay's root under `/tmp_mnt/root/norgay`, and then (appear to) create a symbolic link from `/root/norgay` to the mount point.

Since SEER is aware of symbolic links, it will discover that the reference to `/root/norgay/foo` is actually a reference to `/tmp_mnt/root/norgay/foo`. As it turns out, following the link can be problematic, because there are circumstances when the correlator needs to refer to a file independently of any other program (*e.g.*, to discover its size). The difficulty arises because the automounter will only respond to the original pathname; paths beginning at `/tmp_mnt` will not cause automounting. Thus, an attempt to reference `/tmp_mnt/root/norgay/foo` will fail unless `/root/norgay` has been already mounted for some other reason.

The obvious solution to the difficulty is to rewrite the pathname so that it will be processed by the automounter. With an appropriate automounter configuration, the rewriting can be done by simply stripping off the string "`/tmp_mnt`". When the `correlator` attempts to discover the size of the file it knows internally as `/tmp_mnt/root/norgay/foo`, it will actually refer to `/root/norgay/foo`, the automounter will get invoked, and all will be well.

Unfortunately, the simple solution doesn't work in general. In particular, it fails when referring to the automounted directory itself, something that is very common because of the correlator's internal pathname canonicalization. For example, suppose the correlator wishes to discover whether `/root/norgay` is a directory or a symbolic link. It issues a `stat` system call on that pathname, and the automounter intervenes to NFS-mount the desired directory. The `stat` operation then informs the correlator that `/root/norgay` is a symbolic link, which the correlator will follow to the path `/tmp_mnt/root/norgay`. The next step in canonicalization would be to issue a `stat` call on the target path. However, because of pathname stripping, the call would actually be issued on `/root/norgay`, resulting in an infinite loop.

	Component Explored	`stat` on	Result
	`/`	`/`	Directory
	`/root`	`/root`	Directory
	`/root/foo`	`/root/foo`	Symbolic Link to `/tmp_mnt/root/foo`
*	`/`	`/`	Directory
	`/tmp_mnt`	`/tmp_mnt`	Directory
*	`/tmp_mnt/root`	`/root/.`	Directory
	`/tmp_mnt/root/foo`	`/root/foo/.`	Directory
	`/tmp_mnt/root/foo/bar`	`/root/foo/bar`	File

Figure 6.1: Exploration of an Automounted Path

To get around the problem, we use a UNIX-specific trick. In certain places, when a `stat` call is made on an automounted directory (determined by a nonzero strip length, Section B.2.2), the automounter prefix (`/tmp_mnt`) is still removed, but the string `/.` is appended. Adding the period ensures that the automounting will happen if necessary, but the `stat` operation will be performed on the actual automounted directory.

The complete exploration of the pathname `/root/norgay/foo` is summarized in Figure 6.1. Lines prefixed with an asterisk are discovered from internal tables, without issuing a system call. Note that the `stat` of `/root/.` should really be accessing `/tmp_mnt/root/.`, since the latter is an actual directory rather than an automounter artifact, but the current approach works because all such directories are necessarily replicated under both their true name and the automounter subtree.

One final point regarding the rewriting of pathnames relates to the lack of generality in the current design. Some automounters use a more general rewriting scheme than the simple prepending of a string. There are also other remote-access and replication systems that use general pathname rewriting, which is not supported by the current implementation. However, there is nothing to prevent such support from being added; we omitted it only for simplicity. Were SEER to be ported to a system that required the more powerful approach, it would be easy to add such a capability.

6.12 Parameter Setting

As discussed in Chapters 3 and 4, SEER's semantic-distance and clustering algorithms make use of a number of parameters and thresholds to make their decisions. The correct settings for these parameters are not obvious, and interactions among them are complex and difficult to predict.

Any of a number of well-known methods can be used to search the parameter space to find optimal combinations. The interesting problem is to define the word "optimal" in a meaningful way. We used two different definitions to find the values we currently use.

Our first definition attempted to find parameters that would locate clusters meaningful to the user. To do so, we specified a group of files that we felt should belong to the same cluster (*e.g.*, all source and object files needed to build a particular program). A script then examined the clusters generated by SEER, evaluating those that contain at least one of the specified files. A parameter setting was considered "good" if it produced a cluster containing a large fraction of the specified files, with few extraneous ones.

While this definition produced some useful initial settings, it proved to be tedious to develop the correct lists of desired files. For example, the files needed to compile a single program might appear to be neatly contained in a single directory, but in actuality would also incorporate hundreds of system `include` files. Evaluating such a long list of files to determine which ones "belonged" proved to be impractical.

We then developed a second definition in which a cluster was considered "good" if all files within it were accessed within a small number of references from each other. Thus, a project accessed as a unit, but long ago, would be a cluster, as would a project accessed yesterday.[8] The clusters from a parameter setting were sorted according to the variation in their reference times (measured from the sequence of references, not the wall clock), and settings that produced smaller variation were then chosen. Because it requires no human intervention, this method has been able to evaluate many more clusters and has produced the parameter settings we are now using in the field.

A third method has become possible only recently, with the development of the miss-free-hoard-size measure discussed in Section 7.1.3 (p. 78). This measure is ideally suited for comparative purposes, since it is both objective and completely automated. The only drawback is that it is extremely expensive to compute (the larger simulations take over 12 hours to run on our fastest machines). We hope to use this method to refine our parameter settings in the future.

6.13 Deadlock

Since Seer both traces and issues system calls, the possibility of deadlock exists if the trace buffer fills. To avoid this problem, the trace mechanism does not record calls made by the observer and correlator themselves. However, experience showed that this step was not enough. Some of the system calls made by Seer can activate daemons, notably those that support the Network File System (NFS) [Pawlowski *et al.* 1994], and deadlock can occur due to calls made by these processes. We solved this problem by modifying the kernel so that it doesn't trace most calls made by the superuser ("`root`"). However, we still trace file renames, file destruction, and process exits performed by the superuser. The first two are captured because it is critical to track file names accurately and to keep the file table clean, while the third is followed primarily to ensure that the process table does not grow without bound in certain error cases.

An alternative approach would be to identify all processes that lead to deadlock, and arrange for Seer to only avoid tracing those processes. Although this solution would have certain advantages, it would require modifications to the source code of these system daemons, and would be less robust because it would depend on accurately identifying the potential offenders.

6.14 Tracing System Calls

As explained in Section B.1.1 (p. 132), most system calls are traced through their common dispatch point, but a few calls require special treatment. The following list discusses unusual Unix system calls that need special treatment:

`exec` Is traced both before and after the system call is performed. If `exec` fails, it returns via the normal path, but if it succeeds, it uses an extraordinary return method. In the latter case, the name of the executed file is lost. The `observer` handles `exec` by caching the name of the file being executed before the system call is performed. If it fails, the cached name is discarded; otherwise, it is recorded as a success when the process performs some other system call.

`exit` Is not captured as a system call, but instead is traced by a special-purpose hook in the process-termination code. The hook is necessary because a process can exit without passing through the `exit` system call (*e.g.*, by being aborted).

`fork` Is handled normally on Linux but requires a special-purpose hook under most AT&T-derived Unix implementations, because the new subprocess does not return to user mode via the normal path.

[8] Of course, the method depends on very LRU-like behavior from the user, as opposed to the more complex behavior actually observed over long periods. Thus, we were careful to apply it only to carefully selected intervals in which the usage patterns satisfied this restriction.

6.15 Debug Traces

Although it was not necessary for correct operation of the system, we found it useful to keep traces of the input to SEER. Early in the development process, we would lose copies of the `correlator` database because of bugs in the software, and find ourselves starting over from scratch. To avoid losing data, we installed two logging facilities, one in the `observer` and one in the `correlator`. The `observer` trace simply logs the input received from the kernel.[9] The `correlator` log is somewhat more complex, recording not only the inputs received from the `observer` but all other interactions with the operating system (primarily `stat` and `readlink` system calls). Because it records the results of all system interactions, the correlator trace can be replayed for debugging, exactly reproducing the input conditions that caused a particular problem.

The trace files proved themselves valuable many times over. We included versioning information, so that we could extend the trace format when necessary without discarding old traces.[10] The traces helped us find innumerable bugs and recover from damaged databases. When we evaluated SEER's performance, the `observer` traces were available for replay into the system for simulation purposes.[11]

One minor detail is that the trace files can grow very quickly. The `observer` monitors the size of the trace file, and when it becomes too large, it automatically creates a new one and compresses the old file. We found it useful to have a `cron` job that periodically moves compressed files to an archive disk on the network (when the machine is connected) so that the portable machine's disk would not fill up with old trace data.

[9]Under LINUX, the `observer` also inserts an occasional extra record when it discovers the current working directory of an active process.

[10]Despite the versioning, we were twice forced to discard traces because they did not contain records that turned out to be critical to correct operation. More commonly, we found it possible to synthesize or do without information we had previously failed to collect.

[11]We did not normally save `correlator` traces, since they can always be reconstructed by replaying `observer` traces.

Chapter 7

Experimental Methodology

One of the difficulties in any research effort is evaluating the level of success achieved. In the case of predictive hoarding systems, evaluation turns out to be an especially difficult problem, one worthy of its own research project. In this chapter, we introduce metrics useful for measuring hoarding systems, and outline the experimental methodology we used to investigate SEER's performance.

7.1 Metrics

There are many ways to evaluate caching and hoarding systems, some more appropriate than others. We will introduce a number of possible metrics, discuss their applicability, and then justify those that we chose for measuring SEER.

7.1.1 Previous Work

Although other predictive hoarding systems have been built [Alonso *et al.* 1990, Huston and Honeyman 1993, Kistler 1993, Tait *et al.* 1995], no one has attempted a quantitative evaluation of the effectiveness of their algorithms. Some qualitative information is given in both [Kistler 1993] and [Satyanarayanan *et al.* 1993], but there are no tables or graphs showing the effectiveness of the algorithms.[1]

A list of suggested evaluation methods is given in [Satyanarayanan *et al.* 1993, Section 7.2], although there is no indication whether any were ever implemented. These include:

Time to first miss. The amount of time[2] that elapses between disconnection and the first hoard miss. This metric characterizes the length of time a user can operate (or the amount of work that can be accomplished) before noticing adverse effects due to disconnection. *Disadvantages*: Sensitive to hoard size (a sufficiently large hoard will make this metric infinite, while a small enough one will make it zero; minor variations in intermediate values may cause very large changes in the measured value, and the changes will not necessarily even be monotonic.) Quantitative differences do not translate into corresponding quality differences in hoarding methods.

Time to the n^{th} miss. Similar to the time to the first miss, but allowing more misses. *Disadvantages*: Sensitive to hoard size.

Time to the first "critical" miss. Similar to the time to the n^{th} miss, but attempting to quantify the severity of the miss. The authors do not define what they mean by "critical." This measure character-

[1]Both publications contain analyses of working-set sizes and the performance of the underlying replication system, but the success of the hoarding method is not quantitatively addressed.

[2]For all metrics, time can be measured either as clock time or as a number of references. The choice of method depends on the particular measure.

izes the amount of work a user can accomplish before disconnection becomes intrusive. *Disadvantages*: Sensitive to hoard size. Requires a definition of "critical" and a method for evaluating it.

Time until cumulative effect of misses passes a threshold. Similar to time to the n^{th} miss, but the "effect" of misses is proposed as a more meaningful criterion. No method of measuring the effect has ever been suggested. This metric characterizes the amount of work a user can accomplish before disconnection makes the system unusable. *Disadvantages*: Sensitive to hoard size. Requires a definition of "cumulative effect" and a method for evaluating it.

7.1.2 Potential Metrics

There is a host of metrics that could be used to evaluate the effectiveness of a hoarding system, whether predictive or not. First are those suggested by prior researchers, already listed in Section 7.1.1. We will not repeat the descriptions here.

Second, there are those metrics that have been traditionally used in studies of cache systems:

Number of misses. This is the number of references that are not satisfied from the cache. *Disadvantages*: Has little relevance to the actual usability of the system, since even a single miss can be catastrophic during disconnected operation. Sensitive to the configured hoard size.

Miss ratio. This is the number of misses, divided by the total number of references. It is often useful to instead consider the hit ratio, which is simply 1 minus the miss ratio. It is also possible to calculate the miss and hit ratios in terms of disk blocks referenced, rather than number of files. *Disadvantages*: Since the miss ratio is another way to express the number of misses, it shares the same disadvantages.

Working set size. The size, either in number of files or in megabytes, of the working set (all files referenced) for a given disconnection period. For disconnected operation, the working-set size characterizes the minimum hoard size needed to operate without misses, or the space needed by an optimal "oracle" algorithm.

Although the traditional metrics have been very useful in characterizing caching systems, where the cost of a miss can be precisely quantified as a time penalty, the miss count and miss ratio are not useful for comparing hoarding systems where misses have a near-infinite time cost. For this reason, we have invented a number of new metrics that can be used to characterize various aspects of hoarding systems. We have divided them into several categories to simplify discussion.

For metrics that can be calculated until one or more misses occur, we describe them for n misses, with the understanding that $n = 1$ is a valid option.

Success Measures

The following metrics generally characterize aspects of system success (*i.e.*, whether it performs as desired):

Miss-free hoard size. The amount of disk space that the hoard would have to occupy so that there would be no misses during a disconnection period. In other words, if the hoard were at least this large, the user would experience no misses under the chosen hoarding algorithm. It is easy to compare this value to the optimum (which is the working-set size). *Disadvantages*: Requires after-the-fact knowledge of actual user behavior, *i.e.*, traces.

n-miss hoard size. The amount of disk space that the hoard would have to occupy so that there would be no more than n misses during a disconnection period. *Disadvantages*: Requires after-the-fact knowledge of actual user behavior. Measures a factor that is not very interesting to the user, who doesn't want to see misses at all.

Successful disconnections. The fraction of all disconnections that were completely free of misses.[3] *Disadvantages*: Sensitive to hoard size.

Disconnections before n^{th} miss. The number of successful (miss-free) disconnections before the n^{th} miss occurs. This metric is closely related to the previous one. *Disadvantages*: Sensitive to hoard size.

Excess space over working set. This is the difference between the miss-free hoard size and the working-set size, which is the space needed by a perfect algorithm. It provides a metric that can be used to rank the success of various hoarding algorithms. *Disadvantages*: Requires after-the-fact knowledge of actual user behavior.

Difference from working set. This is the size or count of the number of files whose automatically generated hoard state differs (in either direction) from their state in a "perfect" hoarding. It provides a metric that can be used to rank the success of various hoarding algorithms. *Disadvantages*: Requires after-the-fact knowledge of actual user behavior.

Cluster miss ratio. Similar to the traditional cache miss ratio, but calculated on a per-cluster rather than a per-file basis. It could be calculated based either on file counts or on actual cluster sizes. *Disadvantages*: Suffers the same disadvantages as the file miss ratio.

Cluster members hit. The fraction of a cluster's files actually used during a disconnection period. *Disadvantages*: Sensitive to hoard size.

Efficiency Measures

The following metrics characterize aspects of system efficiency (*i.e.*, resource usage):

Wasted hoard space. The amount of disk space devoted to hoarding files that are never actually referenced. *Disadvantages*: Sensitive to hoard size.

Files unused before the n^{th} miss. This is the total number of files, minus the count of distinct files referenced before the n^{th} miss. *Disadvantages*: Sensitive to hoard size.

Disk space unused before n^{th} miss. Files unused before the n^{th} miss, expressed in terms of file sizes. *Disadvantages*: Sensitive to hoard size.

Incorrectly hoarded clusters. The membership and size of clusters that were hoarded but not needed, which could have been left out to make room for a missed cluster.

Hoarded siblings in missed clusters. The count or size of files that are members of missed (unhoarded, but referenced) clusters, but that are themselves hoarded (due to being members of other clusters.)

Space occupied by unreferenced files, per cluster. This is the average space wasted by the clustering algorithm, calculated on a per-cluster basis. *Disadvantages*: Sensitive to hoard size.

Unreferenced files in referenced clusters, per disconnection. This is the number or size of files that were in clusters referenced during the disconnection, but that were not themselves needed in the hoard. *Disadvantages*: Sensitive to hoard size.

[3] The complement of this, failed disconnections, is sometimes a more convenient way to express this metric.

Ancillary Measures

The following metrics are do not relate directly to the performance of hoarding, but are useful in characterizing other aspects of the system:

Total disconnections. The total number of disconnections observed during an experimental period.

Length of disconnection. This is the elapsed clock time for the disconnection. It is useful to subtract out the amount of the time that the computer is not actually being used, which for laptops can be estimated as the time the computer spent in a "suspend" mode. This metric gives insight into the stress placed on the system by the user, and provides a context for interpreting the time to first miss.

Length of disconnection, in references. This is the number of references per disconnection. It gives an indication of how active the user was during disconnected periods.

Files in the hoard. The number of files actually hoarded.

Total disk space referenced before n^{th} miss. The total sizes of all distinct files referenced before the n^{th} miss. This metric characterizes the minimum hoard size needed until the n^{th} miss. *Disadvantages*: Sensitive to hoard size.

Time in and out of the hoard. The amount of time an individual file spends in the hoard, versus the time it spends not being hoarded, measured across all disconnections. *Disadvantages*: Sensitive to hoard size.

Attention shifts. The number and timing of attention shifts (as defined in Section 2.3, p. 9). This metric helps to characterize the difficulties facing the hoarding system. *Disadvantages*: Sensitive to the attention-shift parameters.

Cluster size. The average size of clusters generated by the hoarding algorithm. This metric gives insight into the behavior of the clustering algorithm. The cluster size should be neither too small nor too large; instead, it should reflect the project size.

Distinct clusters referenced before n^{th} miss. Indicates the number of clusters that actually participated in user activity, up until the n^{th} miss. By implication, also indicates the number of unused clusters. *Disadvantages*: Sensitive to hoard size.

Cluster working set. This metric is the total size of all clusters referenced during the disconnection. *Disadvantages*: Sensitive to hoard size.

Sensitivity to hoard size. Many, if not most, of the preceding metrics are sensitive to hoard size. This metric quantifies that factor. *Disadvantages*: Can be very expensive to measure, since it requires calculating the subject metric for many different hoard sizes.

7.1.3 Preferred Metrics

The list of metrics in Section 7.1.2 is lengthy and incorporates far more options than can be reasonably measured. Our primary purpose in listing them was to develop and illustrate the rich variety of metrics that is available for measuring hoarding systems.

For our own experiments we selected a much smaller list of metrics that we feel best characterize the performance of SEER. Our preferred metrics include:

Working-set size. Provides a lower bound on the achievable miss-free hoard size.

Miss-free hoard size. Gives a number that can easily be used to compare hoarding algorithms.

Excess space over working set. Characterizes the absolute efficiency of a hoarding method in comparison with an optimal oracle algorithm.

Attention shifts. Characterizes the challenges presented to the hoarding system. We chose to report 20% attention shifts for both daily and weekly disconnections, to facilitate comparison with results in Chapter 2.

Total disconnections. Indicates the amount of disconnected use experienced by a particular user.

Length of disconnection. Indicates the amount of user activity during disconnected operation.

Time to first miss. Characterizes the achieved performance of the system in actual use.

Failed disconnections. Characterizes the achieved performance of the system in actual use. We chose this measure, rather than its complement "successful disconnections," because the number of failures was so low that the high success rates tended to overwhelm the presentation, making it difficult to evaluate the numbers.

7.2 Experiments

Having selected metrics for evaluating performance, the next step is to choose a measurement method. For the metrics above, there are two primary candidates: simulation and real-world deployment.

7.2.1 General Approaches

Simulation

In the simulation approach, the hoarding system is provided with a "canned" reference stream, either artificially generated or based on traces captured from real users. Certain points in simulated time are chosen to mark disconnection intervals, and appropriate statistics are calculated for those disconnections.

The great advantages of simulation are controllability and reproducibility. The disadvantage is that it can sometimes be difficult to validate the simulations against real-world performance. Simulations can also suffer from inaccuracy if the assumptions made by the simulation do not sufficiently reflect reality.

Real-World Deployment

The second candidate for experimentation is to simply deploy the system in a real-world environment, let users live with it, and record the results. In the past, this approach has been used primarily for qualitative evaluation [Kistler 1993, Tait *et al.* 1995]. The major advantage of this method is that it inarguably reflects the actual behavior of real users. The disadvantages are that deployment is sensitive to the hoard size, and is subject to the vagaries of real life: uncooperative users, system crashes, software bugs, and stylized use will all interfere with the collection of consistent data.

7.2.2 Experimental Procedure

Since both simulation and real-world deployment have advantages, we chose a combination of the two for our analysis of SEER. The working-set size, miss-free hoard size, excess space over the working set, and attention-shift statistics were evaluated with simulation; the total disconnections, length of disconnection, and time to first miss were collected by observing actual user behavior.

Simulation Methodology

For our simulations, we used traces collected from actual users. The tracing mechanism is one of the debugging features of the `observer`. Each system call that is of interest to the SEER system is recorded in a trace file, including all information that is passed from the kernel in trace packets (see Section B.1.1, p. 131, for more information). The traces are later replayed into a `correlator` that has been started in a special simulation mode.

To ensure that the simulation was not affected by startup transients, we discarded the results from the first two simulated disconnection periods. Surprisingly, the transients disappeared so rapidly that a longer warmup interval was not required.

In simulation mode, the `correlator` does not have full information about the files that existed when the trace was collected, so certain assumptions must be made about these files.

`Correlator` simulation mode differs from normal operation as follows:

- No actual hoard changes are communicated to the replication substrate.
- Files that exist when the simulation is started are assumed to have sizes equal to their current size (this assumption tends to overestimate file sizes, since most files grow during their lifetimes). The effect is to slightly overstate the hoard-size and working-set statistics.
- Files that no longer exist are assigned a size taken from a geometric distribution with a parameter of 0.00007, for an average file size of 14284 bytes. This value was chosen by examining the actual distribution of file sizes in traces observed by SEER.[4] To the extent that this distribution does not reflect actual file sizes, using it slightly distorts the hoard-size and working-set statistics.
- Files that have existed sporadically, and currently exist, will not be removed from the list of potential hoard members during the simulation run, even during times when they should not exist. The inclusion of these extra files can cause the hoard-size statistics to be slightly larger than they would be in reality, and can also cause some important file relationships to be missed by the clustering manager.
- Disconnections are assumed to be of fixed length, beginning at a convenient time, and are separated by infinitesimal reconnections during which the simulated hoard is refilled with a new set of files. Simulated daily (24-hour) disconnections were begun at midnight during standard time, or 1 AM during daylight time. Weekly (7-day) disconnections were begun at midnight or 1 AM on Sunday mornings (in the middle of the weekend). This assumption is much stronger than actuality, since our real users normally connected to the network for a significant part of each day and could not suffer a hoard miss during that time.
- Managers that depend on user input (bounded LRU, weighted LRU, and linear LRU; see Section 5.3.1, p. 53) are initialized from user data, but there is no provision for changing their parameters during the simulation run.
- Investigator data is loaded at the beginning of the run, and is not changed thereafter.

Note that several of the differences from reality have the effect of slightly changing the apparent working set or hoard size. However, since all managers are faced with the same list of file sizes, we do not believe that this small error affects the validity of our comparisons.

[4]The fit of this distribution is only fair, because actual file-size distributions do not seem to follow a clean mathematical model. We devoted considerable effort to investigating different potential distributions and calculating their parameters before settling on the geometric distribution.

Simulation Conditions As discussed above, we simulated both daily and weekly disconnections, using traces from each of 9 machines. For three of these machines we ran simulations assuming both the presence and the absence of the external investigators described in Section 5.4 (p. 56). Based on early data from these runs, we simulated the remaining 6 machines only in the absence of external investigators, since the additional relationship and cluster information did not seem to produce a significant performance difference. The chosen conditions produced a total of 60 simulations of daily and 60 of weekly disconnections.

Since some file sizes were drawn from a random distribution, our simulations had a random error component. For each set of conditions, we ran the simulation five times with different random seeds to reduce the variance due to error and to provide information for estimating confidence intervals. The random seed was the same for each pair of simulated daily and weekly disconnections, so that the two simulations were presented with the same file sizes. The random number generator used was `drand48`, a 48-bit linear congruential generator; the seeds were selected using the `/dev/random` device available on LINUX. The experiments were divided into two groups so that they could be run in parallel on two different machines. Within each group, experiments were chosen at random (again using `/dev/random`) to avoid introducing errors due to unexpected trends in experimental conditions [Jain 1991].

Simulation of Hand Hoard Management Under our simulation methodology, the three parameterized LRU-style managers (weighted, bounded, and linear LRU) performed more poorly than they would if the user were actively managing the hoard, since the control information is not modified for each disconnection period. This fact might be cited as an argument to invalidate our data, because one of the modified LRU methods could potentially outperform an automated system.

We believe that this argument is not relevant, because it assumes that the user is willing to accept a far greater burden than in our system. There is no question that an active, interested, and fully knowledgeable user can do a better job of hoarding than any automated system. Only the user knows what projects he plans to work on, and only the user can know that certain subsets of a large project are unneeded. Constant micro-management of the hoard contents will outperform any automated system *if* the user is intelligent and informed enough to make correct and complete decisions.

However, such hand management would call for far more knowledge and time than are available to all but a tiny fraction of users. The LRU-style managers are minor optimizations of hand management, and as such it is unrealistic to assume that the average non-expert user will do anything other than to create an initial hoard profile and forget it. In such an environment, our simulation methodology accurately reflects user behavior. In addition, our results (see Chapter 8) suggest that a good clustering algorithm will work better than all but the most diligent hand hoarding.

Measurement of Working Sets We considered two different methods of measuring working sets. The first reports the total sizes of all distinct files that were referenced during a disconnection period and that existed at the end of the period.[5] The second measure, which is the one we report in Chapter 8, reduces the first measure by the size of any completely new files created during the period. This smaller value is more relevant to the design of hoarding systems, since it is exactly equal to the size of the optimal hoard that would be produced by an oracle algorithm.

Measurement of Attention Shifts To report attention shifts, we first applied the definition given in Section 2.3 (p. 9). When we calculated the attention shifts, however, we discovered that the number of shifts was implausibly high.

After extensive investigation, we learned that the excess shifts were the result of a fundamental problem with the definition itself. A single period of inactivity can cause an attention shift to be registered, even if there was no change in the long-term set of files accessed. These "inactivity-caused shifts" because attention shifts are calculated by comparing adjacent periods, and any sudden increase in the number of files referenced

[5]The total size excludes temporary files, and thus slightly under-reports the peak disk space needed during the period. The underestimate is not a significant drawback for our purposes.

can legitimately be considered to be a shift. If the user works on a very large set of files for one disconnection period, becomes inactive for the next period, and returns to the original large set of files, the increase will be seen as a shift by the statistics-gathering process, even though neither the user nor SEER would consider that anything unusual had happened.

We believe that this problem showed up only in our current measurements, rather than in the earlier study reported in Chapter 2, because of user behavior differences. The earlier study was conducted in an office environment, using machines that did not run any background daemons. Users tended to work fairly steadily during the week, and not at all on weekends. The rare weekend worker would engage in almost as much activity as during the week. The result was that there were relatively few periods of light activity (zero-activity periods were ignored completely in that study).

In the current study, carried out in an academic environment, user behavior was much more sporadic. Some days would be devoted primarily to meetings, classes, reading, and other non-computer activities. Users would occasionally run a different operating system for part of the day, so that relatively little activity appeared in the traces. In addition, users normally took their machines home on weekends, and might engage in brief use in odd moments, either to do small amounts of work or for ancillary purposes such as retrieving a telephone number from an online directory. Finally, machines that were left idle, but unsuspended, sometimes ran `cron` daemons that would generate small amounts of activity. All of these factors contributed to the presence of low-activity periods in our traces.

One way to deal with the problem of low-activity disconnection periods would be to calculate attention shifts compared to the last (recent) period of similar size, essentially applying a smoothing function to the daily working set. In our case, smoothing was impractical due to the structure of our measurement software, so we chose a simpler alternative. We modified the attention-shift calculation to ignore shifts in which the first period contained references to a relatively small number of files. Thus, a low-activity period could not trigger an attention shift. This definition is imperfect, because a true attention shift preceded by an inactive period will not be detected. However, after querying our users about their past behavior, we believe that our measurements were not significantly affected by a failure to detect this type of attention shift.

We chose a threshold of 300 files accessed for daily disconnections periods, and 1000 for weekly periods. These values were chosen by examining the traces and the effects of various thresholds on the attention-shift counts. Although the values might seem high, they are actually a reflection of the complexity of the behavior of modern users and applications, and they work well in practice.

Live Methodology

To measure real users, we deployed SEER on the portable machines belonging to nine users working in the offices of the File Mobility Group (a subgroup of UCLA's Travler Project). Each user's machine was initially configured with a 50-Mb hoard, although one user later increased the hoard size to 98 Mb and another should have increased it. No restrictions were placed on user behavior. As well as collecting the traces we used in simulation, we installed several utility programs and daemons to gather statistics. These programs included:

- A daemon that pings a well-known site every 5 minutes, and records changes in accessibility in a log file. The results are used to detect network disconnections.
- A script that examines the `cron` logs[6] to discover times when the computer was in "sleep" or "suspend" mode. These times are subtracted from the record of disconnections, to get a better measure of when the computer was actually being used in a disconnected mode.
- A simple shell script that the user can run when he experiences a hoard miss,[7] which records the identity of the missed file and the perceived severity of the miss.

A post-processing script then examines the logs produced by these programs and generates the statistical summaries reported in Chapter 8.

[6]We run a local version of cron that is aware of times when the CPU is inactive, whether due to reboots or any other reason.
[7]Hoard misses are independently recorded by the `correlator`, but without severity information.

Chapter 8

Results

In previous chapters, we have presented a complete design for our automated hoarding system. We now review the results of experiments that demonstrate the effectiveness of our work.

8.1 Simulation Results

As discussed in Section 7.2.2, p. 80, we ran 120 simulations of disconnected operation with SEER under various conditions. We report the results of the simulations here. The simulations were driven by traces comprising approximately 2.5 Gb of collected data, and took about 17 days to run on a dedicated 200 MHz PENTIUM® Pro.

8.1.1 Working Sets

In our simulations, we calculated the working set for each disconnection period. As discussed in Section 7.1.2, p. 76, the working set provides a lower bound on the hoard size needed to operate without misses.

Daily Working Sets

Figure 8.1 shows an example of the development of daily working sets over time on the machine for which we have the longest trace, Norgay. This figure is taken from the first simulation run, using information from external investigators. The points labeled "daily" give the total size of files referenced during a given day. The points labeled "Cumulative" give the total size of files referenced since the beginning of the trace; the significance of this curve will become clear when we examine required hoard sizes in Section 8.1.2.

As can be seen from the graph, daily working sets for this machine are relatively small. The cumulative working set, on the other hand, increases steadily over time, with occasional jumps when a new project is attacked. Attention shifts are marked by arrows. Not every jump in the cumulative working set is associated with an attention shift; the lack of shifts at some jumps could be due either to our method of calculating attention shifts (see Section 7.2.2, p. 80) or to the creation of large numbers of new files, which do not contribute to attention shifts.

Figure 8.2 shows the same information for a second machine, Aldrin, taken from the second simulation run (without external investigators). This shorter trace shows similar characteristics, although with a less consistent increase in the cumulative working set.

Table 8.1 shows means and 99% confidence intervals (columns "C.I.") for daily working sets for all simulation runs. (We also calculated medians and quartile information, to verify that the distribution is approximately normal.) The column labeled "Inv.?" indicates whether external investigators were included in the simulation; the differences between investigated and non-investigated figures are caused by variations in the random seed.

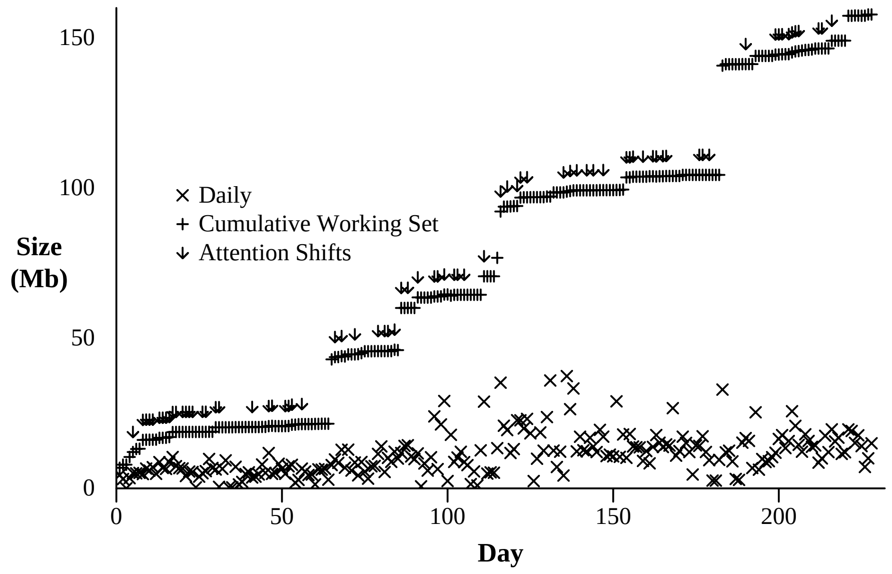


Figure 8.1: Daily Working Sets For Machine “Norgay” Versus Time, Run 1 (With Investigators)

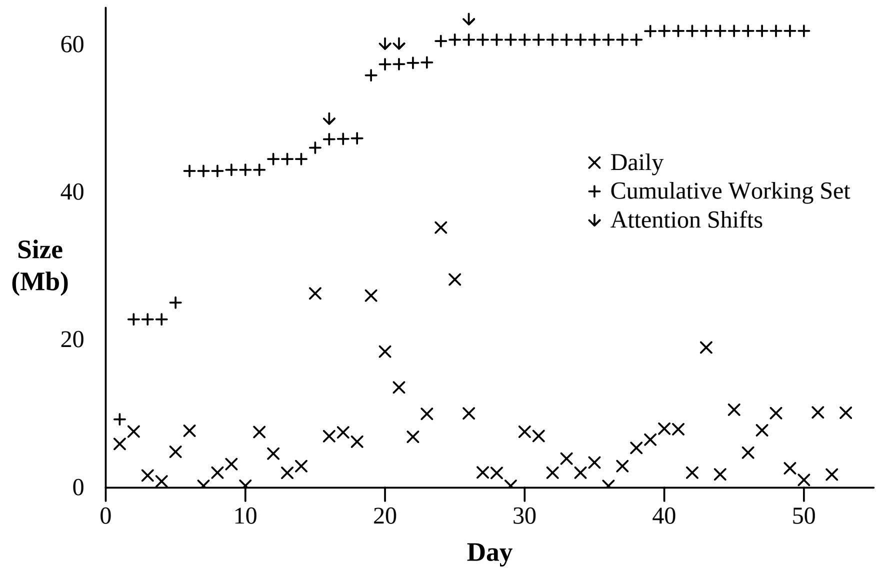


Figure 8.2: Daily Working Sets For Machine “Aldrin” Versus Time, Run 2 (No Investigators)

			W.S. (Mb)		
Machine	Inv.?	Periods	Mean	C.I.	
Aldrin	N	53	7	7	7
Chengho	Y	37	9	9	10
Chengho	N	37	9	9	9
Crockett	N	61	7	7	7
Erasmus	N	108	15	15	15
Muir	N	13	13	13	14
Norgay	Y	228	11	11	12
Norgay	N	228	11	11	12
Sacajawea	Y	112	20	20	21
Sacajawea	N	112	21	21	21
Spaulding	N	58	9	9	10
Yeager	N	138	15	15	16

Table 8.1: Daily Working-Set Statistics

Weekly Working Sets

Figure 8.3 gives a graph similar to Figure 8.1, showing the development of weekly working sets over time on machine Norgay. The graph contains no surprises; weekly working sets are similar to daily ones except for being slightly larger.

Table 8.2 shows means and 99% confidence intervals for weekly working sets for all machines.[1] (Again, the distributions approximate a Gaussian.)

8.1.2 Required Hoard Sizes

As discussed in Section 7.1.3, p. 78, one of the best ways to compare predictive hoarding systems is to calculate the hoard size needed to operate entirely without misses.

Required Hoard Sizes for Daily Disconnections

Figure 8.4 shows the sizes needed by the clustering and (plain) LRU managers for machine "Norgay" for simulated daily disconnections.[2] It is clear that in general the LRU manager requires an increasing amount of space over time for this user. We believe that this trend is partly due to the attention-shift phenomenon, and partly because even when a user doesn't shift his attention, there is a tendency to refer to an occasional old file.

By contrast, the clustering manager requires relatively constant space, and much less than the LRU manager. To make this difference easier to see, Figure 8.5 shows the same data, sorted by the LRU hoard size required to operate without misses. (The X scale is not shown in this figure because the ordinals are meaningless.) This figure makes it clear that the clustering manager requires essentially constant space even in the face of "ancient" references that cause the LRU manager to perform badly. This pattern was observed for all machines in the experiment; for example, Figure 8.6 shows a similar graph for machine Sacajawea.

Figure 8.7 shows a similar simulation run for machine Norgay, but without the benefit of external investigators. It is easy to see that even without investigators, the clustering manager significantly outperforms the LRU manager.

[1] The apparently contradictory results for machine "Muir," in which the daily working sets exceed the weekly ones, are an artifact of the short traces for this machine and the size of the warmup period. In practice, there is little overlap between the periods chosen for the daily and weekly traces, so that they are effectively two different sets of data.

[2] This graph is for simulation run number 1; other simulations show essentially identical results.

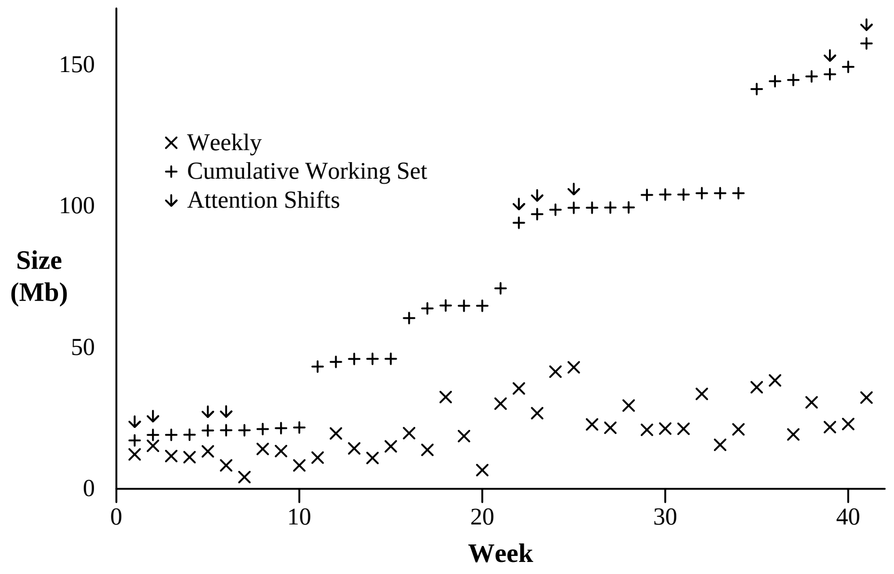


Figure 8.3: Weekly Working Sets For Machine "Norgay" Versus Time, Run 1 (With Investigators)

			W.S. (Mb)		
Machine	Inv.?	Periods	Mean	C.I.	
Aldrin	N	10	14	14	15
Chengho	Y	4	17	17	18
Chengho	N	4	17	17	17
Crockett	N	12	14	14	14
Erasmus	N	19	40	39	41
Muir	N	3	8	8	8
Norgay	Y	41	22	21	22
Norgay	N	41	22	21	22
Sacajawea	Y	17	50	49	50
Sacajawea	N	17	49	48	50
Spaulding	N	10	19	18	20
Yeager	N	22	34	33	36

Table 8.2: Weekly Working-Set Statistics (Mb)

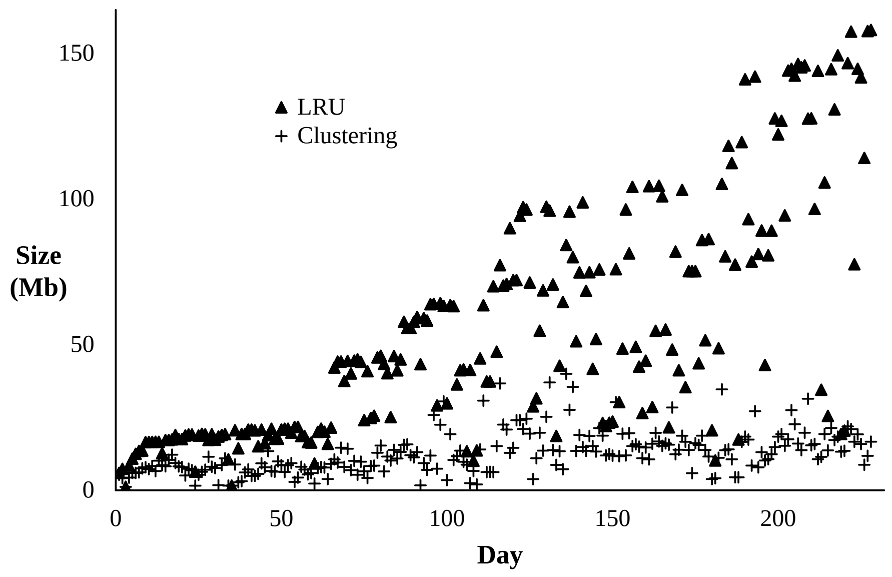


Figure 8.4: Daily LRU and Clustering Hoard Sizes for Machine "Norgay," Sorted by Day Number

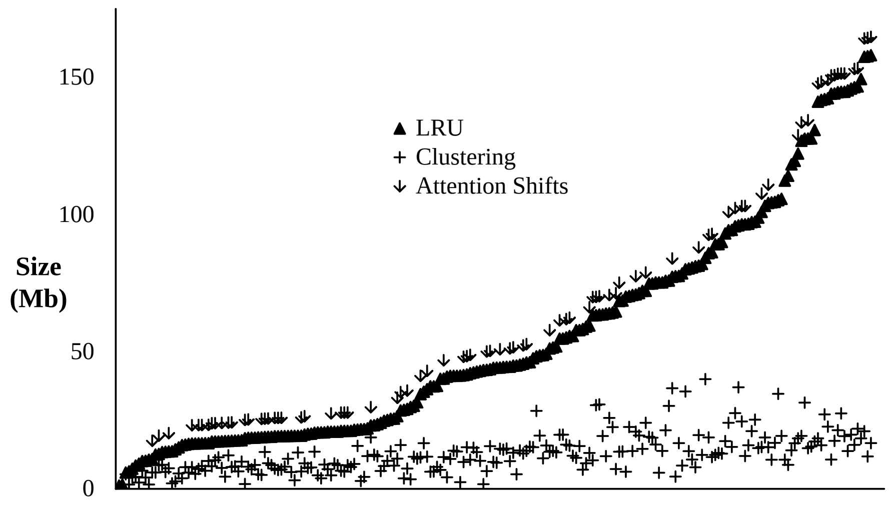


Figure 8.5: Daily LRU and Clustering Hoard Sizes for Machine "Norgay," Sorted by LRU Hoard Size, Run I-1

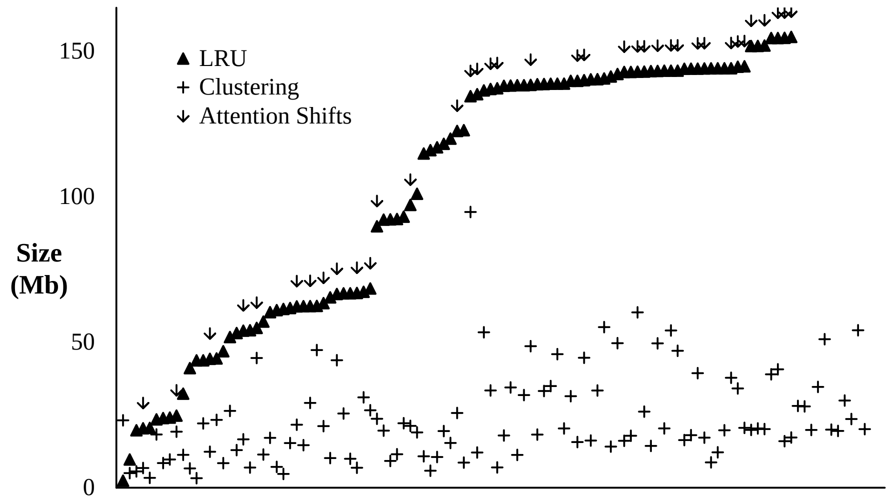


Figure 8.6: Daily LRU and Clustering Hoard Sizes for Machine "Sacajawea," Sorted by LRU Hoard Size, Run I-3

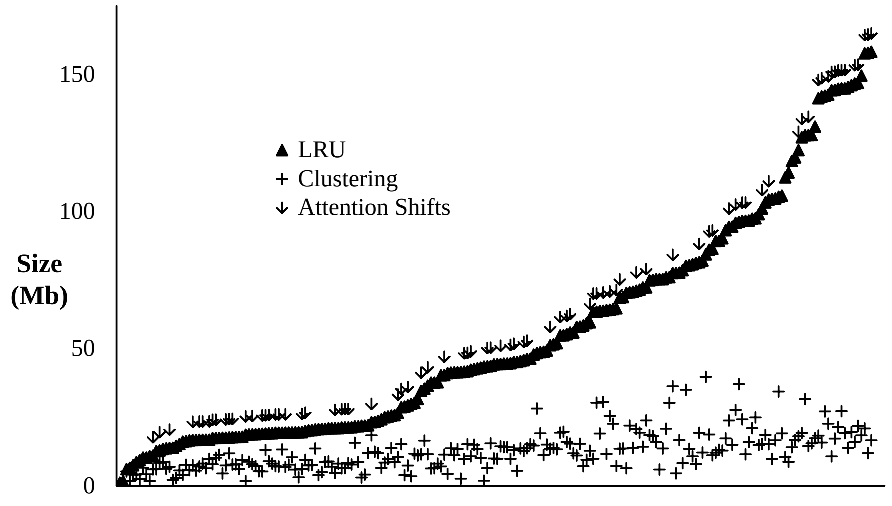


Figure 8.7: Daily LRU and Clustering Hoard Sizes for Machine "Norgay," Sorted by LRU Hoard Size, Run N-1

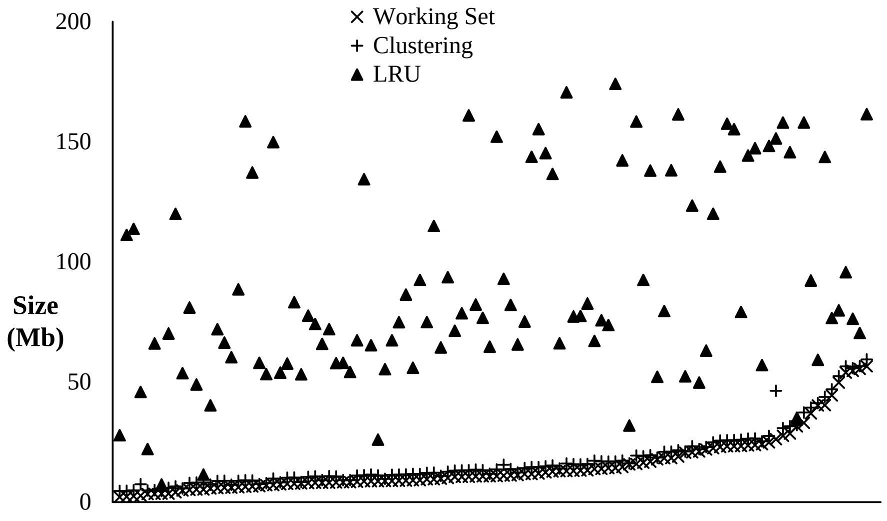


Figure 8.8: Daily Working Sets vs. LRU and Clustering Hoard Sizes for Machine "Erasmus," Sorted by Working Set Size, Run N-5

Perhaps the most impressive evidence of SEER's success can be seen by plotting the required hoard sizes against the corresponding working-set size. Figure 8.8 shows these values, sorted by working-set size for ease of comparison. It is easy to see that, while the LRU hoard manager generally requires far more hoard space than the working set, the clustering manager nearly always falls very close to this minimum.

Table 8.3 summarizes this information by showing means and 99% confidence intervals for required daily hoard sizes for all machines. For each hoarding method, the columns labeled "Cost" indicate the percentage of space required over the optimum (which is the working set). The column labeled "L" gives the length of the simulation run, in disconnection periods, and "Inv.?" indicates whether external investigators were used. The same table also shows the mean, confidence interval, and range for the difference between the two hoarding methods. The final column gives the percentage of days on which the LRU manager outperformed the clustering manager (difference less than or equal to zero). It is interesting to note that, although the LRU manager occasionally outperforms the clustering manager, it does so rarely and by only an extremely small margin.

One surprising result in Table 8.3 is that the presence of external investigators did not significantly improve SEER's performance. It was this observation that caused us to modify our experimental plan to discard external investigators in some of the simulation runs.

Required Hoard Sizes for Weekly Disconnections

Figure 8.9 shows the sizes needed by the clustering and (plain) LRU managers for machine "Norgay" for simulated weekly disconnections, sorted by the LRU hoard size required to operate without misses. Note the similarity of this graph to the equivalent daily graph in Figure 8.5. Again, the clustering manager consistently and significantly outperforms the LRU manager.

Table 8.4 summarizes this information in the same manner as Table 8.3. Note that the only machine with a significant percentage of LRU "victories" (Chengho) was also the one that had the shortest trace, so that the absolute number of victories (1) was still small. Also note that, as with the daily results, the LRU manager never outperformed clustering by a large amount.

			LRU				Clustering				Difference					% LRU
Machine	L	Inv.?	Mean	C.I.		Cost	Mean	C.I.		Cost	Mean	C.I.		Min	Max	Wins
Aldrin	53	N	27	26	29	287	8	8	9	20	19	18	20	-2	58	4
Chengho	37	Y	29	28	30	211	10	10	10	8	19	18	19	-1	52	8
Chengho	37	N	29	29	29	215	10	10	10	8	19	19	19	-1	52	8
Crockett	61	N	27	26	28	311	8	8	8	22	19	18	20	-3	47	8
Erasmus	108	N	91	91	91	494	18	18	18	16	73	73	73	3	157	0
Muir	13	N	68	65	70	413	14	14	15	8	53	51	55	18	142	0
Norgay	228	Y	57	55	58	395	13	13	13	14	44	43	45	0	150	1
Norgay	228	N	57	56	58	395	13	13	13	12	44	43	45	0	151	1
Sacajawea	112	Y	110	107	114	442	24	23	24	15	87	84	90	-21	186	1
Sacajawea	112	N	112	112	112	441	24	24	24	15	88	88	88	-21	186	1
Spaulding	58	N	28	26	29	199	10	10	10	9	18	17	18	0	55	0
Yeager	138	N	99	97	100	539	17	17	18	12	81	80	82	4	197	0

Table 8.3: Daily Hoard-Size Statistics

				LRU				Clustering				Difference				% LRU
Machine	L	Inv.?	Mean	C.I.		Cost	Mean	C.I.		Cost	Mean	C.I.		Min	Max	Wins
Aldrin	10	N	43	42	45	206	15	15	16	8	28	27	29	16	50	0
Chengho	4	Y	29	28	29	65	18	18	18	3	11	10	11	0	28	25
Chengho	4	N	29	28	29	67	18	17	18	3	11	11	12	0	28	25
Crockett	12	N	42	42	42	202	15	15	15	7	27	27	27	4	45	0
Erasmus	19	N	110	108	111	176	43	42	44	9	66	66	67	3	129	0
Muir	3	N	89	89	89	1041	9	9	10	21	79	79	79	46	127	0
Norgay	41	Y	72	71	73	233	24	23	24	9	48	48	49	1	126	0
Norgay	41	N	72	71	73	234	23	23	24	8	49	48	50	1	126	0
Sacajawea	17	Y	137	136	138	176	52	52	53	6	84	84	85	3	186	0
Sacajawea	17	N	135	130	139	176	52	50	53	5	83	80	86	3	187	0
Spaulding	10	N	50	48	52	165	20	18	21	4	30	29	31	18	52	0
Yeager	22	N	123	121	125	258	36	35	38	5	87	86	88	-3	183	2

Table 8.4: Weekly Hoard-Size Statistics

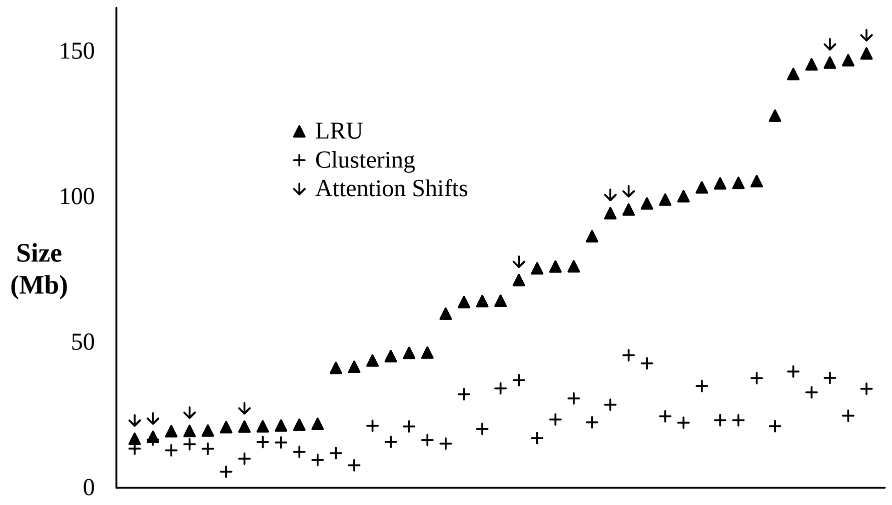


Figure 8.9: Weekly LRU and Clustering Hoard Sizes for Machine "Norgay," Sorted by LRU Hoard Size

	Shifts		Shift				Unshift			
Machine	N	%	Diff.	C.I.		Wins	Diff.	C.I.		Wins
Aldrin	4	8	30	28	31	0	18	17	19	10
Chengho	4	11	26	26	26	0	18	18	19	15
Crockett	0	0	—	—	—	—	19	18	20	25
Erasmus	32	30	82	82	82	0	70	70	70	0
Muir	4	31	28	27	30	0	65	62	67	0
Norgay	82	36	50	49	51	0	40	39	41	10
Sacajawea	42	38	94	94	94	0	85	85	85	5
Spaulding	1	2	25	25	25	0	17	16	18	0
Yeager	36	26	85	84	87	0	80	79	81	0

Table 8.5: Attention Shifts for Daily Disconnections

8.1.3 Attention Shifts

Table 8.5 summarizes attention shifts for daily disconnections. For each machine, the table gives the number and percentage of attention shifts observed, and breaks down the "Wins" column and the mean value and confidence intervals of the "Difference" column from Table 8.3 into separate values for days with and without ("Unshift") attention shifts. Table 8.6 provides the same information for weekly disconnections. As hypothesized, the LRU algorithm never outperforms the clustering method when attention shifts occur.

8.2 Usage Experience

8.2.1 Statistics

Table 8.7 gives statistics on the general disconnection behavior of actual users.[3] There are a few important

	Shifts			Shift				Unshift		
Machine	N	%	Diff.	C.I.		Wins	Diff.	C.I.		Wins
Aldrin	2	20	27	26	27	0	28	27	30	0
Chengho	1	25	10	10	10	0	11	11	12	5
Crockett	0	0	—	—	—	—	27	27	27	0
Erasmus	8	42	83	82	84	0	54	54	55	0
Muir	0	0	—	—	—	—	79	79	79	0
Norgay	9	22	45	45	46	0	50	49	51	0
Sacajawea	6	35	45	43	47	0	104	100	108	0
Spaulding	0	0	—	—	—	—	50	48	52	0
Yeager	3	14	30	30	30	0	96	95	97	2

Table 8.6: Attention Shifts for Weekly Disconnections

	Number of			Duration (Hours)				
Machine	Disconn's	Mean	Sdev	Min	Max	Quartiles		
Aldrin	38	11.16	15.82	0.26	71.89	1.03	3.24	15.33
Chengho	10	43.20	127.19	0.43	404.94	0.43	0.57	1.47
Crockett	75	9.94	40.87	0.26	348.20	0.35	1.12	4.88
Erasmus	90	3.01	4.46	0.26	26.50	0.60	1.38	3.19
Muir	25	1.87	2.54	0.26	12.08	0.52	0.81	1.62
Norgay	184	9.30	16.33	0.26	90.62	1.01	2.00	12.19
Sacajawea	107	8.06	38.29	0.26	390.60	0.78	1.47	4.34
Yeager	116	2.36	4.26	0.25	27.68	0.43	0.78	1.81

Table 8.7: Disconnection Statistics

points to note about the measurement methodology and user behavior:

- Disconnection duration only considers the time that the computer is actively running while disconnected. This restriction is the reason for the relatively short disconnection intervals for some machines (*e.g.*, "Norgay" has a median disconnection time of only 2.00 hours, even though its user disconnected overnight and across weekends on an almost continuous basis).
- Disconnections of less than 15 minutes were ignored in the analysis, for two reasons. First, short disconnections tend to reflect anomalous situations such as brief network failures rather than true disconnections (the shortest observed was a mere two seconds, which is hardly meaningful for a predictive hoarding system). Second, we feel that a hoard miss during a short disconnection is rarely debilitating to the user. (As it happened, none of the observed hoard misses occurred during disconnection periods of less than 15 minutes.) However, the elimination of unreasonably short disconnections does have an effect on the minimum-duration column in the table above.
- Some users rarely suspend their computers when not actively traveling, so their average disconnection durations are much longer.

The most striking information in Table 8.7 is the highly skewed distribution (the median is far below the mean in every case, and the maxima are far greater than the mean and usually much greater than the third quartile). For this reason, we believe that it is better to characterize disconnection behavior with the median rather than the mean.

Table 8.8 summarizes statistics on failed disconnections (those in which there was at least one hoard miss). For each machine and severity level, the table gives the hoard size used (in megabytes), the absolute number of failures and the percentage of the total failed disconnections, and the automatically detected failure count and percentage (automatically detected failures do not depend on user input, but SEER cannot detect failures where the user becomes aware of a missing file by looking at a directory listing). The "Total" column does not give the sum of all severity levels, but rather the total number of failed disconnections (excluding automatically detected failures); this can differ from the sum when a particular disconnection experienced failures at more than one severity level.

Table 8.9 gives the time to first miss, in hours and in percentage of total disconnection period, for failed disconnections (to save space, only nonzero rows are listed). Note that in some cases there were too few misses to generate meaningful quartiles. The number of misses is listed in Table 8.8.

Table 8.10 gives statistics on the time to first miss, in hours and in percentage of total disconnection period, averaged across all disconnections.[4]

8.2.2 Discussion

The simplest evidence of SEER's success is found in the "Percentage" columns of Table 8.8. Most users reported zero or miniscule failure rates; even the automatically-detected failures ("Auto" column), which went unnoticed by users, show that 94% or more of actual disconnections were miss-free. The sole exception was machine "Norgay," whose user reported a total of 13% failures, although only 4.9% were at significant severities. When we investigated this high failure rate, we found that the user of this machine frequently had a working set that exceeded the hoard size of 50 Mb (see Figure 8.1). In the light of this observation, a 13% failure rate is actually doing very well. The user of this machine has since increased his hoard size to 100 Mb, which has caused his failure rate to drop to the same negligible level reported by the other users.

From Table 8.9, it can be seen that users were generally able to work for significant periods before being affected by a hoard miss. It is interesting to note that the time to first miss usually *increases* as the severity level becomes more intrusive (smaller severity codes), another indication that automated predictive hoarding does not inconvenience the user significantly.

[3]The live-usage data for machine "Spaulding" was lost through a system administration error.

[4]The "minimum" and "maximum" columns in this table primarily represent the minimum and maximum disconnection times given in Table 8.7.

	Hoard	Failures							Percentage						
Machine	Size	0	1	2	3	4	Total	Auto	0	1	2	3	4	Total	Auto
Aldrin	50	0	0	0	0	0	0	2	0	0	0	0	0	0	5.3
Chengho	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Crockett	50	0	0	0	0	0	0	1	0	0	0	0	0	0	1.3
Erasmus	50	0	0	0	0	0	0	5	0	0	0	0	0	0	5.6
Muir	50	0	0	0	0	0	0	1	0	0	0	0	0	0	4
Norgay	50	0	3	6	11	9	24	2	0	1.6	3.3	6.0	4.9	13.0	1.1
Sacajawea	98	0	0	0	0	0	0	3	0	0	0	0	0	0	2.8
Yeager	50	0	1	0	0	0	1	5	0	0.9	0	0	0	0.9	4.3

Table 8.8: Summary of Failed Disconnections at Various Severities

		Hours							Percentage						
Machine	Sev.	Mean	Sdev	Min	Max	Quartiles			Mean	Sdev	Min	Max	Quartiles		
Aldrin	Auto	1.8	2.3	0.21	3.4	—	—	—	43.5	59.7	1.3	85.7	—	—	—
Crockett	Auto	1.6	—	1.6	1.6	—	—	—	48.8	—	48.8	48.8	—	—	—
Erasmus	Auto	0.9	0.5	≈ 0	1.3	≈ 0	1.0	1.0	42.7	43.5	8.5	98.6	8.5	11.6	13.9
Muir	Auto	11.0	0	11.0	11.0	—	—	—	90.8	0	90.8	90.8	—	—	—
Norgay	1	10.6	16.3	≈ 0	29.4	—	—	—	49.9	49.6	0.2	99.3	—	—	—
	2	6.6	9.1	≈ 0	21.5	≈ 0	0.9	2.5	50.1	30.8	0.2	94.8	0.2	46.0	52.5
	3	3.4	4.9	0.1	12.9	0.2	0.5	3.1	49.9	35.5	0.5	95.5	1.8	35.6	74.6
	4	6.2	11.2	0.1	29.3	0.3	0.5	0.7	45.1	40.9	1.7	100.0	3.8	6.0	69.1
	Auto	20.4	28.4	0.3	40.5	—	—	—	54.0	57.1	13.6	94.4	—	—	—
Sacajawea	Auto	0.5	0.3	0.2	0.8	—	—	—	37.4	24.1	21.4	65.1	—	—	—
Yeager	1	1.0	—	1.0	1.0	—	—	—	20.8	—	20.8	20.8	—	—	—
	Auto	0.9	0.6	0.1	1.8	0.1	0.6	0.9	32.2	24.0	5.7	60.2	5.7	8.4	42.6

Table 8.9: Times to First Miss for Failed Disconnections

		Hours							Percentage						
Machine	Sev.	Mean	Sdev	Min	Max	Quartiles			Mean	Sdev	Min	Max	Quartiles		
Aldrin	0	11.2	15.8	0.3	71.9	1.0	3.2	15.3	100	0	100	100	100	100	100
	1	11.2	15.8	0.3	71.9	1.0	3.2	15.3	100	0	100	100	100	100	100
	2	11.2	15.8	0.3	71.9	1.0	3.2	15.3	100	0	100	100	100	100	100
	3	11.2	15.8	0.3	71.9	1.0	3.2	15.3	100	0	100	100	100	100	100
	4	11.2	15.8	0.3	71.9	1.0	3.2	15.3	100	0	100	100	100	100	100
	Auto	10.7	15.9	0.2	71.9	1.0	2.5	13.3	97.0	16.1	1.3	100	100	100	100
Chengho	0	43.2	127.2	0.4	404.9	0.4	0.6	1.5	100	0	100	100	100	100	100
	1	43.2	127.2	0.4	404.9	0.4	0.6	1.5	100	0	100	100	100	100	100
	2	43.2	127.2	0.4	404.9	0.4	0.6	1.5	100	0	100	100	100	100	100
	3	43.2	127.2	0.4	404.9	0.4	0.6	1.5	100	0	100	100	100	100	100
	4	43.2	127.2	0.4	404.9	0.4	0.6	1.5	100	0	100	100	100	100	100
	Auto	43.2	127.2	0.4	404.9	0.4	0.6	1.5	100	0	100	100	100	100	100

Table 8.10: Times to First Miss Across All Disconnections *(continued on following pages)*

		Hours							Percentage						
Machine	Sev.	Mean	Sdev	Min	Max	Quartiles			Mean	Sdev	Min	Max	Quartiles		
Crockett	0	9.9	40.9	0.3	348.2	0.3	1.1	4.9	100	0	100	100	100	100	100
	1	9.9	40.9	0.3	348.2	0.3	1.1	4.9	100	0	100	100	100	100	100
	2	9.9	40.9	0.3	348.2	0.3	1.1	4.9	100	0	100	100	100	100	100
	3	9.9	40.9	0.3	348.2	0.3	1.1	4.9	100	0	100	100	100	100	100
	4	9.9	40.9	0.3	348.2	0.3	1.1	4.9	100	0	100	100	100	100	100
	Auto	9.9	40.9	0.3	348.2	0.3	1.1	4.9	99.3	5.9	48.8	100	100	100	100
Erasmus	0	3.0	4.5	0.3	26.5	0.6	1.4	3.1	100	0	100	100	100	100	100
	1	3.0	4.5	0.3	26.5	0.6	1.4	3.1	100	0	100	100	100	100	100
	2	3.0	4.5	0.3	26.5	0.6	1.4	3.1	100	0	100	100	100	100	100
	3	3.0	4.5	0.3	26.5	0.6	1.4	3.1	100	0	100	100	100	100	100
	4	3.0	4.5	0.3	26.5	0.6	1.4	3.1	100	0	100	100	100	100	100
	Auto	2.8	4.2	0.0	26.5	0.6	1.3	2.8	96.8	16.1	8.5	100	100	100	100

Table 8.10 *(continued)*: Times to First Miss Across All Disconnections

		Hours							Percentage						
Machine	Sev.	Mean	Sdev	Min	Max	Quartiles			Mean	Sdev	Min	Max	Quartiles		
Muir	0	1.9	2.5	0.3	12.1	0.5	0.8	1.7	100	0	100	100	100	100	100
	1	1.9	2.5	0.3	12.1	0.5	0.8	1.7	100	0	100	100	100	100	100
	2	1.9	2.5	0.3	12.1	0.5	0.8	1.7	100	0	100	100	100	100	100
	3	1.9	2.5	0.3	12.1	0.5	0.8	1.7	100	0	100	100	100	100	100
	4	1.9	2.5	0.3	12.1	0.5	0.8	1.7	100	0	100	100	100	100	100
	Auto	1.8	2.4	0.3	11.0	0.5	0.8	1.7	99.6	1.8	90.8	100	100	100	100
Norgay	0	9.3	16.3	0.3	90.6	1.0	2.0	12.2	100	0	100	100	100	100	100
	1	9.3	16.3	0.0	90.6	1.0	2.0	12.2	99.2	8.2	0.2	100	100	100	100
	2	9.1	16.2	0.0	90.6	0.9	2.0	11.8	98.4	10.2	0.2	100	100	100	100
	3	9.0	16.3	0.1	90.6	0.9	2.0	11.1	97.0	14.5	0.5	100	100	100	100
	4	8.8	16.0	0.1	90.6	0.9	2.0	11.1	97.3	14.6	1.7	100	100	100	100
	Auto	9.3	16.3	0.3	90.6	1.0	1.0	12.2	99.5	6.4	13.6	100	100	100	100

Table 8.10 *(continued)*: Times to First Miss Across All Disconnections

		Hours							Percentage						
Machine	Sev.	Mean	Sdev	Min	Max	Quartiles			Mean	Sdev	Min	Max	Quartiles		
Sacajawea	0	8.1	38.3	0.3	390.6	0.8	1.5	4.3	100	0	100	100	100	100	100
	1	8.1	38.3	0.3	390.6	0.8	1.5	4.3	100	0	100	100	100	100	100
	2	8.1	38.3	0.3	390.6	0.8	1.5	4.3	100	0	100	100	100	100	100
	3	8.1	38.3	0.3	390.6	0.8	1.5	4.3	100	0	100	100	100	100	100
	4	8.1	38.3	0.3	390.6	0.8	1.5	4.3	100	0	100	100	100	100	100
	Auto	8.0	38.3	0.2	390.6	0.7	1.4	4.3	98.2	10.9	21.4	100	100	100	100
Yeager	0	2.4	4.3	0.3	27.7	0.4	0.8	1.8	100	0	100	100	100	100	100
	1	2.3	4.3	0.3	27.7	0.4	0.8	1.8	99.3	7.4	20.8	100	100	100	100
	2	2.4	4.3	0.3	27.7	0.4	0.8	1.8	100	0	100	100	100	100	100
	3	2.4	4.3	0.3	27.7	0.4	0.8	1.8	100	0	100	100	100	100	100
	4	2.4	4.3	0.3	27.7	0.4	0.8	1.8	100	0	100	100	100	100	100
	Auto	2.2	4.2	0.1	27.7	0.4	0.8	1.7	97.1	14.5	5.7	100	100	100	100

Table 8.10 *(concluded)*: Times to First Miss Across All Disconnections

The data in Table 8.10 is even more encouraging. Since the average disconnection is miss-free, the mean time to first miss over all disconnections is nearly equal to the disconnection times given in Table 8.7. For the same reason, all of the percentage quartiles are equal to 100%.

Tables 8.5 and 8.6 again highlight the superiority of the clustering algorithm over LRU. LRU performs reasonably well when there are no attention shifts, and sometimes even outperforms clustering. When there are shifts, however, LRU rarely outperforms clustering and, as seen in Tables 8.3 and 8.4, never by a significant margin. Furthermore, the clustering algorithm generally achieves performance that is within a few percent of the optimum ("Cost" column), while the LRU manager often requires space several hundred percent greater than the optimum.

The poor performance of LRU is due to its simplistic approach to the problem of hoarding. LRU is only able to hoard all files referenced since a certain time in the past. Thus, if a user wishes to work on a file last accessed several months previously, an LRU-style algorithm must hoard every file used since that point, regardless of whether files referred to in the interim are currently of interest. Thus, accessing a single "old" file will force LRU to hoard hundreds or thousands of unneeded files, which in turn requires an overly large hoard size. (In practice, a hoarding system usually starts with a given amount of space, and then works backwards in time until the hoard is full. In this case, an LRU algorithm will not be able to hoard the "old" file unless the hoard is large enough to also contain all of the unwanted material.)

It is easy to see from the data in these tables that the system has been a success in actual use. This conclusion is consistent with reports from users, who have generally been very pleased with SEER's ability to correctly predict their future activities.

8.3 Performance Impact

Since SEER constantly collects information on user behavior so that it can make its predictions, there is an unavoidable impact on performance. This section quantifies that impact, dividing it into several components, and discusses the implications for users.

All measurements were carried out on a Texas Instruments TRAVELMATE^{TM} 6030, which is a PENTIUM-based laptop machine. The particular configuration used operated at 133 MHz and had 64 Mb of RAM, running the LINUX operating system. Elapsed times within the kernel were collected using the `kitrace` measurement tool [Kuenning 1995].

8.3.1 Kernel Performance

System Calls

As discussed in Section 5.2, p. 52, a small hook is placed into the operating-system kernel to allow SEER to observe selected system calls. Due to the nature of this hook, it is executed for every system invocation, whether or not the particular call will be traced.

The cost of this hook has several components. First, we investigated the average time added to the system-call path, calculated by measuring the entry to and exit from the observation routine, without regard to whether a trace entry was recorded. After subtracting out the overhead due to measurement,[5] we found that the average time was 0.42 μs, measuring both traced and untraced calls and weighting the average by the observed frequency of traced calls.[6] Although this average will vary depending on the exact mix of traced and untraced calls, it is certainly safe to conclude that SEER adds only about half a microsecond to the cost of an untraced system call on this computer.

Traced system calls are significantly more expensive, costing 26 μs to capture the arguments into a buffer.[7]

[5] About 13.5 μs on this machine.

[6] The 99% confidence interval was 0.3 μs about the mean, due primarily to inaccuracies in measurement.

[7] The 99% confidence interval is 2.8 μs about the mean; the width is due to variations in the number of arguments captured.

Trace Buffer Access

Whenever system-call trace entries are recorded, the `observer` reads them from the kernel trace buffer and forwards them to the `correlator`. We measured the cost of reading entries from the trace buffer by tracing the routine that copies these entries into user space.[8] The average size read was 34 bytes,[9] with a narrow confidence interval of 1.4 bytes. The average cost of reading these entries was only 21.9 μs each.[10]

Summary

From the above figures, we can see that the total kernel cost of tracing a system call is very small. An untraced call suffers a negligible slowdown. A traced call is more expensive, requiring a total of about 35 μs to capture a trace entry and later copy it to the observer. However, the only calls that are traced are themselves relatively expensive in terms of elapsed time; for example, an `open` call must resolve a path name, lock and modify several internal data structures, and usually access the disk. In this context, 35 μs does not cause a noticeable performance impact for the user.

8.3.2 User-Level Impact

The user-level programs comprising SEER are not currently optimized for performance. Instead, they were designed for research flexibility, and many design decisions were made that sacrificed performance in favor of ease of development and modification.

The most significant example of this decision was the choice to store the file table in the `correlator`'s memory, rather than on disk. We knew when we made this decision that it would speed development at the cost of possible performance problems. As a result, the current `correlator` is something of a memory hog; it requires about 1 Mb of virtual space for each 1000 files, which translates to 10-20 Mb for the ten to twenty thousand files that it tracks on behalf of a typical user. Most of this size is occupied by data structures needed by the clustering manager; LRU management alone would be significantly less expensive.

The size of the database has been a problem on smaller machines, but current portables with 32 megabytes or more of RAM are not noticeably affected by the large database. Nevertheless, we feel that a production design should store the database more efficiently and on disk, and we plan to do so with SEER's `correlator`.

Other than virtual memory, the `observer` and `correlator` do not consume significant resources in normal operation. The notable exception is during hoarding; the clustering manager needs about 2 minutes of CPU time on a 133 MHz PENTIUM processor to complete its hoarding decisions (the LRU-style managers generally take only a few seconds). However, our users have not found clustering performance to be a significant problem, because changing the state of the hoard is a relatively infrequent operation whose total time is dominated by the time needed for the underlying replication system to transport files and update internal state. Nevertheless, we plan to optimize this operation in the future.

[8] There is a small additional overhead, which we did not measure, for dispatching the system call that reads the entries.

[9] The small read size implies that the observer was normally able to track the trace buffer closely, reading each trace entry as soon as it was collected.

[10] The 99% confidence interval was extremely narrow, only 0.1 μs.

Chapter 9

Related Work

Predictive hoarding for mobility builds on two existing concepts. First is the idea of caching memory or disk information to improve performance, including recent work intended to predict future access patterns to reduce the impact of low bandwidth or long latencies for remote disk accesses. Second is the development of updatable replicated file systems. We will briefly survey the work in each of these areas, which serves as a foundation for disconnected operation and for our own research.

Several previous researchers have also built systems that allow disconnected operation, most of which provide some sort of automated or semi-automated hoarding support, although none is as ambitious as SEER. After considering the foundation work, we will discuss these other systems in detail.

9.1 Foundations

9.1.1 Predictive Caching

Caches are a fundamental concept in computer science, and much work has been done on the subject of caching. Early work concentrated on improving the apparent speed of main memory with a small cache; later systems (such as UNIX) used the same idea to improve file-system performance. Nearly every modern computer system uses some form of cache as an integral part of its operation.

More recently, the advent of networked file systems has increased the performance gap between local and remote storage, with an attendant increase in the cost of a cache miss. Many researchers have investigated the use of prediction to ameliorate this cost [Blaze and Alonso 1992, Griffioen and Appleton 1994, Grimsrud *et al*. 1993, Howard *et al*. 1988, Korner 1990, Kotz and Ellis 1990, Kroeger and Long 1996, Lazowska *et al*. 1986, Lei and Duchamp 1997, Muntz and Honeyman 1992, Palmer and Zdonik 1991, Schroeder *et al*. 1985, Staelin and Garcia-Molina 1990, Tait and Duchamp 1991, Vitter and Krishnan 1996]. These schemes generally interpose an agent between the application software and the file system. This agent observes and records the disk access patterns of the applications. At a later time, when the beginning of a pattern is repeated, the agent predicts that later accesses will continue in the same pattern, and pre-fetches appropriate data from the file system. The hope is that these accesses will retrieve data that will actually be used, reducing apparent latency and increasing perceived performance. (This approach is also related to the prefetching of instructions and data in conventional memory caches, but because of their dependence on fixed-size objects and spatial reference patterns as a prediction factor, the methods produced by that work are not easily applied to the current problem.)

Another closely-related field is file migration in supercomputer centers [Buck and Coyne 1991, Drakopoulos and Merges 1991, Foglesong *et al*. 1990, Foster and Habermehl 1991, Hogan *et al*. 1990, Kohl *et al*. 1993, Kure 1988, Mecozzi and Minton 1991, Nydick *et al*. 1991, Peterson 1991, Smith 1981]. These investigators are generally concerned with creating the illusion of infinite storage, rather than with improving the apparent performance of a remote disk. As such, their interest tends to be centered more on the question of which files should be discarded from the high-speed portion of the cache because they will be unused in the next several hours or days, rather than predicting which files or parts of files will be needed in the next few minutes. Although this problem is more closely related to hoarding, the concentration on ejection of undesired data, rather than on selection of desired items, again makes it difficult to directly apply this work.

A third approach to predictive caching is to ask the client process to provide extra information to the caching subsystem [Cao *et al.* 1994, Ebling *et al.* 1994, Gibson *et al.* 1992, Patterson *et al.* 1995, Steere and Satyanarayanan 1994]. Presumably, the client knows what files it needs, and thus can make more intelligent predictions than a system that tries to make inferences. However, this approach is impractical for hoarding, for several reasons. First, it assumes that application designers are willing to modify their code to support disconnected operation, an assumption unsupported by the realities of the commercial marketplace. Second, it assumes that application designers are actually knowledgeable about the files their application needs. In the modern world of complex GUIs and powerful library functions, this assumption is frequently untrue. Third, a system that depends on the application will break down if the user runs an application that does not know how to cooperate. Since an alternative approach will still be needed to deal with this situation, we believe that it is better to simply design the alternative approach well enough that the burden of specification can be completely lifted from the application designers.

9.1.2 Replicated File Systems

The whole idea of hoarding for mobility presumes that the physical location of a particular file may change from moment to moment, depending on the needs of its users. It also presumes that files are shared, and that shared updates are allowed without regard to location.

The first characteristic can be achieved by providing a location-independent file naming service together with an automated file migration facility. Such systems are common in large supercomputer installations [Buck and Coyne 1991, Drakopoulos and Merges 1991, Foglesong *et al.* 1990, Foster and Habermehl 1991, Hogan *et al.* 1990, Kohl *et al.* 1993, Kure 1988, Mecozzi and Minton 1991, Nydick *et al.* 1991, Peterson 1991, Smith 1981]. However, shared accesses and updates in such a system are only possible if the physical location of file is dynamically accessible from any application that needs to use the file.

In general, the only solution to making a file available to multiple users in the face of a partitioned network (such as a disconnected mobile computer) is to replicate the file. If the file is to be updatable regardless of where the user is located, the replicated file system must allow disconnected updates. Several such systems have been built and proven workable in the last few years [Guy 1991, Kumar and Satyanarayanan 1995, Popek and Walker 1985, Reiher *et al.* 1994, Reiher *et al.* 1996, Satyanarayanan *et al.* 1990]. Most of these systems could serve as a base for our predictive-hoarding work, since our system has been designed so that it is largely independent of the underlying replication system.

9.2 Systems for Disconnected Operation

Several investigators have built prototype mobile systems based on existing replicated file systems [Alonso *et al.* 1990, Huston and Honeyman 1993, Kistler 1993, Skopp and Kaiser 1993, Tait *et al.* 1995]. However, with the exception of Tait's SPY UTILITY [Tait *et al.* 1995], these systems place most or all of the file-specification burden directly on the user.

9.2.1 Coda

The first system providing significant support for disconnected operation appears to be CODA [Kistler and Satyanarayanan 1992, Kistler 1993, Kumar and Satyanarayanan 1993, Mummert *et al.* 1995, Satyanarayanan *et al.* 1993]. Much of the effort in CODA was directed toward replication, specifically the problem of allowing disconnected writes as in FICUS [Guy 1991, Heidemann *et al.* 1992], but its most significant contribution was the provision of mechanisms that, while connected, could semi-automatically hoard files that might be used after disconnection. The CODA hoard manager decided which files to keep by integrating LRU data with hints specified by the user.[1] Although the system placed nontrivial burdens on the user, who was expected to guide the hoarding process, it was successful and flexible enough to allow useful disconnected work for periods of over a week. It was the success of CODA's semi-automated hoarding that encouraged others, including us, to pursue a more automated approach.

Because CODA is the system most often discussed in the literature, we will devote more attention to it than to other related work.

Overview of Coda

CODA was an outgrowth of the Andrew File System (AFS) [Howard *et al.* 1988], which was an early distributed file system that did not support optimistic replication. A major goal of the project was to extend AFS to the mobile environment by supporting disconnected operation [Kistler 1993, Section 2.2.1]. CODA used a client/server architecture, with the servers replicated for reliability. Clients, which could be fixed workstations or portable machines, cached whole files retrieved from the servers. While clients were connected, the server maintained a *callback list* of all clients who cached a particular file; when the file was updated, the server notified all clients that their local copy was no longer valid. (This part of the design was inherited directly from AFS).

When a client was disconnected from the server network, callbacks were no longer possible. Instead, a local pseudo-server emulated the server, and any file changes were logged rather than being sent directly to the server. When the connection returned, the log was replayed in a process called *reintegration.* If conflicting updates occurred during the disconnection period, they were resolved as part of the integration process when possible, and otherwise postponed for later user resolution.

Before disconnection, CODA attempted to fill the hoard with files that would be useful to the user. These files were chosen according to a formula [Kistler 1993, Section 5.3.2.1] that combined LRU information with user-provided adjustments. The hoard was automatically filled ("walked") at regular intervals, or alternatively the user could explicitly request that the hoard be filled. The hoard was generally kept as full as possible, within the limits of local disk space.

The user-provided adjustments to LRU information were specified using so-called *hoard profiles,* stored in `.hoardrc` files. The system supported dynamic loading and unloading of adjustments, and multiple `.hoardrc` files could be combined at will. There was a capability for recursive specification, so that the same adjustment could be applied to an entire directory tree.

The Coda Hoarding Formula

As mentioned above, CODA combined LRU information on all files with user-provided adjustments from one or more `.hoardrc` files. The formula in [Kistler 1993] gives p_f, the hoarding priority of file f, in terms of h_f, the user-specified hoard priority for that file, and a_f, the time (in references) since the file was last referenced. a_f is set to 0 at the most recent reference to the file and increases over time. Higher values of p_f indicate a higher probability that the file f will be hoarded. We prefer a slightly different characterization of the CODA formula, as follows:

$$p'_f = \min(a_f, (R-1)\beta) - h'_f$$

where R and β are global constants with values of 1000 and 16, respectively, and h'_f is related to h_f as:

$$h'_f = \frac{\alpha\beta}{1-\alpha} h_f$$

where $\alpha = 0.75$. The sense of p'_f is the inverse of CODA's p_f, so that smaller values indicate a higher probability of hoarding.

The CODA approach can be summarized as setting a system-global upper bound on the LRU age (in references) of the file, and then subtracting a user-provided and file-specific offset. The constants and limits

[1] In practice, some important directories, such as `/bin`, were effectively locked in place, and were not dynamically managed.

involved are chosen such that a file given the lowest priority (h_f near zero) will never displace one given the highest allowable value, because the maximum offset is somewhat greater than the age bound. In fact, this same relationship applies to any pair of files whose user-specified hoard priorities h_f differ by more than 250 (the allowable range of values is 0 to 1000).

For files whose hoard priorities differ by less than 250, the effect of the formula is that a higher-priority file will stay in the hoard somewhat longer, even if it has been referenced less recently than a lower-priority one. For example, if A has a higher priority than B, and the recent reference sequence included $\{A, B\}$, the algorithm would discard B from the hoard before A even though B has been more recently referenced.

However, the global bound on the LRU age (16000 references) causes this adaptive nature to take effect only during relatively short or inactive periods. The trace-driven simulations reported in Chapter 8 showed that most 24-hour periods involved fewer than 10000 references, but occasional extremely active periods could generate up to two orders of magnitude more than that. Thus, even in an inactive period, files that had not been referenced in the past few days would be more than 16000 references old, and the global bound would cause the fixed values from the `.hoardrc` file to dominate. The relatively low bound means that if (1) A has even a slightly higher hoard priority than B, (2) A has not been referenced in six months, and (3) B was referenced only last week, then A will still be hoarded in preference to B. If A has a priority of 1000 and B is set at 1, the choice of A over B is probably what the user would prefer. But if A is set at 250 and B at 249, it may be undesirable.

On the other hand, if there are three files A, B, and C, with relative priorities of 200, 250, and 300 respectively, and all last referenced long ago, the Coda scheme will correctly capture their relative importance. Similarly, if the same three files were referenced within 800 references of each other and given these priorities, Coda would tend to keep C in preference to A, which again probably reflects the user's intentions. (But if the files were separated by 4000 references, the user-specified priorities would dominate, regardless of reference order.)

Published Experience with Coda

The published results on Coda [Kistler and Satyanarayanan 1992, Kistler 1993, Satyanarayanan *et al.* 1993] have given relatively few quantitative results on the success of the hoarding methods. Most of the material in [Kistler 1993] is concerned with the implementation of disconnected operation, and all of the quantitative results in both [Kistler 1993] and [Satyanarayanan *et al.* 1993] involve measurements of the performance of the disconnection/reintegration process and of user working sets (which are not significantly affected by disconnection, though as discussed in Chapter 2 they have a large impact on the feasibility of hoarding). The earlier paper [Kistler and Satyanarayanan 1992] presents the material at a high conceptual level. These results are discussed further below.

Several possible metrics for evaluating a hoarding system are suggested in [Satyanarayanan *et al.* 1993]. All of these metrics are repeated and discussed in Section 7.1.2 of this dissertation (p. 76). However, the published results do not include evaluation of any of these measures.

Difficulty of Managing Hoard Profiles The hoard profiles (`.hoardrc` files) necessary to use Coda successfully were quite easy to understand, yet appear to have been difficult to manage. The problem is that modern applications make use of numerous files, many of which are stored in multiple directories. The user is often unaware of the complete list of files involved in running a particular application.[2]

The developers of Coda were aware of this problem; for example, [Kistler 1993, Section 5.3.1.4] discusses the challenges of generating cache "hints" for the benefit of a hoard manager:

> The cost of generating cache hints can be significant. Hint generation may involve scanning directories, gratuitously executing programs, even examining source code.

[2]For example, [Kistler 1993, p. 193] relates a situation in which the windowing system would hang when started in disconnected mode; this turned out to be because fonts were stored in a compressed format, requiring the `uncompress` program to use them. A reference-tracing program was required to discover this hidden relationship so that `uncompress` could be added to the list of files needed by the windowing system.

This level of effort may be beyond what one can expect of the average computer user; it may be better to automate such an operation using a facility similar to our external investigators, discussed in Section 5.4 (p. 56).

An indication of the complexity of the problem is given in [Satyanarayanan *et al.* 1993, Section 5.1.1]:

> Most users employ about 5-10 profiles at any one time. Typically, this includes one profile representing the 'personal' data: the contents of his or her root directory, notes and mail directories, etc. Several others cover the applications most commonly run by the user: the window system, editors and text formatters, compilers and development tools, and so forth. A third class of profile typically covers data sets: source code collections, publication and correspondence directories, collections of lecture notes, and so on. A user might keep a dozen or more profiles of this type, but only activate a few at a time (*i.e.*, submit only a subset of them to the local Venus). The number of entries in most profiles is about 5-30, with very few exceeding 50.

This implies that many users would set up their personal hoard profiles on a per-directory basis, without attempting to achieve a separation between important and unimportant files in those directories, possibly because of the difficulty of making this separation in an environment where the importance of any given file may be fluid. It also appears that the users maintained a constant background awareness of their hoard profiles, so that the proper one could be submitted at the right time to ensure work could could continue.

The paragraph immediately following that quoted above states, "Contrary to our expectations, there has been little direct sharing of hoard profiles. . . We expect the degree of direct profile sharing will increase as our user community grows, and as less sophisticated users begin to use CODA." However, increased sharing has not been subsequently reported, possibly because writing a successful hoard profile is a difficult, user-specific task that is accessible only to the expert.[3]

Success of Coda Hoarding Section 5.1.2 of [Satyanarayanan *et al.* 1993] discusses experience with hoard misses:

> Many disconnected sessions experienced by our users, including many sections of extended duration, involved no cache misses whatsoever.

This observation is attributed to two factors: "Hoarding has been a generally effective technique for our user population," and "most of our disconnections were of the voluntary variety, and users typically embarked on those sessions with well-formed notions of the tasks they wanted to work on. . . [T]hey did not normally disconnect with the thought of choosing among dozens of distinct tasks." The second point deserves emphasis: CODA was a success partially because users tailored their behavior to it, and because they limited their activity. This is a phenomenon common in computing. If a system responds in an undesirable fashion, users quickly adapt their activities to avoid the problem.

Although this might seem to be a flaw in CODA, we believe that it is not, and instead highlights a factor that eases the design of many types of systems: users will adjust their behavior to make the system perform better. In this case, mobility places certain restraints on the user, and just as a traveler is careful to put airplane tickets in his briefcase before departure, he will tend to "tickle" his mobile computer in a fashion that increases the probability of its being useful while disconnected. This observation applies to all hoarding systems, including SEER, although one of the advantages of SEER is that it requires less user adaptation of this sort.

The same section of [Satyanarayanan *et al.* 1993] also states:

> When disconnected misses did occur, they often were not fatal to the session. In most such cases the user was able to switch to another task for which the required objects were cached. Indeed, it was often possible for a user to "fall-back" on different tasks two or three times before they gave up and terminated the session.

[3]Brian Noble [Noble 1997] has indicated that direct sharing has recently increased, partly due to use of a relatively unfamiliar software package for which a single expert has created a standardized hoard profile..

Although the adjectives in the above two passages are not quantified, there is an implication that a nontrivial fraction of disconnections experienced misses, and that when misses occurred, they sometimes forced the user to switch to a new project, or even to cease work altogether. The discussion of falling back on a different task "two or three times" suggests a relatively high failure rate.

CODA was the first system to make disconnected operation a normal usage mode. Although CODA's hoarding system required a high level of user involvement, the system made major contributions by implementing the first client-server optimistic-replication system, and by demonstrating that disconnected operation could work at all.

9.2.2 Face, Little Work, and Laputa

Besides CODA, disconnected operation has also been supported in FACE [Alonso *et al.* 1990] and LITTLE WORK [Honeyman *et al.* 1992, Huston and Honeyman 1993]. However, the developers of these systems have concentrated primarily on the problem of optimistic file replication, and only secondarily on hoard loading.

LITTLE WORK uses a simple LRU scheme for choosing the files to be cached. Before a user disconnects, he or she is expected [Huston and Honeyman 1993]

> ...only to use the laptop on a network for a LITTLE WHILE... If she performs work similar to what she intends to do on the road, the cache will contain all the files necessary to support her needs.

Although this statement minimizes the inconvienence of the approach, the method is surprisingly workable, if only because users of such a system will learn how to "tickle" the applications they use so that they will access all necessary files. Of course, the burden is placed even more heavily on the user than in CODA, where it is at least true that once a `.hoardrc` file has been created for a particular project, the user rarely needs to worry about the project contents again. By contrast, every time a LITTLE WORK user disconnects, she must perform the proper sequence of actions over again, without error. If the work procedure involves more than a few steps, there is a high probability that one will be overlooked, especially if the user is in a hurry, with attendant disastrous consequences for the ability to work disconnected. Like all LRU-based systems, LITTLE WORK will also perform badly when attention shifts occur.

The designers of FACE suggested a hybrid approach in which the LRU concept would be combined with a `.stashrc` control file to choose files that belonged in the hoard. They also suggested automatically interpreting a user's `makefile`s, although it is not clear whether this idea was ever implemented. This work predates CODA, but it apparently was not carried to fruition, and no usage data of any sort has been reported, so it has had less impact.

Finally, a planned system named LAPUTA was described in [Skopp and Kaiser 1993], though no further results have been published to date. LAPUTA's hoarding system was not described in great detail, but appeared to incorporate augmented LRU in the manner of CODA, plus a rule-based system that used detailed knowledge of the user's work process, possibly generated by a human expert.

9.2.3 Spy Utility

To date, the most ambitious hoard-filling project is the SPY UTILITY built by Tait *et al.* to run on the OS/2® operating system [Tait *et al.* 1995], and now marketed by IBM® as part of their MOBILE FILE SYNC product. Unlike CODA, FACE, and LITTLE WORK, Tait attempted to detect an application's working set automatically, building on his previous work in file prefetching [Tait and Duchamp 1991].

The primary innovation of SPY UTILITY was the observation of process trees to generate projects for hoarding. For each executable program in the system, a record was kept of the identity and ordering of the files accessed by that program. Subprocesses and their accesses were also tracked, producing a pattern tree that characterized the behavior of a particular application, including child processes. Trees similar to each other were collapsed to save space and help recognize common behaviors. Like SEER, access histories were kept permanently, so that attention shifts could be handled gracefully. This latter feature was in sharp contrast to Coda and Little Work, which discarded any dynamic information beyond the LRU age of files currently in the hoard.

At hoard-filling time, the application and its related files were grouped and presented to the user as possible hoarding selections. For each program, the user could choose either to hoard only the most recent working set (*i.e.*, to load the same files that would be selected by an LRU algorithm), to hoard all working sets ever observed for a given application, or to omit the application from the hoard entirely. Other options allowed the user more low-level control of the hoard contents.

Spy Utility also introduced "generalized bookends," which were a convenient way for the user to inform the system of the beginning and end of a user-defined activity—for example, an edit-compile-debug cycle. This automation of Coda's `spy` program [Kistler 1993] was an improvement on that system's requirement for expertise, though it still placed more burden on the user than one would like.

Spy Utility was a major step forward, but was still less ambitious than Seer. Some of its flaws were merely details of implementation: for example, it asked for more user interaction that may have been strictly necessary, and it was somewhat inflexible in asking the user to choose between hoarding the most recently observed program tree or hoarding all trees ever observed (rather than the intermediate option of selecting an arbitrary subset). Even more than Seer, Spy Utility was also hampered by having to work on top of an uncooperative operating system, and its designers were forced to address many questions similar in flavor to those discussed in Chapter 6.

More seriously, Spy Utility suffered from its top-down approach. Projects were defined in terms of the top-level programs that invoked them, rather than by the cluster of data files actually accessed by the user, and there appears to have been no way of detecting relationships between the top-level programs themselves.[4] Thus, for example, Spy Utility was incapable of noticing that the standard edit-compile-debug cycle was a single project involving three separate utility programs, each spawned from a single shell. Nor could it recognize relationships between two files accessed by two related subprocesses, except through the existence of those subprocesses themselves. Attention shifts were recognized only when a top-level executable file was activated [Tait 1997]. On the other hand, the bookend facility allowed users to provide hints that could go a long way towards alleviating this difficulty.

[4] Tait has commented [Tait 1997] that "this was probably the biggest motivation for bookends."

Chapter 10

Future Work

During any project of this magnitude, a number of ideas and suggestions arise that are not practical to implement within the available time frame. In this chapter, we discuss the most important of these extensions to the research.

10.1 Improvements to Seer

The first category of future work involves simple improvements to the existing design and implementation of SEER. These improvements fall in turn into several sub-categories.

10.1.1 Improving Project Detection

Although SEER performs very well (see Chapter 8), there is a number of ways in which its detection of projects could be further enhanced. This section discusses some of the possible modifications that could help with project detection.

Extended-Activity Files. Some files remain open for long periods of time; log files and long-running processes are two examples of this behavior. Such files will have a very small lifetime reference distance (Definition 3.2.1) to a large number of other files. It has been suggested that such files should be treated like often-referenced files (Section 6.2, p. 65), being excluded from semantic-distance calculations and instead hoarded automatically.

Popular Relatives. Some files are related to by large numbers of other files, in that many other files list them in their near-neighbor list. Such "popular relatives" are a problem because they can cause excessive clustering. The frequent-file detection discussed in Section 6.2 helps with this problem, but does not necessarily eliminate it. Another approach would be for every file to keep a "number pointing at me" counter, and to refuse to participate in clustering if this counter exceeded some threshold. We plan to investigate the severity of this problem and the effectiveness of such a solution.

This problem is very similar to the problem of noise words in information retrieval, and techniques from that field may be applicable to our situation.

Other Approaches to Frequent Files. Currently, SEER detects frequently referenced files by comparing their reference counts to a threshold (Section 6.2, p. 65). An alternative method would be to base this decision on inter-file relationships. If a file is related to (or related to by) a large number of other files, it is not useful for clustering, so it would be better to just ignore relationships involving this file. This modification is closely related to the previous idea of "popular relatives."

Extended Name Investigation. Our current name investigator detects only very simple patterns. It might be useful to improve this detection; for example, a sequence of the form xaa, xab, ... probably represents related files.

Process Trees for Project Detection. Besides clustering based on semantic distance, there are a number of other approaches to detecting project membership. For example, Tait [Tait *et al.* 1995] defines a project as every file accessed by a process tree whose immediate parent is an interactive shell. (The shell programs are listed in a control file.) It would be instructive to implement this algorithm and compare it to SEER using the miss-free hoard-size metric.

Clustering with Process Trees. Process trees could also be used to drive the clustering algorithm more directly. Clusters would be based on the file accesses observed directly below some high-level process, usually one spawned by a shell. Special heuristics would be needed to handle processes that perform multiple functions, such as compilers. However, it might be possible to handle these programs as special cases.

Another option would be to use clusters developed from process trees as "seed" clusters, and then expand them using semantic-distance information.

Task Boundaries. Another approach to project detection would be to automate Tait's "generalized bookends." If it were possible to detect the boundary between two tasks, identifying the task contents would become trivial. One way to find boundaries without user interaction might be to detect a reference to a file that has been unused for a relatively long time.

Categorizing Files by Hand. A similar idea, first suggested in [Floyd and Ellis 1989], is to hand-categorize users and their files into gross major categories, and then to take advantage of these categories to apply more specialized algorithms for project detection. For example, Floyd proposes that users be separated into system daemons, "plain" users, and a few others; files are categorized into temporary, permanent, and log files. Our current design supports the idea of temporary files (see Section 6.6, p. 67), but it might benefit from also recognizing log files. The user categorization could be extended to support major behavior categories, such as software development, project management, and so forth (see Chapter 2). The clustering algorithm might then use these categories to advantage.

Automatically Categorizing Files. The present design of SEER considers all files as "equal" in terms of clustering. An alternative would be to automatically categorize files according to their function in the system. *Operators* would be programs (*e.g.* `make`) that created or operated upon other files. These files might be cluster members, but would not cause clustering and would not change the cluster priority.[1] *Objects* would be files that were operated upon; they would participate actively in clustering.[2] Finally, *regenerable* files would be those that could easily be recreated upon demand, such as program object files. Regenerable files might participate in clustering, but with less impact than objects.

It is an open question how these various categories might be detected. Clearly, operators must be executable files, but some executables might be objects or regenerable files. In general, operator files would only be opened for read or execution, while objects would be occasionally opened for write, but this approach ignores the case of database files such as personal spelling dictionaries, which might be better considered objects than operators.

Hand specification, as discussed above, is a plausible approach, but one that is at odds with the goals of SEER. Another approach would be to hand-classify files into major types, as suggested above; however, real systems have large numbers of types,[3] so that hand classification may be impractical.

[1] Jerry Popek has observed [Popek 1996] that utility programs are the "ands" and "ofs" of clustering.

[2] Some files, such as programs in development, might be both operators and objects.

[3] For example, consider the number of types supported by the Apple MACINTOSH®, or the number of lines in `/etc/magic` on UNIX systems.

A final difficulty with this approach is that the operator/object dichotomy inherently assumes that all objects should be hoarded together with their associated operator(s). This assumption is not always valid; for example, PowerPoint is a sufficiently large package that the user might prefer to hoard only the subset that is actually being used for the current presentation.

Improved Categorization of Files. There are many clues to file relationships that Seer does not currently use. For example, executable programs tend to operate on numerous target files, while target files tend to be closely associated with a particular application (this observation is especially true in an environment such as Windows, where files often have an application-specific format). Furthermore, public executables (*e.g.*, those in `/usr/bin`) tend to be associated with more files than those in the user's working directories, which are more likely to be specialized utilities. There is a rich collection of information that could be taken advantage of, for example in the calculation of overlapping clusters.

This information could also be used in the hoarding decision. For example, a reference to an editor should not cause the hoard to be filled with every edited file, but a reference to a text file should also bring in the editor—and using the editor should ensure that any necessary fonts are available. However, further research will be needed to develop algorithms for implementing these concepts.

Analyzing Variance in Semantic Distance. Tom Kroeger [Kroeger 1996] has suggested that the correlator could dynamically monitor the variance of semantic-distance values, and then use the variance to estimate the reliability of the measurement. Semantic distances with low coefficients of variation would be given more weight in the clustering algorithm.

Weighting Distance in Process Trees. In the current implementation, a process inherits its parent's file-reference history when it is created, and the history is merged into the parent's history at termination time, allowing cross-process file relationships to be detected. However, it is not necessary to apply a unit weight to the merged distances. A non-unit weight could be used to give less importance to relationships that were the result of deep process trees. Thus, for example, the subprocesses spawned by a complex `make`, itself initiated from `emacs`, would not develop a spurious relationship with an unrelated file that had been edited just before the `make` began. (On the other hand, since process execution is treated as a file reference, a natural distance will tend to be introduced in this situation anyway.)

Another option would be to modify hoard priority according to where a program appears in the process tree. For example, command shells, which appear very high in the tree, are more important to the user than deeper programs, and thus should receive a boost in hoard priority. (However, those shells are also constantly in use, so perhaps an artificial boost is not necessary in this particular case.)

Alternate Measures of Meaninglessness. The inference of meaningless activity based on watching directory opens and counting potential file accesses, discussed in Section 6.1, p. 63, could be modified in various ways. For example, it might be useful to consider creates by meaningless processes to be themselves meaningful, since files that the user is writing are almost always of interest to him. Another option would be to ignore only `PEEK` references from such processes, on the assumption that most meaningless processes merely look at file attributes, rather than opening the files. Exceptions such as `tar` would need to be listed in the instruction file (Section B.2.4, p. 143). We plan to investigate the effects of these alternatives in the future.

Process-Specific Distance Summaries. Although Seer calculates semantic distance separately for each process, the value that is stored in the file table is a summary of the distances for all processes. It might be interesting to maintain a separate summary for each invoking program. Summarizing by application would keep large distances generated by one program (*e.g.*, an editor) from "polluting" smaller values produced by another (*e.g.*, a compiler). This approach is impractical with the current data structures, but would be a fruitful area for further research.

Automated Detection of Critical Files. The current approach to dealing with critical startup files (Section 6.3, p. 65) is unsatisfyingly *ad hoc*. We would prefer to invent an automated method, and plan to investigate various options in our further research.

Other Relationship Hints. There are a few UNIX system calls that are not currently traced, but could be. For example, `chmod` and `chown` generate legitimate file references. The lack of tracing for these calls has not been a problem because they tend to occur in conjunction with other activity, but we plan to add them to the trace list in the future. More interesting is the `pipe` call, which indicates an unusually tight relationship between the processes at the ends of the pipe. SEER currently detects this relationship as a side effect of other activity (primarily the parent/child relationship), but it might be useful to treat `pipe` as a special case and use it to infer a tighter binding than is currently detected.

Improved Rename Handling. As discussed in Section 6.9, p. 68, important file relationships can be lost in certain renaming situations, such as backup-by-rename. It would be desirable to preserve the source file's relationships immediately after a rename, in case it was immediately recreated, discarding it later through a garbage-collection algorithm similar to that used by the deletion manager.

10.1.2 Alternative Designs for Prediction

SEER predicts hoard needs through a combination of clustering based on semantic distance, and LRU analysis of the clusters. There are other ways to predict; this section summarizes some of the more promising approaches.

Text Compression. Several authors have suggested using text-compression techniques to drive predictive caching algorithms [Curewitz *et al.* 1993, Kroeger and Long 1996, Vitter and Krishnan 1996]. Similar techniques could be adapted to generating clusters: if the text-prediction table indicated that the reference sequence $\{A, B, C\}$ is followed by D more than a certain percentage of the time, then it would make sense to cluster D with the preceding three files.

Incremental Clustering. When SEER develops its clusters, it always starts from scratch. This design is wasteful, since most of the clusters do not change quickly, and the slowness of clustering suggests that significant performance benefits could be achieved if old information could be reused. An incremental clustering algorithm would start with the previous clustering and try to improve it based on recent the changes in file relationships. Iterative clustering algorithms are amenable to this approach. We would like to investigate adapting one of these algorithms to SEER's needs, in hopes of improving the performance at clustering time.

Hierarchical Clustering. Clustering algorithms are generally divided into two classes: hierarchical and non-hierarchical [Duran and Odell 1974, Hartigan 1975, Späth 1980]. The former class naturally produces a tree of "nested" clusters, generally rooted by a single cluster containing all objects. This structure is attractive for a hoarding system, since one could start at the bottom of hierarchy and hoard files while climbing higher, until the hoard was full. This design would tend to fill the hoard with the maximal set of files related to the important projects, possibly decreasing the chance of a work-stopping hoard miss. Of course, such an algorithm would still have to tolerate overlapping clusters, which is a requirement that is at odds with the characteristics of most existing algorithms. We plan to investigate hierarchical algorithms in the future.

Cluster Merging. The agglomerative clustering algorithm is driven only by the distance between clusters. It would be interesting to investigate an algorithm that merged several very similar clusters into a single slightly larger one. For example, if two clusters each contained over 15 members and over 90% of the members were held in common, it might be beneficial to combine them into a single larger cluster.

However, one must be wary of pathological situations. For example, many compilation projects have large numbers of include files in common, but it would be unwise to combine these common files into larger clusters because of this sharing.

Decaying Relationships. The World Wide Web has the characteristic that the transitive closure approaches an all-ones matrix, because many high-level Web pages contain pointers to other major pages that in turn lead indirectly to most of the Web. For example, the Yahoo directory service attempts to catalog the Web comprehensively, so any page that contains a link to Yahoo will indirectly point at millions of other pages. A clustering method that simply looked at link relationships would probably generate a single cluster containing the entire Web. An alternative would be to use a method that decayed in some manner as it transited links, so that no page would cluster with one that was more than a few links away unless there were also a more direct path to that page. It is possible that this decay could be implemented as an aspect of the distance measure calculated between Web pages.

Adaptive Distance Heuristics. When a hoard miss occurs, there is currently no way for the user to give feedback that might improve SEER's internal algorithms. A more adaptive system might display an access history to the user, allowing him to indicate a point where an inference about relationships might have been made, but wasn't. This feedback could then be used to modify the parameters of the semantic-distance or clustering algorithms so that they would perform better in the future.

Considering Communication Costs. Currently, SEER assumes that communication is cheap when changing hoard contents. Ashvin Goel [Goel 1996] has noted that if the hoard is being updated over a slow link, file size might be an important factor in the equation. It might be better to risk a hoard miss on a medium-priority file rather than spending a long time downloading it, if skipping it would make it possible to use the time loading a larger number of other files, even if they were of lower priority. This additional consideration would be relatively easy to implement by simply adding file size as a weighting factor when files were being sorted by priority, prior to hoarding (Section 5.3.4, p. 55). Ideally, the weight would based on the cost of moving the entire cluster, which would be affected both by the sizes of other members and by which members were already present in the local hoard.

Cluster Temperatures. Some other researchers [Salem *et al.* 1992, Staelin and Garcia-Molina 1990] have suggested using a "temperature" to characterize the activity or popularity of a cache object. The idea is that the temperature rises higher (usually in a linear fashion) whenever an object is accessed, and decays over time, usually exponentially. An object that is used frequently will gain a high temperature, which will then take a significant time to decay. (LRU is a temperature algorithm in which the temperature rises to a ceiling on each access, and decays linearly.) If a cluster temperature (without a ceiling) were used to drive the hoard-filling algorithm, the user might perceive better behavior than the current LRU scheme, because a high-temperature cluster would remain in the hoard even if the user worked on different files for a few days.

Other Methods of Incorporating Investigators. When clustering, SEER uses Manhattan distance to combine information from external investigators with the shared-neighbor count from semantic-distance measurements. We would like to investigate other methods of combining measures, including using modified Euclidean methods and doing the combination at a level other than the shared-neighbor count.

10.1.3 Implementation Improvements

As with any large software project, the final implementation of SEER is not ideal in all respects. This section discusses major candidates for improvements in the code.

Pathname Rewriting. We have mentioned (in Section 5.2, p. 52) that the `correlator` can accept trace information from multiple `observers`. This feature is designed to support users who may prefer to use a larger desktop machine when it is available. In many installations, different machines may use different pathnames to access the same files. For example, the user's home directory might be stored under `/h/users` on one machine and under `/home` on another. In such situations, it would be desirable for the `correlator` to be able to rewrite pathnames on a per-machine basis, so that a reference to `/h/users/geoff/foo` on the desktop would cause `/home/geoff/foo` to be hoarded on the laptop.

A related issue is the question of multiple hardware architectures. Most versions of UNIX require executable files to contain machine code for the executing host.[4] If the user executes a program on a SPARC® desktop, it would be useless for SEER to hoard the binary on a PENTIUM-based portable. Instead, the corresponding binary for that architecture should be selected. On many networks, the correct choice can be achieved through pathname rewriting (for example, replacing the string "`/sparc/`" with "`/x86/`").

External Predictors. Although SEER makes very successful predictions based on generalized inferences, it is conceivable that a specialized predictor might be able to do better in certain circumstances. It would be useful to have an interface, similar to that used for external investigators (see Section 5.4.1, p. 56), that would allow an external prediction mechanism to feed information to the `correlator` for use in hoarding decisions. Such an interface would need to include priority information in a form that could be integrated with the internal priorities used to fill the hoard.

An extension of this idea would be to create an external hoard manager. Currently, hoard-management decisions are made internally in the `correlator`, and then fed to the replication substrate. It might be more flexible to have the `correlator` feed a prioritized list of files to an external hoard manager, which could then integrate other information in making the final hoarding decision. The drawback to such an approach is that the current design makes hoarding decisions based on cluster membership, rather than a simple per-file priority, so that an external manager would also have to be aware of these clusters. The extra communication overhead and complexity required for such an approach might overshadow the benefits of flexibility, so that a more efficient alternative design might be preferable.

Investigation Requests. The current design for external investigators (see Section 5.4.1, p. 56) assumes that the investigators are invoked by an unspecified mechanism outside the purview of SEER, and that they spontaneously feed information to the `correlator`. An alternative design would support a more automated and interactive approach, in which the `correlator` could select a group of files that were of particular interest (perhaps because they were near the "boundary" of a hoarding decision), and ask that various investigators be run on them. This capability would allow the investigators to respond to the dynamic needs of the system, rather than gratuitously providing information that might never be of practical value.

Trace Annotation. For development evaluation purposes, it would be useful to know the user's opinion regarding the beginning and end points of various tasks. A utility similar to that described in [Tait *et al.* 1995] would make it possible to record this information for later analysis, which could be invaluable in comparing various parameter settings and clustering algorithms.

Maintaining Constant Free Space. To add control for research purposes, we implemented SEER so that the total hoard size is fixed. For a production system, it would make more sense to attempt to maintain a certain amount of free disk space, and let the hoard size vary as a function of the free space. If free space became low while the machine was disconnected, SEER could ask the replication system to evict the files least likely to be needed. This decision would be made based on the most recently available information, rather than being restricted to the hoarding priorities calculated before disconnection.

On-Disk Data Structure. To simplify the research implementation, we chose to store all of SEER's data structures in the `correlator`'s main memory, even though we knew that the database would be very large. SEER would have significantly less impact on small machines if the database were kept mostly

[4]Two notable exceptions are systems that provide automatic emulation of differing architectures, and the "fat" binaries supported under NeXTStep™.

on disk, with only current files kept in main memory. Moving the information to disk is primarily an engineering problem, albeit a nontrivial one. One helpful characteristic is that most files are not actively referenced at any given time, and the database only needs to be accessed and updated when a file is referenced. If an incremental clustering algorithm were used, the on-disk portion of the database could also be excluded from the clustering process. This observation leads us to believe that an on-disk solution could be both practical and efficient.

File Compression. At its most fundamental level, SEER is a disk-space manager. One of the most popular ways of managing disk space is to compress files. This fact leads to the idea, first suggested by An-I Wang, that SEER could initiate file compression and decompression rather then moving complete files between the hoard and some other replica. Ideally, such a facility would be integrated with the replication system so that compression was an intermediate state between being stored and unstored.

Remote Correlation. The current `correlator` runs on the portable computer, which is wasteful of precious resources. It would be better to have a different split of responsibilities, performing only minimal activity on the portable, and moving the rest of the computation to a fixed machine on the host network. However, since this approach would require extensive logging on the laptop and replay of the logs at reconnection time, it is possible that the cost of logging and replay would exceed the cost of running the `correlator` locally.[5] We plan to investigate this tradeoff in detail in the future.

10.2 Further Studies

Although we have done extensive studies that show SEER works well in a real setting, there is ample opportunity for further investigation of the interaction of various factors.

Parameter Settings. The current values of internal parameters were chosen using relatively *ad hoc* methods (see Section 6.12, p. 71). Now that we have developed the miss-free-hoard-size measure (Section 7.1.3, p. 78), it is possible to do a much better job of evaluating the effects of various parameters. We plan to do extensive simulations to optimize the parameters in the future. At the same time, we will investigate the sensitivity of the various hoarding methods to deviations of the parameters from the optimum.

Effect of Investigators. As discussed in Chapter 8, the external-investigator feature has proven disappointing in that it does not seem to have the expected effect on system performance. Since the weights assigned to external investigation are also system parameters, the same parameter-finding techniques discussed above can be used to optimize these weights. We hope that improved weight settings will make a large difference in the utility of external investigators.

Effect of Process Trees. Although we have adopted Tait's paradigm of tracking the UNIX process tree and separating references according to process (Section 6.8, p. 68), we have not yet compared SEER's performance with and without this feature. SEER has a run-time option that allows this feature to be disabled, so we plan to run further simulations to quantify its effect.

Cluster Stability. Currently, SEER rebuilds its clusters from scratch every time it needs to recalculate the hoard. We have hypothesized above in the discussion of an on-disk data structure (Section 10.1.3) that advantages could be gained from using an incremental clustering algorithm. However, an incremental approach would only be useful if cluster membership tends to be stable. We plan to investigate the stability of cluster membership in future studies before experimenting with incremental algorithms.

[5] Some replication systems, such as CODA and LITTLE WORK, already keep such logs for their own purposes. Remote correlation might be especially attractive on these systems.

Other Operating Systems. Although some of the issues addressed in Chapter 6 are applicable to all operating systems, others are UNIX-specific. A port to a dissimilar system such as WINDOWS 95 would reveal many of these differences, and would highlight important questions for operating system designers.

Other User Communities. A port to WINDOWS 95 would also allow SEER to be deployed in other application environments. Based on the studies reported in Chapter 2, we have strong reason to believe that SEER would work well in these environments, and we plan to perform studies to quantify this belief.

10.3 Other Applications of Predictive Hoarding

The ideas behind predictive hoarding can be applied to other arenas. This section discusses some of the more interesting applications of the concept of prediction.

Directory Reorganization. Leonard Kleinrock has suggested [Kleinrock 1994] that SEER could be used as an aid to directory reorganization. When a user had an overly large directory, or felt that his files were scattered among unrelated directories, the clustering information developed by SEER could be displayed to the user or fed into an automated tool that would help the user find a more logical organization. (Of course, the user would probably also need an auxiliary tool to help him find his files in their new locations!)

Disk Layout. Recent studies [Akyürek and Salem 1995] have suggested that dramatic performance improvements can be achieved by rearranging blocks on the disk in response to access patterns. Current work has concentrated on "hot" blocks without considering their interrelationships, but it is likely that even more benefit could be gained by using using SEER's high-level semantic awareness to drive a disk reorganizer. The clusters generated by SEER could be placed physically close to one another on the disk, potentially improving I/O performance. With appropriate changes to the on-disk data structures, the I/O system could also take dynamic advantage of SEER's clusters to further enhance performance.

Access Control. In an environment with strict access controls, an approach similar to SEER might be able to contribute to access management by detecting users who need access to certain groups of files (projects).[6] It could also help with security auditing by detecting unusual inter-project relationships.

Higher-Level Adaptation. A side effect of managing a portable computer's hoard is that SEER is aware of what applications a user is likely to (or indeed is able to) run. This information could be provided to higher-level services so that they could optimize their behavior. For example, if the hoard contained a large number of communication-dependent programs, a network manager might choose to maintain more routing information so that a connection could be established more quickly.

Other Prediction Environments. The techniques pioneered by SEER may be applicable to a large number of other environments that might benefit from prediction. Some of these applications include Web browsing, disk access, network routing [Krishna *et al.* 1997], ftp caching, process scheduling, and network capacity planning. Outside computers, the ideas pioneered by SEER could be applied to fields such as logistics or warehousing.

Many of these applications pose significant research problems of their own. For example, a Web predictor should be able to draw inferences even in the face of the daily link-name changes used by a number of information providers.

[6]This idea was first suggested by Mike Beresford [Beresford 1995].

10.4 Ancillary Research

The development of SEER has raised several side issues that, while not directly relevant, are nevertheless interesting in a larger context.

Hoard Residence Lifetimes. Ashvin Goel has suggested [Goel 1996] that it would be interesting to track statistics on how long files spend in and out of the hoard. There are actually two distributions of interest: the time spent in the hoard, and the time spent as a member of the daily or weekly working set. The shapes of these distributions might provide insight into both user behavior and the effectiveness of various hoarding schemes.

Algorithms for Overlapping Clusters. Most past clustering research has concentrated on algorithms for finding distinct clusters. We believe that there are many applications for the overlapping clusters employed by SEER. For example, clustering algorithms are sometimes used for biological classification, but recent research has suggested that it is not always beneficial to separate species according to strict boundaries. In the future, we plan further research to develop and characterize new algorithms that can support the concept of overlapping clusters.

Chapter 11

Conclusion

Our research has encompassed a number of significant subproblems and produced a complex and successful system for predictive hoarding. In this chapter, we will review the highlights and important contributions of the project.

11.1 Summary of the Problem

The need for predictive hoarding arises because of the lack of adequate network connectivity for mobile computers. Users who operate disconnected must be assured that they will have access to all necessary files while they are away from the network. There are a number of approaches to addressing this problem, including limiting the user's working set to what will fit on a portable machine's disk, or requiring the user to hand-specify the chosen files in some fashion.

A third approach is to automate the file selection, dynamically predicting the set of files that the user will need while disconnected, and hoarding only those files on the laptop. SEER is a system designed to accomplish this automation.

Two of the biggest difficulties facing any automated hoarding system are hoard misses and attention shifts. Hoard misses, in which a required file is found to be unavailable during disconnection, are very damaging to the user's ability to get work done. Often, the lack of a single file will be sufficient to stop work on a particular task, and the user must respond by shifting to a less important task or even shutting the computer down and stopping work. Even when work can continue on the main task, the character of the work must usually change because of the lack of the desired file.

Attention shifts occur when a user switches to a new task, one that has not recently been active. This causes problems because the hoarding system must respond quickly and accurately to this behavior change by hoarding the files necessary to accomplish the new task.[1]

11.2 Seer's Approach to Hoarding

SEER addresses the problems of hoarding by considering the user's files in terms of a set of projects, each of which is treated as a unit. By grouping together all files needed to work on a particular task, SEER is able to take an all-or-nothing attitude towards hoarding: if a task is important to the user, he can be confident that 100% of the files needed to accomplish that task are present when he disconnects. This greatly reduces the probability of a hoard miss, because if a task is currently active, it will be hoarded in its entirety. A hoard

[1] Since hoard changes can only be accomplished when the computer is connected, the system cannot respond well to attention shifts that occur during disconnection. In this case, the system should recognize the desire to accomplish new work, and hoard the appropriate files at the next opportunity. However, our data indicates that most attention shifts begin while the computer is still connected, so that the system can often recognize a shift and prepare for it prior to disconnection.

miss can only occur if a task has been inactive and suddenly becomes of interest, which is the definition of an attention shift.

No hoarding system can deal with attention shifts that occur while disconnected, but SEER's project-based design lessens the impact of while-connected attention shifts. As soon as the user begins work on a long-ignored project, SEER will recognize that the project is again active and arrange to hoard it. It is not necessary for the user to explicitly notify the system of the new activity, nor is it necessary for him to access every file that is a member of the project. The user's normal work patterns will serve to ensure that the new project is hoarded.

SEER discovers projects by observing the user's actual behavior. The file reference stream is monitored both to infer relationships among files, and to learn which files are currently in use. The inter-file relationships are then used to build clusters that represent projects. Finally, the projects represented by active files are chosen for hoarding, and a separate replication mechanism arranges for these projects to be present on the mobile computer.

11.3 Contributions to the Hoarding Problem

The SEER research has addressed a number of significant subproblems as part of building a predictive hoarding system. These include:

Analysis of user behavior. To determine whether predictive hoarding was feasible, we conducted an extensive long-term study of user behavior in the real world. We were fortunate to be allowed to investigate three types of users in a business environment, rather than being limited to observing a small software research group. This work is reported in Chapter 2.

Semantic distance. To allow SEER to infer file relationships, we have invented a new measure, called semantic distance, proved fundamental theorems to establish complexity bounds, and developed efficient estimation algorithms for the measure. Semantic distance and the associated algorithms are described in Chapter 3.

Fast clustering algorithm. SEER's project-based approach uses semantic distance as input to a clustering algorithm. Because of the amount of data involved and the requirement for overlapping clusters discussed in Section 6.4 (p. 66), we developed a new algorithm to generate the necessary clusters. The algorithm and its background are discussed in Chapter 4.

Predictive hoarding system. Building on the foundations of semantic distance and clustering, we designed and implemented a working portable predictive hoarding system. This system is described in Chapters 5 and 6, and Appendix B.

Measurement methodology. Previous studies of predictive hoarding have been hampered by the lack of a standardized metric that can be used to summarize and compare the performance of different systems. We have invented a number of methods for characterizing hoarding systems, including several that allow differing approaches to be compared in a controlled, scientific, and meaningful manner. These measures are presented in Chapter 7

Successful results. Using our new metrics, we have analyzed the behavior of our own hoarding system, finding that it performs even better than we had hoped. In simulations, SEER was able to operate miss-free with a hoard size only a few percent greater than the optimum, compared to an excess of as much as 1000% required by LRU-style approaches. In live deployment with real users, 96% of all disconnections were completely successful (in the sense of being free of misses), and there were no instances where a user was forced to completely stop work because of missing files. A detailed analysis of SEER's performance is given in Chapter 8.

11.4 Final Comments

Hoarding is a relatively new problem, having arisen only with the development of portable computers with nontrivial capabilities. Earlier systems have attempted to ease this problem in various ways, but all have placed some level of the burden on the user, and none have reported quantitative measures of success. SEER extends this prior art by removing all requirements for user input and providing a numerical evaluation of the effectiveness of the system.

We believe that the concept of prediction has applications far beyond mobile hoarding. Simple modifications to the SEER concept can be applied to Web caching and disk performance, and further extensions raise the possibility of insights in areas as diverse as multiprocessor schedulers and user interfaces. SEER is a successful system, and may also be a pointer to a new direction for computer research.

Appendix A

A Simple Replication System

SEER is designed to be able to operate on top of nearly any file replication service. To demonstrate its portability and flexibility, and to simplify testing, a simple replication system called CHEAP RUMOR was developed. This appendix describes the major features of that system.

A.1 Features of Cheap Rumor

As discussed in Section 5.5, p. 58, SEER places only very minimal requirements on the underlying replication substrate. However, its usability will be greatly enhanced by the addition of a few extra features. CHEAP RUMOR was designed to provide all of the features that were both easy to implement and necessary for a reasonable level of user convenience. These features include all of SEER's basic requirements, plus the following additions:

- Multiple replicas (limited to a *master-slave* configuration, in which a master machine stores all files and a slave stores a selected subset)
- SEER-independent hoard changes
- Conflict detection
- Status queries
- Location queries
- Command batching
- Disconnected queries

In addition to implementing only limited capabilities, CHEAP RUMOR does not address some of the more subtle and complex issues handled by the FICUS [Guy 1991] and RUMOR [Gunter 1997, Reiher *et al.* 1996] systems. For example, RUMOR goes to great lengths to detect and properly process rare pathological cases involving files that have changed and had their modification times reset during a 1-second window while RUMOR is reading their contents. CHEAP RUMOR ignores situations of this sort, instead expecting that the user will avoid Byzantine behavior.

A.2 Design of Cheap Rumor

CHEAP RUMOR consists of two simple programs, `cheap_control` and `cheap_reconcile`, both written in the PERL [Wall and Schwartz 1991] scripting language. `Cheap_control` handles replication control (subset replication), status reporting, and the creation of new Cheap Rumor volumes, while `cheap_reconcile` handles update propagation and conflict detection.

A.2.1 Cheap Rumor Volume Organization

Files controlled by Cheap Rumor are organized into *volumes* consisting of selected subsets of directory trees. A Cheap Rumor volume comprises a root directory and all files within it, plus all files and directories contained in descendants of the root, but excluding any subtree that is itself a Cheap Rumor subvolume.

A Cheap Rumor volume is identified by the presence of a directory named `cheap_rumor_`x`_`y in the root of the volume, where x and y are the host names of the two machines storing copies of the volume.[1] This directory contains control information used by the replication system. Currently, two control files are used:

`parameters` stores volume parameters, such as the location of the matching replica on the cooperating (fixed or portable) machine.

`filelist` contains a database of all known files in the volume, including attribute information needed to control subset replication and support update and conflict detection.

A.2.2 The Cheap Rumor Database

The heart of Cheap Rumor is the file database, which is a Perl associative array indexed by the full pathname of the file on the local machine. For each known file, Cheap Rumor stores the following information:

- A file-existence flag, which indicates that the file exists on one or both machines,
- A replication flag, indicating whether the file is stored on both the master and slave machines, or only on the master,
- An *ignored* flag that indicates that the file is allowed to differ between the two replicas, and updates will not be propagated,
- The file type (directory, plain file, etc.),
- The file permissions and ownership,
- The size of the file in bytes,
- The last modification time of the file,
- A checksum of the file's contents, and
- The link target, if the file is a symbolic link.

This information is sufficient to detect new files, updates to existing files, and deletions. It cannot detect renames (which are seen as a deletion and a creation of a new file) and does not maintain "hard" links (which are seen as a second file containing the same data).

The "ignored" flag is of special interest, as to our knowledge Cheap Rumor is the first replication system to support it. We feel that this feature is critical to to the usability of a replication system that runs on dissimilar machines. For example, a user might wish to mark a configuration file as ignored so that the portable machine can be set up slightly differently than the fixed workstation.

[1]To ensure consistency in naming, x is always the lexically first name.

A.3 Cheap Rumor Algorithms

A.3.1 Terminology

As mentioned previously, CHEAP RUMOR supports a master-slave replication model. When an operation requires both machines to be involved, the machine that begins the operation is called the *initiating machine*. During a multi-machine operation, one machine is a *client* to the other's *server*. Usually the initiating machine is the client, but the roles are reversed during the second phase of reconciliation.

A.3.2 Functions of `cheap_control`

As mentioned above, `cheap_control` handles the following functions:

- Volume creation
- Subset replication
- Status reporting

All of these functions are very simple, but only status reporting can be done without access to the remote machine.

Volume Creation

Creation of a CHEAP RUMOR volume is very simple, because of the way unknown files are handled by `cheap_reconcile`. `cheap_control` merely creates the root directory of the volume (if necessary), the `cheap_rumor_`x`_`y directory, and the `parameters` file within it. These operations are carried out on both of the computers that store the volume. The all-important database is then created by `cheap_reconcile`, which is invoked automatically by `cheap_control` for this purpose.

Replication Changes

Changing the replication status of a file involves looking up the file in the CHEAP RUMOR database to compare the current status against that desired. If they differ and the file is currently replicated on both master and slave, `cheap_control` also compares the update status of the local and remote files against the database (using the server functions of `cheap_reconcile`, described in Section A.3.3, p. 129), and refuses to change the replication status if information would be lost. Assuming all of the tests have been passed, `cheap_control` then updates the database (both locally and remotely) and either deletes or creates the file on the slave machine, as appropriate, again using the reconciliation server to perform the operation remotely.

In addition, `cheap_control` runs the `controller` to notify the `correlator` of the change in replication status. The notification is done before the file is deleted or created, which is critical because SEER does not give any special treatment to `cheap_control`. (If the file were deleted without first informing SEER, SEER would remove the file from its internal tables and lose all clustering information and awareness of the file's very existence. By keeping SEER informed, the internal tables are modified to indicate that the file exists but is not stored locally, so that it can later be recovered when the user needs it.)

Status Reporting

Status reporting is the only CHEAP RUMOR function that can be done without contacting the remote replica. Information from the database is simply extracted and formatted for presentation to the user.

A.3.3 Functions of `cheap_reconcile`

`cheap_reconcile` handles the remaining functions of CHEAP RUMOR:

- Two-way update propagation
- Deletion propagation
- Conflict detection

All of these functions are handled simultaneously by comparing the saved database with the current status of both machines.

The Reconciliation Process

Reconciliation can be initiated on either the master or slave replica of a volume. Regardless of where the process is initiated, there are two similar phases, the first controlled by the initiating machine and the second by its partner. In each phase, the controlling machine operates as a client to a reconciliation server running on the other side. During the two phases, a new database is built based on the old database and the information discovered during the phase. At the end of reconciliation, the new database replaces the old one.

The first reconciliation phase searches the entire CHEAP RUMOR volume on the initiating machine for files and directories, starting from the root of the volume. If a particular directory (other than the root) contains its own `cheap_rumor_`x`_`y control directory, then it is the root of a sub-volume and is not examined further. Otherwise, the database is checked to see if the file is ignored. This check is done before descending into a directory, so that marking a directory as ignored will suppress CHEAP RUMOR's control of an entire subtree. If so, no further processing is done on the file.

In all other cases, three sets of file attributes are collected: one from the old database, one from the file on the client machine, and one from the file on the server. These attributes are then compared and reduced to six Boolean variables:

`oldExists` is true if the file was listed in the database,

`fullyReplicated` is true if `oldExists` is true and the file was marked as being replicated on both the master and slave machines,

`localSame` is true if the client file matches the recorded values in the database,

`remoteExists` is true if the file exists on the server machine,

`remoteSame` is true if the server file matches the recorded values in the database,

`remoteMatchesLocal` is true if the client and server files match each other.

The reader will note that there is no variable named `localExists`, because it would always be true.[2]

The meaning of the term *comparison of attributes* varies with the file type. It always includes the type itself. For everything except symbolic links the ownership and permissions are also compared; for symbolic links only the link targets are compared (after being adjusted to compensate for the different pathnames of the root of the volume on the two machines). In the case of ordinary files, the modification time, size, and (if necessary) checksum are also considered.

Reconciliation involves analyzing and resolving the 64 states of these six variables. Of course, some of the states are impossible; for example, if `remoteMatchesLocal` is true, then `localSame` and `remoteSame` must necessarily be equal.

Briefly, the important cases are as follows (the cases must be evaluated in the order listed). Boolean negation is indicated with a preceding exclamation point.

[2] The implementation actually includes this variable, but only for clarity and consistency.

`remoteMatchesLocal` Regardless of the previous state and whether the file previously existed, it is now fully replicated and is consistent across both copies. The current status of the file is entered into the new database.

Because this case is independent of the original state of the database, it handles both the common situation of an unchanged file and the case where a new file is copied to the other machine by the user. It also takes care of building the initial database when a new volume is created.

`!remoteExists` The file exists on the client but not the server. The precise action to take depends on the previous state in the database. If `oldExists` is true, then the file must have been deleted, so we delete it locally. Otherwise, a new file has appeared on this replica. If it is the slave replica, the file is copied to the master (because all files must appear on the master). Then the new information, including partial-replication status, is entered in the database.

`!localSame` or `!remoteSame` The file has been updated on one or both machines. If the update was on both computers, a conflict exists, which is reported to the user (who may optionally take immediate action to resolve the problem). If the update occurred on only one computer, it is propagated to the other by copying data and attributes.

In the second phase of reconciliation, the client and server switch roles.[3] The actions of each are the same, except that the new client does not need to re-examine any file that was already tested during the first phase. Thus, this phase will only discover files that do not exist on the initiating side, *i.e.*, files for which `remoteExists` will be false.

As an important side effect of the two-scan design and the fact that the new database is built from scratch, files that have been deleted from all replicas will simply disappear from the database with no special handling by `cheap_reconcile`. Although this behavior is normally correct, it does mean that if the user marks a file to be ignored by CHEAP RUMOR, deletes that file on both replicas, and later recreates it, CHEAP RUMOR will forget that the file should have been ignored. However, our experience has shown that this situation is rare and does not cause problems for users.

One other important detail concerns the handling of directory removal. UNIX does not allow a non-empty directory to be destroyed. CHEAP RUMOR takes most actions immediately, while it is scanning the local filesystem in a breadth-first fashion. Thus, the deletion of a directory is detected before its children are removed, which is problematical since a non-empty directory cannot be destroyed. To work around this restriction, most directory deletions are pushed onto a pending-destruction list, which is processed just before reconciliation terminates. To ensure that subdirectories are removed before their parents, the list is sorted and processed in reverse order of pathname length.

The `cheap_reconcile` Server

We have mentioned that the computer that initiates reconciliation starts up a server process on the remote machine. This server implements the following functions:

- Calculate and return the attributes or checksum for a given file;
- Store new attributes for a file in either the new or old database (the latter is used by the replication-control facilities of `cheap_control`);
- Remove information regarding a file from either the old or the new database;
- Read file data and transfer it to the client, or accept file data from the client and write it to the file;
- Create a directory, named pipe, or symbolic link;

[3] Since both phases involve recursive scans of the local disk, it is theoretically possible to do both scans simultaneously to reduce elapsed time. However, the current implementation does not take advantage of this inherent parallelism.

- Destroy a file or directory, optionally delaying the directory destruction until the end of reconciliation;
- Change the ownership, permissions, or modification time of a file or directory;
- Ask the user for guidance about an action to be taken (this function is used only during the second phase when the user is physically present at the server machine and has chosen to work interactively);
- Inform the SEER system on the slave machine of changes in replication status;
- Terminate the server and enter the second phase of reconciliation.

Appendix B

Detailed Design of Seer

Chapter 5 gives a general overview of the structure of SEER and of important considerations in its design. In this appendix, we discuss certain secondary details of the implementation that are nevertheless important to a full understanding of the system.

B.1 Details of `observer` Design

To make inferences from user behavior, one must be able to observe the user in action. Since the behavior of interest to SEER is embodied by system calls, our implementation must be able to observe or infer what those calls are. This information must then be collected and integrated to deduce file relationships for hoarding purposes.

B.1.1 Observing System Calls

As discussed in Section 5.2 (p. 52), a small kernel hook is used to collect information on system calls and feed them to an `observer` that tracks user behavior. In this section, we discuss the rationale behind this design and the details of our method.

Design Options

There are several ways to track the system calls made by a user application. There is also a subsidiary question of the amount of processing that is to be done at the time of collection. The choice of methods will have a significant effect on the efficiency and reliability of the system.

Options include:

1. Modify the application to report the necessary information.

2. Run the application under a tracing or debugging facility, interrupting it at appropriate times to collect system calls and their parameters [Alexandrov *et al.* 1997].

3. Implement a pseudo-NFS file system that traces I/O operations as a side effect of executing them [Duchamp 1997].

4. Replace the shared library containing the system-call routines with one that collects the information.

5. Modify the operating-system kernel to collect the information.

We rejected the first option, modifying applications, as being too difficult and inflexible. There is no single point in an application where the necessary information can be collected, so changes would have to be sprinkled throughout the application code. Worse, this approach requires access to the source code, so that it would not work with third-party applications.

The second option, using a tracing facility, is very flexible, does not require invasive modification of existing code, and can trace all applications. Unfortunately, tracing can have a very significant impact on the performance of the traced program. Although we chose to make efficiency a secondary goal, we felt that we could not ignore the issue entirely, and this choice would have made the system too slow to be acceptable to users.

The third option (a pseudo-NFS server) suffers from a similar efficiency drawback. There is no way to intercept only opens and closes, without also intercepting reads and writes. Since we were not willing to pay a hefty performance penalty for every I/O operation, we rejected this choice.

Option 4, replacing the shared library, is very attractive for a number of reasons. By avoiding kernel changes, it greatly simplifies development. Also, potential users are often much more willing to accept a replacement library than a completely different kernel. However, UNIX implementations generally contain some programs that do not use the shared library, so these programs would either have to be re-linked, or would remain untraced. Worse, this approach would require every application to have some sort of communication path between itself and the SEER software. Under UNIX, establishing this path is difficult and time-consuming, and may cause unexpected side effects for applications that make assumptions about the values or availability of file descriptors.

The reader will note that the objections to option 4 are UNIX-specific. We have not rejected this option for other potential implementations, such as WINDOWS, where modification of Dynamically Loaded Libraries ("DLLs") is a standard method for accomplishing such tasks. However, for our current UNIX-based implementation, this approach does not satisfy our requirements.

The final option, modifying the kernel, presents greater implementation difficulties, but also offers significant advantages. The tracing can usually be done by capturing system calls at a single common point (with minor exceptions; see Section 6.14, p. 72), and maximum efficiency is possible because all of the required information is already available as part of the system call. Communication with the rest of SEER can be implemented through "extracurricular" means without affecting the traced processes. All applications can be traced, regardless of provenance, and the cost of tracing is paid only for those calls that are actually captured. Because of these considerations, we chose to add tracing code directly to the kernel for our UNIX and LINUX implementations.

Observation under Unix

Kernel Hook As discussed above, the implementation we chose for UNIX involves modifying the kernel. However, we were careful to limit and encapsulate these modifications to ease portability to other systems.

Most UNIX and UNIX-like systems use a single point of entry into the kernel to provide system-call services to client applications. The user-level process provides a *system-call number* and a list of arguments specific to the particular operation. The format and manner of passing the system-call number and arguments vary between UNIX implementations and processor architectures, but they are always readily available to kernel code.

It is the common point of dispatch that greatly eases the task of observing system calls. In our modification, we added a new routine, `observer_syscall`, which is called both before and after every system call is executed. Arguments to `observe_syscall` include the system-call number, a flag indicating the circumstances of the call (before or after dispatch, or a special-case call), a pointer to the arguments of the call, and return values from the call (if the trace is collected after the call is executed).

The tracing routine performs a table lookup on the system-call number and circumstances to determine whether a trace should be collected. If so, it is placed in a buffer for later delivery to the `observer`. If the buffer is full, and an `observer` is actually running, the traced process is blocked until space is available. If there is no `observer`, however, the oldest saved trace is simply discarded, preventing deadlock at startup or if the `observer` crashes.

Table B.1 lists the system calls traced under UNIX.

`access`	`chdir`	`chroot`	`close`	`creat`
`dup2`	`dup`	`execve`[b]	`execv`[b]	`exit`[c,e]
`fchdir`	`fchroot`	`fcntl`	`fork`[d]	`link`
`lstat`	`mkdir`	`mknod`	`open`	`readlink`
`rename`[e]	`rmdir`	`stat`	`symlink`	`truncate`
`unlink`[e]	`utimes`	`vfork`[d]		

[a]Traced before system call executes.
[b]Traced both before and after system call executes.
[c]Not traced through the common dispatch routine.
[d]Traced through common dispatch under LINUX but not SUNOS.
[e]Traced even for the super-user.

Table B.1: Tracing of UNIX System Calls.

The variable-length trace packet is designed to minimize space requirements. For each traced call, the kernel records an internal code indicating the system call invoked, the return value of the call, the error code (if it failed), the exact time in microseconds, the user ID, the ID of the invoking process, and a set of flag bits. In addition, one or more selected arguments, usually filenames, are captured.

Special-Case Traces Certain UNIX system calls do not lend themselves to tracing via the common point of dispatch, for reasons related to the internal implementation of the kernel. These calls are discussed more extensively in Section 6.14 (p. 72). To capture them, we were forced to modify the call-processing routines themselves. Fortunately, only a very small number of calls need such treatment (see notes *c* and *d* in Table B.1).

Untraced Processes Because tracing is done in a blocking fashion, deadlock is a possibility if any process in the SEER subsystem is itself traced. For example, the system calls made by the `observer` must be ignored by the tracing subsystem.

SEER uses several mechanisms to avoid deadlock. First, all system calls made by the `observer` are ignored. Second, any process may suppress tracing by opening a special pseudo-device, `/dev/seer_notrace`. Third, because certain system daemons are invoked indirectly by SEER, most calls made by the super-user (`root`) are also ignored (see Section 6.13, p. 72, for more information).

Trace Driver The traces collected in the kernel are made available to the user-level `observer` via a standard UNIX driver interface. Each `read` call returns one or more complete trace records to the `observer`, removing them from the buffer in the process. The driver does not support `write`s.

B.1.2 Trace Packet Processing

The trace packets are read from the kernel and processed by the `observer`, as outlined in Section 5.2, p. 52, before being passed on to the `correlator`.

Most of the packets sent from the `observer` to the `correlator` are changed very little from what the kernel provides. However, there are three notable exceptions:

File Handles Some system calls (*e.g.* `close`) operate on *file handles* returned by the `open` call, rather than file names. To simplify processing in the `correlator`, the `observer` keeps track of the relationship between file names and file handles, and converts between the two.[1]

[1]Under UNIX, this tracking is complicated by the `dup` system call and by the replication of file descriptors when processes are spawned.

Relative Pathnames When a file is referenced by a user process, the programmer has the option of specifying a pathname relative to the so-called *current working directory* of the process. Again, the `observer` simplifies processing for the `correlator` by converting these relative names into their absolute equivalents.

Canonicalization Normally, the pathnames sent to the `correlator` are absolute, but are not necessarily in their simplest form, and they may contain references to symbolic links. Canonicalization is handled by code inside the `correlator` itself. However, if the `correlator` is not running on the same machine as the `observer`, some of this canonicalization may need to be done locally (because the symbolic links may not exist on the remote machine).

Packets are sent to the `correlator` using standard inter-process communication facilities. To ensure that no traces are lost, they are sent in blocking mode. If the `correlator` is talking to multiple `observer`s (see Section 5.2, p. 52), it will read and process packets in timestamp order. Ordered processing assumes that clocks are synchronized fairly well, which is true in modern systems that run a protocol such as NTP [Mills 1989, Mills 1994]. Timestamp synchronization is discussed further in Section B.2.2.

B.2 Correlator Design

As outlined in Section 5.3 (p. 53), the `correlator` is the heart of SEER and has many responsibilities, which are reviewed in that section. Details of these operations are given below.

B.2.1 Interprocess Communication

The various parts of SEER communicate with each other using standard networking facilities. For simplicity, all communication is done via reliable bidirectional streams; in UNIX, sockets are used for this purpose. The `correlator` is the master process and operates in a "server" mode, accepting connections from other processes as necessary.

`Observer/Correlator` Communication

When the `observer` starts up, it attempts to open connections to one or more `correlator`s. Each `correlator` is chosen based on command-line arguments that identify the location of the `correlator` and the user it is monitoring. At the same time, the `observer` also opens up the SEER kernel device and begins reading trace packets. Each packet is distributed to zero or more `correlator` connections, depending upon the characteristics of the packet and the user observed by each `correlator`.

From time to time, the `observer/correlator` connection may be lost. For example, a `correlator` running on a portable machine may be removed from the network when the user takes it home. Upon disconnection, the `observer` will continue to run and to provide packets to any still-connected `correlators`. Packets destined for disconnected `correlators` are discarded.[2] The `observer` makes frequent attempts to reconnect to disconnected `correlators`, and when an attempt is successful, begins delivering packets again.

The assumption underlying this behavior is that the user serves as a human "activity token" that follows the `correlator` through the world. Thus, when the machine with the `correlator` is disconnected from the network, the user is also disconnected, and therefore he or she will generate no significant activity. This inactivity makes it acceptable to discard the few packets destined for that user. A more ambitious system might wish to preserve these packets for later delivery, but we chose to avoid that complication.

The `correlator` is capable of accepting simultaneous input from multiple `observer`s and `controllers`. Under UNIX, multiple inputs are implemented by using the `select` system call. Control commands are generally executed immediately. If there are multiple `observers` (presumably running on different machines), their packets are queued briefly and processed in timestamp order. However, this timestamp sorting applies

[2]However, they are still written to the debugging save file, if any.

only to `observer`s that actually have input ready. If a particular connection does not have an incoming packet, the `correlator` does not wait for the next packet before comparing timestamps. This skipping of empty connections is necessary to prevent the `correlator` from blocking for arbitrarily long periods, which would eventually cause the system to hang when the connections to other `observer`s backed up.

Since `observers` (and `controllers`) may run on machines with a different architecture than the `correlator`, the latter program will perform byte swapping when necessary.

The `observer` generates packets of the following types:

`CONTROL` A "fake" packet generated by the `observer` to communicate control information, such as whether the `observer` is replaying a saved debugging file or reporting packets in real time.

`NEWPROC` Creation of a new system process, *e.g.* via `fork`.

`EXEC` Execution of a file by a running process.

`INTERPRET` Execution of a script interpreter. Scripts require two files, the script itself and an interpreter to run it. The script is reported by the kernel as an `EXEC`, but the interpreter is discovered and reported by the `observer`.

`EXIT` Exit of a running program.

`PROCEXIT` Termination of a system process. The difference between `EXIT` and `PROCEXIT` is that the former is like a close, indicating that the file referenced by the last `EXEC` is no longer active. The latter records the final destruction of the process itself, and allows the `correlator` to remove that process from its internal tables.

`READ` Read access to a file without opening it (*e.g.*, `chdir`).

`RW` Read/write access to a file without opening it (*e.g.*, an attempt to open an unhoarded file for read/write).

`CREATE` Create (write) access to a file without opening it.

`DESTROY` Destruction of a file or directory.

`RENAME` Rename or move of a file or directory.

`OPENREAD` Open of a file for reading.

`OPENWRITE` Open of a file for reading and/or writing.

`OPENCREATE` Open of a file for creation (truncate/rewrite).

`CLOSE` Close of a previously opened file.

`PEEK` Query regarding a file's attributes (*e.g.*, permissions or modification time).

`MAKELINK` Creation of a symbolic link.

`PEEKLINK` Access to the contents of a symbolic link (in UNIX, `readlink`).

`DUP` Duplication of an existing file descriptor, creating a second reference to the open file.

In addition, the following reference types are not generated by the `observer`, but are used internally by the `correlator`:

`UNKNOWN` Reference of an unknown type, used to detect uninitialized packets.

`FOLLOW` Access to the target of a symbolic link, *e.g.* by opening the file that the link points to.

`Controller/Correlator` Communication

Like the `observer`, the `controller` must communicate with the `correlator` so that the user may set parameters, query status, and perform various control functions. The `controller` uses the same client/server mechanisms as the `observer`, but it does not attempt to force its commands and responses into the `observer`'s packet format. Instead, an *ad hoc* communication format is used.

Each interaction is introduced by a single command byte, followed by optional parameters whose format is specific to the command. Usually, these optional parameters use a compact binary encoding.

If the command requires a response, the `correlator` generally provides it in an ASCII stream format (which is convenient because the `controller` can just copy it to its own standard output device, if desired). The `correlator` allows multiple commands to be sent sequentially over a single connection, although the `controller` does not currently take advantage of the capability.

The `correlator` accepts the following commands from the `controller`:

`EXIT` Save the correlation database to a file and terminate the `correlator`.

`RESTART` Save the database to a file and then re-execute the `correlator`, maintaining all connections to `observer`s. This function is useful for debugging, to invoke a new version of the `correlator` after changes have been made, while ensuring that no packets are lost.

`SAVE` Checkpoint the correlation database to a file.

`GETPARMS` Retrieve the manager-specific parameters of a named hoard manager, or of all hoard managers.

`SETPARMS` Set one or more manager-specific parameters.

`MANAGE` Run all hoard managers.[3]

`MANAGE_DUMP` Run a particular hoard manager and print manager-specific information (usually results) to the stream connection.

`READ_INSTRUCTIONS` Read or reread a file containing special instructions for the `correlator`.

`SET_INVESTIGATION` Read relations or clusters provided by an external investigator and add them to or remove them from the database.

`SET_STORAGE` Inform the `correlator` of changes in the storage status of files that are to be made by the underlying replication system. This command is an important part of the interface between the `correlator` and some replication substrates, and will be discussed further in Section 5.5, p. 58.

B.2.2 Data Structures

Like any large program, the SEER `correlator` has a number of important internal data structures. This section discusses those structures and their contents.

Manager Table

As discussed in Section 5.3.1, p. 53, the `correlator` supports multiple hoard managers to control hoarding decisions. The manager facility has also turned out to be convenient for certain other purposes. For example, the *save* manager makes no hoarding decisions, but periodically causes the internal database to be checkpointed to a file so that it will be preserved against crashes.

All manager classes are derived from a common base class. Managers that make hoarding decisions are derived from an intermediate base class that provides additional abstractions relevant to hoarding.

[3]This option is now little-used.

The manager base class maintains a table of all known managers, and provides facilities for locating a manager by name, querying and setting parameters, deciding whether hoard management is due, running the management or pseudo-management routines, tracking file references, and collecting and displaying statistics. Hoarding managers also provide routines for locating their private data in the file table and for inheriting information from the parent directory when a file is first entered into the table.

There are currently eight managers, five of which actually make hoarding decisions. The hoarding managers are described in Section 5.3.1. The remaining three managers are:

Global The global manager maintains parameters that are common to multiple managers, such as the total size of the hoard. All functions other than those related to parameters are no-ops.

Deletion The deletion manager handles garbage collection in the file table. It is invoked either periodically or when the number of deleted files grows beyond a threshold parameter.

Save The save manager periodically checkpoints the file table to disk, to protect against crashes.

Initially, we followed the design of CODA [Kistler and Satyanarayanan 1992] in allowing management to be initiated either periodically or by explicit user request. However, we found that the periodic feature was not useful in our environment, and it added unnecessary system load, so we disabled it. We do not believe that automatic management is an inherently bad idea, but we found that users will only accept it if the imposed load is low. Periodic initiation is still used by the save and deletion managers.

Management is scheduled from the main loop of the `correlator`, by calling a virtual function that decides whether the manager should run. The base class provides a function to handle time-based management; more complex decisions can be made by the derived class. Currently, only the deletion manager uses this feature. Deletion management is run either when a fixed interval expires, or when the number of files marked for deletion exceeds a threshold. (There are several phases of deletion, so the threshold is applied to the total number of files ready to enter each phase. The current threshold is 200 files per phase.)

File Table

The most important data structure in the `correlator` is the *file table*, which tracks information about all known filesystem objects. The file table is hashed on the absolute pathname of the file,[4] but most internal access is done using pointers to particular files.

As discussed in Section 5.1, p. 52, we chose to simplify the research implementation by choosing flexibility over performance. One of the most important consequences of this decision is that we store the file table in main memory. A production version of SEER would probably store the file table on disk to minimize memory requirements.[5]

General File Information The entry for a particular file stores general information about the file, plus data needed to track file relationships and information used by particular hoard managers. The general information includes:

File pathname. The absolute pathname of the file, canonicalized by removing unnecessary slashes, "." and ".." references, and symbolic links. If the file resides on an automounted NFS volume, the pathname includes the temporary mount point (usually named `tmp_mnt`).

Strip length. For automounted files, it is sometimes necessary to refer to the file by its user-visible name, which causes the automounter to mount the volume if necessary. The "strip" length gives the number of characters to strip from the front of the pathname to get the user-visible name. This limited form of

[4]Files are identified internally by their pathname, rather than an implementation-specific identifier such as the i-node number. Using the pathname adds some complexity, but enhances portability.

[5]An ideal system would probably store all SEER-specific information as file attributes similar to the data stored in the UNIX i-node.

pathname rewriting has proven sufficient to date, although it is possible that a more complex form will be needed in the future. The issue of whether to use the stripped or unstripped name in a particular case is discussed further in Section 6.11 (p. 70).

Flags. Various bit flags, discussed below.

Type. The file type (plain file, directory, symbolic link, etc.).

State. The current "state" of the file-table entry. Legal states include unknown (certain file-table fields are not valid), known (file exists and all fields have been filled in), unstored (file exists but is not stored locally), and three deletion-related states (see Section B.2.2).

Size. The number of blocks the file would require in the hoard.

Parent Directory. A pointer that can be used to locate the parent directory.

Children. If this object is a directory, a list of the known children, and a count of the number of unknown children. (The latter is used to decide whether a process is engaging in meaningless activity; see Section 6.1, on page 63, for more information.)

Link Target. If this object is a symbolic link, a pointer to the target of the link.

The one-bit flags in the file entry include:

Ignore. Completely ignore all references to this file. This feature is useful for temporary files, which are too transient to contribute useful information to SEER. See Section 6.6, p. 67, for more information.

Meaningless. If this file is executed as a process, consider its references to be meaningless. See Section 6.1, p. 63, for more information.

Controlled. This file is controlled by the automated hoarding mechanism. (This bit is set for most files, but can be left unset for certain critical system files.)

Must save. Even if this file disappears, its file-table entry must remain because it contains important flag bits.

Writing. This file is currently being written. When it is closed, the `correlator` will recalculate its size.

Fake size. The size of this file was falsified for debugging purposes.

Directory size discovered. For directories, the correct number of children is known.

Storage set. The storage status of this file was set externally. This flag is used in simulations to help distinguish (simulated) unstored files from those that have been deleted.

Marked. A scratch bit used by internal algorithms.

The ignore, meaningless, and controlled flags are inherited by children of a directory when they are first created. Thus, for example, `/tmp` can be marked with the ignore flag so that all temporary files will be ignored.

Tracking Reference Activity A number of fields in the file table track general reference activity. These fields are not specific to any particular hoard manager, although not all managers use all of the fields. They include:

Open count. A count of the number of times the file has been opened, less the number of times it has been closed. This value can be used by managers to determine if any process currently has the file open (although managers can also use a similar process-specific field; see Section B.2.2). It is also used in conjunction with the `writing` flag to decide when to recalculate a file's size.

Total references. The total number of references to this file observed since the beginning of time. Note that this field, like many other reference-related fields, is persistent across reboots and re-invocations of SEER.

Last reference. The reference number of the last reference to this file, counted since the beginning of time, and the type of that reference (open for read, etc.; see Section B.2.1).

Peek count. If the file represents an executable process, the number of `PEEK` references that have been performed in all executions of the process.

Potential peeks. If the file represents an executing process, the number of children in all directories that have been examined by the process in all execution lifetimes. This value, together with the peek count, is used to infer meaninglessness (see Section 6.1, p. 63).

File Relations The basis of SEER's operation is the detection of file relationships. To support this capability, the file table provides a very general mechanism for describing relations between files. Each file contains a list of all related files (the relations are one-way: if A lists B as related, B may or may not list A).

Relations are classified into various types. For example, a particular manager could choose to create a manager-specific relationship type, meaningful only to itself, and add such a relation between selected file pairs. Relation types can also be created by external investigators (Section 5.4, p. 56). Each type is identified uniquely by a type-control object that names the type and serves as an access point for code that needs to iterate through all relation types. The type-control object also provides a function that can be called to record a reference to a particular file; this function can be used by individual relation types to implement concepts such as semantic distance.

In the following discussion, if file B appears in file A's related-file list, B is called the *target* of the relation, and A is the *source* file.

Each element on the related-file list is of class `RelatedFile`. This entry identifies the target and contains a list of relations, of different types, between the source and target. The structure is designed to allow a single pair of files to be examined quickly for multiple relation types, and to easily support arbitrary relationships.

File relations are represented by objects derived from a common base class. The base class contains only a back-pointer to the associated `RelatedFile` object. The relation type can be inferred using a C^{++} virtual function; all other data is specific to the individual derived classes. Two relation iterators are also provided by the `File` class: one walks all of a file's related-file entries, while the other limits itself to relations of a selected type.

File References The most heavily used relation is the file reference, which tracks the semantic-distance information discussed in Chapter 3. The derived class provides three fields: a reference count, a total distance, and a "corrected" flag. These fields are used to implement the semantic-distance calculation of Section 3.5.4 (p. 45), all of which is done by the reference-recording routine for the file-reference relation.

Investigated Relations Relations created by external investigators cause a named relation type to be created, with the name provided externally. Because the precise type (name) is not implied by the virtual function call, an investigated relation stores a reference to the underlying type object so that it may be located when necessary. It also keeps a weight, again provided by the external software, which is used by the clustering algorithm to integrate this relation into its decisions.

Clusters The clustering manager forms files into clusters. A cluster is represented as a linked list of files, a storage state (stored, unstored, etc.), a hoard weight, a file count, and the total size (in blocks) of all files in the cluster.

Clusters can be created by arbitrary means, but the cluster class provides some facilities for use in hoarding. In particular, the hoard-weight field can be set by a manager to indicate the desirability of keeping this particular cluster in the hoard. The clustering manager creates clusters dynamically, using the algorithm described in Section 4.2 (p. 49).

External investigators may also create clusters. The mechanism is very similar to that used for externally investigated relations. When a cluster of a given type is encountered, a new type object is created to support it, and then the cluster is built using a special derived class. The clustering manager will then integrate these clusters into its decisions.

Manager-Specific Data Besides generic data, the file table also stores information on behalf of individual managers. For efficiency, a generalized implementation was not chosen for this purpose. Instead, the file class embeds the manager-specific data directly in the file object. Virtual functions provided by the managers allow access to this data (as an opaque object) when necessary.

LRU-Based Managers SEER provides four LRU-style managers: simple LRU, bounded LRU, weighted LRU, and linear LRU. These managers are described in Section 5.3.1, p. 53. The simple LRU manager does not require any manager-specific data; the other three store integer or floating-point parameters appropriate to their particular transformation.

One of the virtual functions provided by the manager-data base class is parental inheritance. When a file is first created, each manager's inheritance routine is called so that the manager may derive data from the file's parent directory, if desired. It is this feature that allows the user to specify parameters for an entire subtree by simply mentioning the root. The inheritance routine is called before any value from the instruction file is applied, so that upper-level values can be overridden by the instruction file for files deeper in the hierarchy.

Clustering Manager The clustering manager is the most complex hoard manager, but most of the data it needs is already provided by the file object. (Although the clustering manager is currently the only one that makes use of file-reference information, it is not inherently specific to that manager, so it is kept generally rather than privately.) The private data consists of several flags that are used in the clustering process (for efficiency rather than as a fundamental part of the algorithm), plus a list of clusters of which this file is a member.

Deletion Manager As mentioned in Section B.2.2, several pseudo-managers do not make hoarding decisions, but use the general management mechanism to achieve other goals. One of these pseudo managers handles deletion of entries from the file table. It does not keep any data of its own in the file object, but it is discussed here because it is responsible for garbage collection in the file table.

Deletion takes place in two or three stages. The first stage, `MISSING`, is entered under two conditions:

- Failure to `stat` a file. This case usually occurs when a saved log is being replayed, and there is a reference to a file (usually a temporary file) that has since been deleted. It can also occur if the `correlator` has been delayed in its reading of `observer` packets, so that it first learns of the existence of a very short-lived file after that file has disappeared.

- A file has inherited an ignored (see Section B.2.4) status from its parent. Such files take up unnecessary space in the file table, since the status can be reconstructed as needed.

The `MISSING` stage was originally designed to allow for certain cases when a file might be transiently deleted. However, it is now obsolete and remains only for historical reasons.

The second stage of deletion, `FORGETTING`, is entered under the following conditions:

- When the deletion manager encounters a `MISSING` file in the table.
- When a file is renamed (for implementation reasons, some renames are handled by creating an entry under the new name, marking the old name for deletion, and copying appropriate semantic data).
- When an `unlink` or `rmdir` operation is observed for the file (except when such calls are generated by the underlying replication system).

The purpose of the second stage is to allow destruction of all pointers to the to-be-deleted entry. For efficiency, many internal classes maintain pointers to various files (for example, the file relations discussed in Section B.2.2 use such pointers heavily). When a file-table entry must be deleted, these pointers must be invalidated. Rather than keep back pointers to all such structures (which is clumsy and wasteful of space), or walk all such structures at the moment of deletion (which is wasteful of time), SEER batches the pointer-invalidation operation with its multi-stage deletion.

For most purposes, `FORGETTING` files are treated as nonexistent. However, since they are still present in the table, pointers to them are still valid. When the deletion manager runs, it walks through the file table and all classes that maintain pointers to files, invalidating all pointers that refer to `FORGETTING` or `FORGOTTEN` files. At the same time, it converts `FORGETTING` files to `FORGOTTEN`, and deletes those that were previously `FORGOTTEN`. This batched design amortizes the cost of the table walk across a large number of files. At the end of the walk, any file marked `FORGOTTEN` is guaranteed to have no pointers targeted at it, and can be safely deleted in the next table walk.

A fortuitous side effect of this approach is that destroyed files remain in the file table for a brief period. Since many programs delete and immediately recreate files, the delay allows the semantic relationships regarding those files to be maintained after their disappearance, in case they return. If not, they will be removed shortly.

Deletion management is run periodically (once an hour in the current implementation), plus whenever the number of pending deletions exceeds a threshold. The latter condition was added to protect against excessive table growth in certain situations where very large numbers of files enter the chain of deletion states in a short period of time.

Process Table

As discussed in Section 6.8 (p. 68), references generated by various processes must be separated so that the calculation of semantic distance does not suffer from undesirable noise effects. The `observer` reports the UNIX process ID of the process generating each reference. The `correlator` then maintains a table, hashed by process ID, of all known processes. The most important element of the process object is a process-specific history of files referenced. When a new reference arrives, the `correlator` evaluates it using only the history of the current process.

Besides the reference history, the process-table entry contains a flag field, a pointer to the file-table entry for the currently running program, a list of open files, and a count of directories that the process has open. The open-file list is used to determine whether the process has a particular file open (the file table only records whether *some* process has opened the file). The directory-open count and the pointer to the running program are used to infer meaninglessness (see Section 6.1, p. 63). Finally, a pointer to the parent process allows the a process's file history to be merged with the parent's when the process exits. The merging allows SEER to infer semantic relationships among files referenced by a child on behalf of a parent (for example, it will correctly handle the files referenced by commands spawned from a shell script).

The process table currently supports two flags. The `MEANINGLESS` flag is set if the currently running program has been marked as meaningless in the instruction file; this indicates that the `correlator` should ignore all references generated by this process and its children. The `IN_GETCWD` flag indicates that the `correlator` has inferred that the process is currently performing a `getcwd` library operation, which generates a short stream of references that are lacking in semantic information and should be ignored. Both of these flags are discussed further in Section 6.1.

Connection Table

Since the `correlator` can accept information from multiple `observer`s on multiple machines, a connection table tracks all `observer`s. For historical reasons, the primary class in the connection table is named `Packet`. This class contains the most recent packet received on the connection (including a timestamp supplied by the `observer`), the process table for the connection, and miscellaneous bookkeeping fields. The process table is kept on a per-connection basis because each `observer` is assumed to be running on a separate machine with a separate process history (this restriction is enforced because `/dev/seer` will only accept a single open).

When the `correlator` is ready to accept a new packet from an `observer`, it chooses the lowest timestamp from the pending packets on all connections. If a connection does not currently have a pending packet, the `correlator` skips it in this computation. This design assumes that the clocks on the observed machines are reasonably well synchronized, which will be true if they are running a time-synchronization protocol such as NTP [Mills 1989, Mills 1994]. If they are not, it is possible that the `correlator` will see some packets that are out of timestamp order,[6] but the validity of the results will not be affected because the timestamps are not used in the actual semantic-distance calculations. The primary purpose of this feature is to ensure fair treatment of all machines and proper ordering of complex tasks that span multiple machines.

If no connection has a pending packet, the `correlator` will wait until at least one connection is ready, a new connection is created, or a timer expires to indicate that management should be run.

The connection table only records connections from `observer`s. `Controller` connections are handled separately.

B.2.3 Tracking File References

When an incoming packet arrives on an `observer` connection, the `correlator` extracts the packet type and parameters, and then processes the packet according to type. Most packets are passed directly to the `File` class, although process-related packets (new process and process exit) receive special treatment.

The work of reference tracking is divided between the `File` class and the various relation classes. The `File` class tracks file opens, updates the last reference type and time, and tracks meaninglessness (see Section 6.1) and `getcwd` activity. It then passes the reference on to each relation type and to each manager for further processing.

The reference-recording routine for the investigated-relation types is null, but the `FileReference` class does substantial work, implementing Algorithm AT-KSK (Section 3.5.4, p. 45) to track reference histories and update semantic distances. The semantic distance calculation is discussed further in Section 5.3.2, p. 54.

The existing managers only track file references for statistical purposes, recording whether a hoard miss occurred. This feature is used during simulation.

All file reference types that are involved in the semantic-distance calculation are processed as described in Section 5.3.2.

[6] In extreme cases, the machine with the fastest clock could be delayed because its `observer` would have to block waiting for packets to read.

B.2.4 Control Files

When SEER is installed on a system, the setup scripts create a number of control files to contain parameters, logs, etc. Many of these files are artifacts of the current implementation and are not interesting from either a research or an engineering standpoint, but a few are worth consideration. They include:

- A save file for the `correlator` database, used to ensure that file relations are persistent in the long term (see Section B.2.2).
- A list of subtrees controlled by the underlying replication substrate; these are the trees that SEER can work with to manage the hoard contents.
- Various log files used to collect statistics regarding hoard misses, the disconnection history of the machine, recent error messages, etc.
- Trace files that record all packets sent from the `observer` to the `correlator`. These logs are unnecessary in a production environment, but were collected so that different hoard-management methods could be simulated and analyzed. They have also proven invaluable in debugging and in recovery from serious failures.
- The master instruction file, outlined in Section 5.6.1, p. 60, and further described below.
- Manager-specific instruction files, described in Section 5.6.2, p. 61.

Instruction File

An overview of the master instruction file is given in Section 5.6.1. Specific features of the instruction file include:

- Specify that a particular subtree should or should not be managed as part of the hoard. Usually, the root tree is specified as unmanaged, and then the replication substrate lists its subtrees as being controlled by SEER. The system administrator might also choose to explicitly list certain critical directories, such as `/etc`, as uncontrolled (not managed by SEER).
- Specify that accesses to a particular file or subtree should be ignored by SEER. For example, `/tmp` is usually listed as an ignored tree, since files in that directory are so short-lived that there is no point in tracking them (and attempting to do so would interfere with semantic-distance calculations). See Section 6.6 (p. 67) for more information on this feature.
- Specify that a particular program is meaningless (see Section 6.1, p. 63). The meaningfulness or meaninglessness of most programs can be automatically inferred by SEER, but a few are tricky enough to require hand specification. Our current instruction files list only one standard UNIX program (`xargs`) as meaningless. In addition, most of the programs of the SEER system itself and the underlying replication system are so listed. This latter specification is not strictly necessary but serves as a safety check when debugging.
- Specify a falsified size for a file. This feature is used for debugging, so that the hoarding of large files can be tested without actually wasting disk space by creating them.
- Specify a string to be stripped from a pathname under certain conditions, to work around problems caused by automounters (see Section 6.11, p. 70).
- Specify instructions or parameters specific to a particular manager.

B.2.5 Manager Parameters

As discussed in Section B.2.2, SEER's managers maintain various parameters that can be used to control their behavior. Many of these parameters were chosen using the techniques explained in Section 6.12, p. 71. The complete list of parameters and their current values is given below.

Global Manager

Age Weighting Factor = 1.2. A weighting factor used in aging references when choosing one to replace; see Section 5.3.2, p. 54, for more information.

Aging Threshold = 25,000. A threshold used to decide when an reference is old enough to consider replacing with a newer one; see Section 5.3.2, p. 54, for more information.

Often-referenced file threshold = 1%. A threshold used to decide when a file is so frequently referenced that it should not participate in semantic-distance calculations; see Section 6.2, p. 65, for details.

Peek threshold = 40%. A threshold used to decide when a program has looked at so many files (compared to the number it could potentially access) that its activity must be meaningless. See Section 6.1 (p. 63) for more information.

Hoard size = 50 Mb. The maximum amount of space to use for the hoard. This varies from user to user; see Table 8.8, p. 95 for a list of the hoard sizes chosen by the users in our study.

Maximum related files = 20. The number of relationships to keep for any given file. This is the parameter k in AT-KSK, described in Section 3.5.4, p. 45.

Use process trees = true. This Boolean variable allows the `correlator` to be run without considering references from different processes to be independent, as discussed in Section 6.8, p. 68. It is used for comparing different hoarding approaches.

Deletion Manager

Deletion trigger = 200. A threshold used to decide when to run the deletion manager. If there are more than this many files waiting for deletion processing, the manager will run.

Management interval = 1 hour. The minimum time between deletion manager invocations. The manager will run at least this often even if the trigger threshold has not been reached.

Save Manager

Management interval = 1 hour. The minimum time between saves of the `correlator`'s file database.

Clustering Manager

Use shared-neighbor similarity measure = true. A Boolean variable allowing an alternate, obsolete measure of file similarity to be used in place of the measure described in Section 4.2.2, p. 49.

Management interval = 0. The time between automatic invocations of the clustering manager. The zero value suppresses these periodic invocations, which we have found to be unnecessary.

Directory distance weight = 1.0. The weight given to the directory-distance similarity measure; see Section 4.2.2 (p. 50) for more information.

Investigated relation weight = 1.0. The weight applied to information acquired from external investigators. This is a global value for all investigators; each investigator can also specify an overall weight (currently 1.0) as well as the weights for individual files. See Section 4.2.2 for more information.

Minimum references for distance calculation = 3. The minimum number of file references that the `correlator` must see before it will attempt a semantic-distance calculation. See Section 4.2.2, p. 49, for more information.

Threshold for tight clustering = 14. The threshold for clustering files together tightly. This is n_1 in Section 4.2.2 (p. 49).

Threshold for loose clustering = 12. The threshold for generating overlapping clusters. This is n_2 in Section 4.2.2.

In addition, the clustering manager supports several other parameters related to the obsolete clustering method mentioned above; we will not discuss those here.

LRU-Style Managers

All of the LRU-style managers support only a single parameter, the management interval that selects the amount of time between automatic management runs. The parameter is set to zero (no automatic management) for all of these managers.

B.3 Controller Design

As discussed in Section 5.6 (p. 60), a program called the `controller` provides facilities allowing interaction with the `correlator`. The controller supports the following functions:

- Display or set any `correlator` parameter.
- Read or reread an instruction file (see Section 5.6.1).
- Load information into the `correlator` from an external investigator (Section 5.4).
- Write the `correlator`'s internal database to a save file, optionally exiting after the save completes.
- Run a specific manager, returning the results of management to standard output in a concise ASCII format.
- Run all managers in sequence.
- Set falsified reference times for a specified list of files.
- Restart the `correlator` from a recompiled executable without dropping any connections or losing `observer` trace packets.
- Inform the correlator of the current storage status (locally hoarded or not) of one or more files, simultaneously telling it the sizes of any that are not locally accessible.

All of these functions are implemented primarily in the `correlator`, with the `controller` providing interface and communication functions. The `controller` accepts information from the command line or its standard input and converts it into packets appropriate for communication across a socket to the `correlator`. Similarly, responses or results are extracted from the IPC socket and copied to standard output for display or further processing.

Trademarks

References

[Akyürek and Salem 1995] Sedat Akyürek and Kenneth Salem. "Adaptive Block Rearrangement." *ACM Transactions on Computer Systems*, **13**(2):89–121, May 1995.

[Alexandrov *et al*. 1997] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. "Extending the Operating System at the User-Level: the Ufo Global File System." In *USENIX Conference Proceedings*, pp. 77–90, Anaheim, California, January 1997. USENIX.

[Alonso *et al*. 1990] Rafael Alonso, Daniel Barbará, and Luis L. Cova. "Using Stashing to Increase Node Autonomy in Distributed File Systems." In *Proceedings of the Ninth IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 12–21, October 1990.

[Baker *et al*. 1991] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Sherriff, and John K. Ousterhout. "Measurements of a Distributed File System." In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 198–211. ACM, October 1991.

[Beresford 1995] Mike Beresford, 1995. Personal communication.

[Blaze and Alonso 1992] Matthew Blaze and Rafael Alonso. "Dynamic Hierarchical Caching for Large-Scale Distributed File Systems." In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, pp. 521–528, June 1992.

[Bock 1974] Hans Hermann Bock. *Automatische Klassifikation: Theoretische und praktische Methoden zur Gruppierung und Strukturierung von Daten*. Vandenhoeck & Ruprecht, Göttingen, Germany, 1974.

[Buck and Coyne 1991] A. Lester Buck and Robert A. Coyne, Jr. "Dynamic Hierarchies and Optimization in Distributed Storage Systems." In Karen D. Friedman and Bernard T. O'Lear, editors, *Proceedings of the Eleventh IEEE Symposium on Mass Storage Systems*, pp. 85–91, Monterey, California, October 1991. IEEE Computer Society Press.

[Cao *et al*. 1994] Pei Cao, Edward W. Felten, , and Kai Li. "Implementation and Performance of Application-Controlled File Caching." In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pp. 165–178. USENIX, November 1994.

[Curewitz *et al*. 1993] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. "Practical Prefetching via Data Compression." In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 257–266. ACM, 1993. Also available as SIGMOD Record 22, 2 (June 1993).

[Drakopoulos and Merges 1991] Elias Drakopoulos and Matt Merges. "Performance Study of Client-Server Storage Systems." In Karen D. Friedman and Bernard T. O'Lear, editors, *Proceedings of the Eleventh IEEE Symposium on Mass Storage Systems*, pp. 67–72, Monterey, California, October 1991. IEEE Computer Society Press.

[Duchamp 1997] Dan Duchamp. "A Toolkit Approach to Partially Connected Operation." In *USENIX Conference Proceedings*, pp. 305–318, Anaheim, California, January 1997. USENIX.

[Duran and Odell 1974] Benamin S. Duran and Patrick L. Odell. *Cluster Analysis: A Survey*, volume 100 of *Lecture Notes in Economics and Mathematical Systems*. Springer-Verlag, New York, 1974.

[Ebling *et al*. 1994] Maria R. Ebling, Lily B. Mummert, and David C. Steere. "Overcoming the Network Bottleneck in Mobile Computing." In *Proceedings of the Workshop on Mobile Computing Systems and Applications 1994*, pp. 34–36, December 1994.

[Floyd and Ellis 1989] Richard A. Floyd and Carla Schlatter Ellis. "Directory Reference Patterns in Hierarchical File Systems." *IEEE Transactions on Knowledge and Data Engineering*, **1**(2):238–247, June 1989.

[Foglesong *et al*. 1990] Joy Foglesong, George Richmond, Loellyn Cassell, Carole Hogan, John Kordas, and Michael Nemanic. "The Livermore Distributed Storage System: Implementation and Experiences." In Karen D. Friedman and Bernard T. O'Lear, editors, *Proceedings of the Tenth IEEE Symposium on Mass Storage Systems*, pp. 18–25, Monterey, California, May 1990. IEEE Computer Society Press.

[Foster and Habermehl 1991] Antony Foster and David Habermehl. "Renaissance: Managing the Network Computer and its Storage Requirements." In Karen D. Friedman and Bernard T. O'Lear, editors, *Proceedings of the Eleventh IEEE Symposium on Mass Storage Systems*, pp. 3–10, Monterey, California, October 1991. IEEE Computer Society Press.

[Garey and Johnson 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., New York, NY, 1979.

[Gibson *et al*. 1992] Garth A. Gibson, R. Hugo Patterson, and Mahadev Satyanarayanan. "Disk Reads with DRAM Latency." In *Proceedings of the Third Workshop on Workstation Operating Systems*, pp. 126–131, Key Biscayne, FL, April 1992. IEEE.

[Goel 1996] Ashvin Goel, 1996. Personal communication.

[Griffioen and Appleton 1994] James Griffioen and Randy Appleton. "Reducing File System Latency Using a Predictive Approach." In *Proceedings of the Summer USENIX Conference Proceedings*, Boston, MA, June 1994. USENIX. Also available as University of Kentucky Technical Report CS247-94.

[Grimsrud *et al*. 1993] Knut Stener Grimsrud, James K. Archibald, and Brent E. Nelson. "Multiple Prefetch Adaptive Disk Caching." *IEEE Transactions on Knowledge and Data Engineering*, **5**(1):88–103, February 1993.

[Gunter 1995] Michial A. Gunter, 1995. Personal communication.

[Gunter 1997] Michial Allen Gunter. *"Rumor: A Reconciliation-Based User-Level Optimistic Replication System for Mobile Computers."*. Master's thesis, University of California, Los Angeles, Los Angeles, CA, June 1997.

[Guy 1991] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. Ph.D. dissertation, University of California, Los Angeles, June 1991. Also available as UCLA technical report CSD-910018.

[Hartigan 1975] John A. Hartigan. *Clustering Algorithms*. Wiley, New York, NY, 1975.

[Heidemann *et al*. 1992] John S. Heidemann, Thomas W. Page, Jr., Richard G. Guy, and Gerald J. Popek. "Primarily Disconnected Operation: Experiences with Ficus." In *Proceedings of the Second Workshop on Management of Replicated Data*, pp. 2–5. University of California, Los Angeles, IEEE, November 1992.

[Hogan *et al*. 1990] Carole Hogan, Loellyn Cassell, Joy Foglesong, John Kordas, Michael Nemanic, and George Richmond. "The Livermore Distributed Storage System: Requirements and Overview." In Karen D. Friedman and Bernard T. O'Lear, editors, *Proceedings of the Tenth IEEE Symposium on Mass Storage Systems*, pp. 6–17, Monterey, California, May 1990. IEEE Computer Society Press.

[Honeyman *et al.* 1992] Peter Honeyman, Larry Huston, Jim Rees, and Dave Bachmann. "The Little Work Project." In *Proceedings of the Third Workshop on Workstation Operating Systems*, pp. 11–14. IEEE, April 1992.

[Howard *et al.* 1988] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[Huston and Honeyman 1993] L. B. Huston and Peter Honeyman. "Disconnected Operation for AFS." In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pp. 1–10. USENIX, 1993.

[Jain 1991] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

[Jarvis and Patrick 1973] R. A. Jarvis and E. A. Patrick. "Clustering using a similarity measure based on shared near neighbors." *IEEE Transactions on Computers*, **C-22**(11):1025–1034, November 1973.

[Kistler 1993] James Jay Kistler. *Disconnected Operation in a Distributed File System.* Ph.D. dissertation, Carnegie-Mellon University, May 1993.

[Kistler and Satyanarayanan 1992] James J. Kistler and Mahadev Satyanarayanan. "Disconnected Operation in the Coda File System." *ACM Transactions on Computer Systems*, **10**(1):3–25, 1992.

[Kleinrock 1994] Leonard Kleinrock, 1994. Personal communication.

[Kohl *et al.* 1993] John T. Kohl, Carl Staelin, and Michael Stonebraker. "HighLight: Using a Log-structured File System for Tertiary Storage Management." In *USENIX Conference Proceedings*, pp. 435–447. USENIX, January 1993.

[Korner 1990] K. Korner. "Intelligent Caching for Remote File Service." In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, May 1990.

[Kotz and Ellis 1990] David D. Kotz and Carla Schlatter Ellis. "Practical Prefetching Techniques for Parallel File Systems." In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, pp. 182–189, Miami Beach, Florida, December 1990.

[Krishna *et al.* 1997] P. Krishna, N. H. Vaidya, M. Chatterjee, and D. K. Pradhan. "A Cluster-based Approach for Routing in Dynamic Networks." *ACM Computer Communication Review*, **27**(2):49–64, April 1997.

[Kroeger 1996] Tom Kroeger, 1996. Personal communication.

[Kroeger and Long 1996] Thomas M. Kroeger and Darrell D. E. Long. "Predicting File System Actions from Prior Events." In *USENIX Conference Proceedings*, pp. 319–328, San Diego, California, January 1996. USENIX.

[Kuenning 1994] Geoffrey H. Kuenning. "Design of the SEER Predictive Caching System." In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[Kuenning 1995] Geoffrey H. Kuenning. "Kitrace: Precise Interactive Measurement of Operating Systems Kernels." *Software—Practice and Experience*, **25**(1):1–22, January 1995.

[Kuenning *et al.* 1994] Geoffrey H. Kuenning, Gerald J. Popek, and Peter Reiher. "An Analysis of Trace Data for Predictive File Caching in Mobile Computing." In *USENIX Conference Proceedings*, pp. 291–306. USENIX, June 1994.

[Kumar and Satyanarayanan 1993] Puneet Kumar and Mahadev Satyanarayanan. "Log-Based Directory Resolution in the Coda File System." In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, January 1993.

[Kumar and Satyanarayanan 1995] Puneet Kumar and Mahadev Satyanarayanan. "Flexible and Safe Resolution of File Conflicts." In *Proceedings of the USENIX Conference Proceedings*, p. xxx, New Orleans, LA, January 1995. USENIX.

[Kure 1988] Øivind Kure. "Optimization of File Migration in Distributed Systems." Technical Report UCB/CSD 88/413, University of California, Berkeley, April 1988.

[Lazowska *et al*. 1986] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. "File Access Performance of Diskless Workstations." *ACM Transactions on Computer Systems*, **4**(3):238–268, August 1986.

[Lei and Duchamp 1997] Hui Lei and Dan Duchamp. "An Analytical Approach to File Prefetching." In *USENIX Conference Proceedings*, pp. 275–288, Anaheim, California, January 1997. USENIX.

[Locus 1993] Locus Computing Corporation, Inglewood, California. *PC/Interface Reference Manual*, February 1993.

[Majumdar and Bunt 1986] Shikharesh Majumdar and Richard B. Bunt. "Measurement and Analysis of Locality Phases in File Referencing Behavior." In *Proceedings of Performance 86 and ACM Sigmetrics 86, Joint Conference on Computer Performance Modeling, Measurement and Evaluation*, pp. 180–192, Raleigh, NC, May 1986. ACM.

[Manber and Wu 1994] Udi Manber and Sun Wu. "GLIMPSE: A Tool to Search Through Entire File Systems." In *USENIX Conference Proceedings*, pp. 23–32, San Francisco, CA, January 1994. USENIX.

[Mecozzi and Minton 1991] Donna Mecozzi and James Minton. "Design for a Transparent, Distributed File System." In Karen D. Friedman and Bernard T. O'Lear, editors, *Proceedings of the Eleventh IEEE Symposium on Mass Storage Systems*, pp. 77–84, Monterey, California, October 1991. IEEE Computer Society Press.

[Mills 1989] Dave L. Mills. "Network Time Protocol (version 2) specification and implementation." RFC 1119, Internet Request For Comments, September 1989.

[Mills 1994] David L. Mills. "Precise Synchronization of Computer Network Clocks." *ACM Computer Communication Review*, **24**(4):28–43, April 1994.

[Mummert *et al*. 1995] Lily B. Mummert, Maria R. Ebling, and Mahadev Satyanarayanan. "Exploiting Weak Connectivity for Mobile File Access." In *Proceedings of the 15th Symposium on Operating Systems Principles*, pp. 143–155, Copper Mountain Resort, Colorado, December 1995. ACM.

[Muntz and Honeyman 1992] Dan Muntz and Peter Honeyman. "Multi-Level Caching in Distributed File Systems." In *USENIX Conference Proceedings*, pp. 305–313, San Francisco, January 1992. USENIX.

[Noble 1997] Brian Noble, May 1997. Personal communication via e-mail.

[Nydick *et al*. 1991] Daniel Nydick, Kathy Benninger, Brett Bosley, James Ellis, Jonathan Goldick, Christopher Kirby, Michael Levine, Christopher Maher, and Matt Mathis. "An AFS^{TM}-Based Mass Storage System at the Pittsburgh Supercomputing Center." In Karen D. Friedman and Bernard T. O'Lear, editors, *Proceedings of the Eleventh IEEE Symposium on Mass Storage Systems*, pp. 117–122, Monterey, California, October 1991. IEEE, IEEE Computer Society Press.

[Ousterhout 1990] John K. Ousterhout. "TCL: An Embeddable Command Language." In *USENIX Conference Proceedings*, pp. 133–146. USENIX, January 1990.

[Ousterhout *et al.* 1985] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System." In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 15–24. ACM, December 1985.

[Palmer and Zdonik 1991] M. L. Palmer and Stanley B. Zdonik. "Fido: A Cache that Learns to Fetch." Technical Report CS-91-15, Brown University Department of Computer Science, February 1991.

[Patterson *et al.* 1995] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. "Informed Prefetching and Caching." In *Proceedings of the 15th Symposium on Operating Systems Principles*, pp. 79–95, Copper Mountain Resort, Colorado, December 1995. ACM.

[Pawlowski *et al.* 1994] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. "NFS Version 3 Design and Implementation." In *USENIX Conference Proceedings*, pp. 137–152. USENIX, June 1994.

[Peterson 1991] Anthony L. Peterson. "E-Systems Modular Automated Storage System (EMASS) Software Functionality." In Karen D. Friedman and Bernard T. O'Lear, editors, *Proceedings of the Eleventh IEEE Symposium on Mass Storage Systems*, pp. 73–76, Monterey, California, October 1991. IEEE, IEEE Computer Society Press.

[Popek 1996] Gerald J. Popek, 1996. Personal communication.

[Popek and Walker 1985] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture*. The MIT Press, 1985.

[Ratner 1995] David H. Ratner. "Selective Replication: Fine-Grain Control of Replicated Files." Technical Report CSD-950007, University of California, Los Angeles, March 1995. Master's Thesis.

[Reiher 1995] Peter Reiher, 1995. Personal communication.

[Reiher *et al.* 1994] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. "Resolving File Conflicts in the Ficus File System." In *USENIX Conference Proceedings*, pp. 183–195. University of California, Los Angeles, USENIX, June 1994.

[Reiher *et al.* 1996] Peter Reiher, Jerry Popek, Michial Gunter, John Salomone, and David Ratner. "Peer-to-peer Reconciliation Based Replication for Mobile Computers." In *Proceedings of ECOOP'96 II Workshop on Mobility and Replication*, July 1996.

[Salem *et al.* 1992] Kenneth Salem, Daniel Barbará, and Richard J. Lipton. "Probabilistic Diagnosis of Hot Spots." In *8th International Conference on Data Engineering*, pp. 30–39. IEEE, February 1992.

[Satyanarayanan 1996] Mahadev Satyanarayanan. "Mobile Information Access." *IEEE Personal Communications Magazine*, **3**(1):26–33, February 1996.

[Satyanarayanan *et al.* 1990] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. "Coda: A Highly Available File System for a Distributed Workstation Environment." *IEEE Transactions on Computers*, **39**(4):447–459, April 1990.

[Satyanarayanan *et al.* 1993] Mahadev Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu. "Experience with Disconnected Operation in a Mobile Computing Environment." In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pp. 11–28, Cambridge, MA, August 1993. USENIX.

[Schroeder *et al.* 1985] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. "A Caching File System for a Programmer's Workstation." In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 25–34. ACM, December 1985.

[Skopp and Kaiser 1993] Peter D. Skopp and Gail E. Kaiser. "Disconnected Operation in a Multi-User Software Development Environment." In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, Princeton, NJ, October 1993. IEEE.

[Smith 1981] Alan J. Smith. "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms." *IEEE Transactions on Software Engineering*, **7**(4), July 1981.

[Sokal 1977] Robert R. Sokal. "Clustering and Classification: Background and Current Directions." In J. Van Ryzin, editor, *Classification and Clustering*, pp. 1–15. Academic Press, 1977. Proceedings of an Advanced Seminar Conducted by the University of Wisconsin at Madison, May 3–5, 1976.

[Späth 1980] Helmuth Späth. *Cluster Analysis Algorithms.* Ellis Horwood, Ltd., Chichester, England, 1980.

[Staelin and Garcia-Molina 1990] Carl Staelin and Hector Garcia-Molina. "File System Design using Large Memories." In *Proceedings of the Fifth Jerusalem Conference on Information Technology*, pp. 11–21. IEEE, 1990.

[Steere and Satyanarayanan 1994] David C. Steere and Mahadev Satyanarayanan. "A Case for Dynamic Sets in Operating Systems." Technical Report CMU-CS-94-216, Carnegie-Mellon University, November 1994.

[Tait 1997] Carl Tait, 1997. Personal communication by e-mail.

[Tait and Duchamp 1991] Carl D. Tait and Dan Duchamp. "Detection and Exploitation of File Working Sets." In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pp. 2–9, 1991.

[Tait *et al.* 1995] Carl D. Tait, Hui Lei, Swarup Acharya, and Henry Chang. "Intelligent File Hoarding for Mobile Computers." In *Proceedings of the MobiCom '95: The First International Conference on Mobile Computing and Networking*, pp. 119–125, Berkeley, CA, November 1995.

[Tichy 1982] W. F. Tichy. "Design, Implementation, and Evaluation of a Revision Control System." In *Proceedings of the Sixth International Conference on Software Engineering*, September 1982.

[Vitter and Krishnan 1996] Jeffrey Scott Vitter and P. Krishnan. "Optimal Prefetching via Data Compression." *Journal of the ACM*, **43**(5):771–793, September 1996.

[Wall and Schwartz 1991] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Sebastopol, CA, January 1991.

[Zupan 1982] Jure Zupan. *Clustering of Large Data Sets.* Research Studies Press, 1982.