Automated Hoarding for Mobile Computers*

Geoffrey H. Kuenning and Gerald J. Popek†

Abstract

A common problem facing mobile computing is *disconnected operation*, or computing in the absence of a network. *Hoarding* eases disconnected operation by selecting a subset of the user's files for local storage. We describe a hoarding system that can operate without user intervention, by observing user activity and predicting future needs. The system calculates a new measure, *semantic distance*, between individual files, and uses this to feed a clustering algorithm that chooses which files should be hoarded. A separate replication system manages the actual transport of data; any of a number of replication systems may be used. We discuss practical problems encountered in the real world and present usage statistics showing that our system outperforms previous approaches by factors that can exceed 10:1.

1 Introduction

The face of computing today is rapidly being changed by the advent of mobility, but the utility of the portable computer is seriously challenged by the problem of *disconnected operation*, where useful work must continue in the absence or near-absence (*i.e.*, available only at high cost or low bandwidth) of the network. Although impressive resources are being devoted to research in wireless networking, with a goal of making communication continuously available, the problem is difficult, and it is likely to be a long time before the mobile user will have the same networking capabilities as we expect from a stationary computer today. In the interim, portable computers will often find themselves either completely lacking communication or significantly restricted by battery power, bandwidth, or cost.

*This work was partially supported by the Defense Advanced Research Projects Agency under contract N00174-91-C-0107.

†The authors are affiliated with the Computer Science Department, University of California, Los Angeles. Gerald Popek is also affiliated with Platinum *technology*. E-mail: geoff@fmg.cs.ucla.edu, popek@platinum.com.

Published in the 16th ACM Symposium on Operating Systems Principles, October, 1997, St. Malo, France.

In the absence of readily available high-quality communication, users are often forced to operate disconnected from the network. But in a world dominated by networking, this is a major drawback, because the computing paradigm has grown dependent on the availability of non-local resources. Lack of access to a remote file can halt work on a particular task or even make the computer unusable.

A very attractive solution to the lack of communication is *hoarding*, in which non-local files are cached on the local disk prior to disconnection. The local files can be managed and kept consistent by a replication system [7, 9, 11].

The difficult challenge is the "hoarding problem" of selecting *which* files should be stored locally. Earlier solutions have simply chosen the most recently referenced files [1, 9] or asked the user to participate at least peripherally in managing hoard contents [11, 21]. The former approach is wasteful of scarce hoard space, while the latter requires more expertise and involvement that most users are willing to offer.

We have taken a fresh approach to this problem, and have succeeded in creating a predictive hoarding system, called SEER, that makes hoarding decisions without user interaction. SEER considers the user's activities to be composed of projects, rather than individual files, which greatly enhances the accuracy of its predictions. In daily use, the system has dramatically improved the achievable quality of hoarding decisions, in general requiring a hoard that is only slightly larger than the working set.

2 System Overview

Automated predictive hoarding is based on the idea that a system can observe user behavior, make inferences about the semantic relationships between files, and use those inferences to aid the user. SEER consists of two major components built atop a replication substrate. First, an *observer* watches the user's behavior and file accesses, classifying each access according to type, converting pathnames to absolute format, and feeding the results to a *correlator*. The correlator evaluates the file references, calculating the *semantic distances* among various files (see Section 3.1). These semantic distances drive a clustering algorithm (Section 3.3.2) that assigns each file to one or more projects.

When new hoard contents are to be chosen, the correlator examines the projects to find those that are currently active, and selects the highest-priority projects until the maximum hoard size is reached. Only complete projects are hoarded, under the assumption that partial projects are not sufficient to make progress.

SEER does not itself do the file hoarding; instead an underlying replication system performs this task. This design frees SEER from the troublesome details of moving files back and forth between computers, making sure updates are propagated to other replicas of the files, and managing conflicts [17]. It also makes SEER more portable because very little is assumed about the underlying system. SEER currently runs atop the RUMOR [6, 18] user-level replication system, a custom-built master-slave replication service called CHEAP RUMOR, and CODA [11], and it could easily be used with other systems such as FICUS [7]and LITTLE WORK [9].

A feature critical to usability is that, unlike previous systems, SEER normally operates without user intervention. There is no need to build explicit lists of important files or to instruct the system that certain activities are of interest. The only user interaction (beyond any that might be required by the underlying replication system) involves informing the computer that a disconnection is imminent, and even this requirement can be eliminated by automated periodic hoard filling if desired. Although SEER allows users to provide explicit hoarding instructions, our experience shows that such intervention is rarely necessary.

3 Underlying Concepts

The fundamental assumption of SEER is that there is *semantic locality* in user behavior. By detecting and exploiting this locality, a system can make inferences about the relationships between various files. Once these relationships are known, there is potential for an automated hoarding system to perform much better than one that is based on LRU-style caching algorithms.

3.1 Semantic Distance

To detect semantic locality, SEER defines a new concept known as *semantic distance*. Conceptually, semantic distance attempts to quantify a user's intuition about the relationship between files. A low semantic distance suggests that the files are closely related and thus are probably involved in the same project, while a large value indicates relative independence and different projects.

In our system, semantic distance is based on measurements of individual file *references*, rather than looking at the files themselves. The distance between references is then summarized (Section 3.1.2) to produce a value that is relevant to the individual files.

In our system, a file reference is considered to be a high-level operation, such as an open or status inquiry. We do not track individual reads and writes, partly for efficiency but primarily because we believe that doing so would obscure the information we are trying to extract. SEER is interested in whole files, rather than individual bytes, so it is more informative to look at whole-file operations.

3.1.1 Measuring Semantic Distance

While the concept of semantic distance is simple, it is not so easy to come up with a quantification that is both meaningful and implementable. The method we have chosen is based on the observation that semantic locality is similar to temporal locality: files that are referenced at the same time tend to be semantically related. This observation is not original to us [4, 5, 12, 21], but to our knowledge we are the first to formalize the notion of semantic locality and its relationship to temporal locality.

This leads directly to a first definition of semantic distance (note that all of our suggested measures are asymmetric):

Definition 1 ***Temporal semantic distance.*** *The temporal semantic distance between two file references is equal to the elapsed clock time between the references.*

This definition has intuitive appeal: it is simple and easy to measure, and it nicely captures the fact that files referenced at the same time tend to be semantically related. Unfortunately, it has a basic flaw, which is the fundamental disparity between computer and human time scales. For example, during a compilation, object files would be considered related to their respective sources, but two source files that were components of the same program would be less closely related because accesses to them during editing may be separated by many minutes. Also, the definition is subject to artificial distortion due to anomalies such as telephone interruptions or large variations in system load.

To avoid these difficulties, we can modify our definition to use the sequence of file references, without regard to clock time:

Definition 2 ***Sequence-based semantic distance.*** *The sequence-based semantic distance between two file references is equal to the number of intervening references to other files.*[1]

This definition allows us to infer semantic relationships from temporal locality without suffering distortions due to time-scale anomalies. However, Definition 2 still needs improvement. If we consider only whole-file references such as opens, an individual reference does not take place at a particular point in time. Instead, a file reference can then be considered to have a *lifetime* reaching from an open to a corresponding close. Our experiences suggest that it is the relationship

[1] In practice, there are several alternative ways of implementing this definition. For example, in the sequence $\{A, A, \ldots, B\}$, SEER uses only the closest pair of references in calculating the distance from A to B. Similarly, in the sequence $\{A, C, C, C, B\}$, a strict interpretation of the definition would result in a semantic distance of 3, which is the choice used by SEER. However, it might be equally sensible to elide the repeated references, so that the distance was only 1. We chose not to do this partly for efficiency, and partly to capture the phenomenon of intensive work on a single project. The various options involved in calculating semantic distance are discussed in detail in [15].

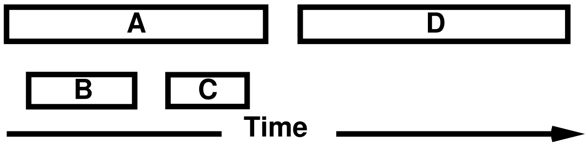

Figure 1: Sample file access sequence.

between these lifetimes, rather than the individual point-in-time references, that is of semantic importance.

For example, consider the compilation of a C module that is composed of a source file S and several included header files $H_1, H_2, \ldots, H_n$. The header files will be opened and closed in sequence, yet the n^{th} header file is just as essential to compiling the program as the first. To capture this important relationship, we can define a measure based on file lifetimes by taking advantage of the fact that S remains open during the entire process:

Definition 3 ***Lifetime semantic distance.*** *The lifetime semantic distance between an open of file A and an open of file B is defined as 0 if A has not been closed before B is opened, and the number of intervening file opens (including the open of B) otherwise.*

For example, consider the reference sequence $\{A^o, B^o, B^c, C^o, C^c, A^c, D^o, D^c\}$, where the superscripts o and c indicate opens and closes respectively. This sequence is diagrammed in Figure 1, where the extent of an access is indicated by the width of the enclosing box. The lifetime-based semantic distance from A^o to each of B^o and C^o will be 0, while the distance from A^o to D^o will be 3. Similarly, the distances $B^o \rightarrow C^o$, $B^o \rightarrow D^o$, and $C^o \rightarrow D^o$ will be 1, 2, and 1, respectively. All other distances ($B^o \rightarrow A^o$, $C^o \rightarrow A^o$, $D^o \rightarrow A^o$, $C^o \rightarrow B^o$, $D^o \rightarrow B^o$, and $D^o \rightarrow C^o$) are undefined in this brief example.

Finally, we need to consider file references other than opens or closes. For example, a file rename may be an essential part of a compilation and thus as semantically meaningful (in terms of hinting at file relationships) as an open. For most purposes, SEER treats such references as if they were an open followed immediately by a close. We discuss these other types of references in more detail in Section 4.

3.1.2 Data Reduction

Semantic distance is calculated between two file-reference events (normally file opens). For SEER's purposes, however, the more interesting information is the semantic relationship between two files, rather than between two references to those files. Tracking files instead of references reduces the amount of data that must be stored, but brings up the issue of how to convert the multiple distances between events into a single distance between files.

The most obvious conversion method is to use a simple mathematical summary, such as the arithmetic mean, to represent the entire sequence of references. The particular summary chosen should be easy to calculate, updatable on-line, small in storage requirements, and defensible as being a reasonable representation of the actual semantic relationship between files.

The arithmetic mean, attractive for its simplicity, satisfies all of these requirements, and was the first method we tried. However, we found that the arithmetic mean produced undesirable results. For example, if three event pairs produce distances of 1, 1, and 1498, the arithmetic mean would be 500. But the user would very likely consider the files involved to be more closely related than two other files represented by semantic distances 500, 500, and 500. The problem is that small numbers are much more indicative of a relationship than are large ones. Because of this disparity in significance, we turned to the geometric mean, which gives smaller values more importance.

3.1.3 Practical Algorithms

Definitions 1 through 3 in Section 3.1.1 have the common characteristic that they define a distance value between every pair of files mentioned in a stream of references. Since SEER is designed to process data from months or even years of references, encompassing tens or hundreds of thousands of files, the $O(N^2)$ storage complexity required to keep track of the distance between every pair of files becomes prohibitive. Furthermore, each new reference to a given file generates new distances between it and all previously-referenced files, so that the cost of processing a single reference online is $O(N)$ in the number of files, which is also unacceptably high. Even if a reference could be processed in 1 μs, keeping track of all pairs would expend 10 ms of CPU time per open if only 10,000 files were known. This is even higher than the base CPU cost of an open in a modern system, and 10,000 files is a very small number for a modern distributed system.

Fortunately, since we are interested in locating files that are semantically close to each other, it is not necessary to store all N^2 distances. Instead, SEER uses an approximation heuristic to calculate semantic distances. The heuristic makes two compromises for the sake of efficiency. First, rather than storing the distance between every pair of files, only n distances ($n = 20$ in our current implementation; see Section 4.9 for more information on how the algorithm's constants were chosen) to a file's closest neighbors are tracked. Second, when processing a new file reference, the distances updated are limited to those from files that are within a distance of M (currently $M = 100$) of the current reference. Although these heuristics can introduce a large error in pathological cases [15], in practice they have produced acceptable results. A compensation algorithm detects and partially adjusts for larger distances by inserting M whenever a value larger than M would have occurred.

From time to time, it is necessary to replace one of the n distances kept for each file (*i.e.*, when a new semantic distance arrives with a small value). In this case, a priority system is used. The highest priority goes to a closely related file that is marked for deletion from the internal table. If no such file exists, the list of n references is scanned to locate the one with the largest current semantic distance (with ties broken randomly). If this reference has a distance larger than that of the new candidate, it is chosen for replacement. Finally, if there is still no candidate, an aging system is applied that allows very old and inactive references to be replaced by newer ones; details are given elsewhere [15]. This aging system is necessary to allow SEER to track fundamental changes in user behavior and to allow incorrectly inferred relationships to be removed over time.

3.2 Other Distance Measures

Besides semantic distance, there is a wealth of other information that can be gleaned from a running computer system to help an automated hoarding system achieve acceptable results. That information includes:

Directory membership. As a general rule, files in the same directory are more closely related to each other than files in different directories.

File naming conventions. Naming often provides clues to important relationships. For example, C++ classes are often described in header files and implemented in source files that differ only in the extension.

"Hot" links. The Object Linking and Embedding facility in WINDOWS® (OLE) allows documents, graphs, and other objects to be interlinked as necessary to build larger structures in a flexible manner. These so-called hot links provide valuable and low-cost information about fundamental relationships among members of a project. A programming-language analog is the `#include` statement in C and C++, which also indicates a very strong inter-file relationship.

To take advantage of directory membership, SEER incorporates a directory-distance measure that is zero for files in the same directory and increases for files in more widely-separated directories.

To handle the other two types of relationships, SEER provides a generalized *external investigator* mechanism. An external investigator is an auxiliary program that can examine selected files and extract application-specific information, which is then supplied to the correlator as extra file relationship data. For example, we have developed a simple script that can read C source files to discover `#include` relationships that are then passed to the correlator for inclusion in the clustering decision. The information is expressed as groups of related files, together with an investigator-chosen weight indicating the strength of the relation. The clustering algorithm discussed in Section 3.3.2 makes use of these relationships when specified, although it does not require them. The method of integrating these relationships is described in Section 3.3.3.

If an external investigator can identify an entire project, this information can be communicated to SEER independently of the internal clustering algorithm. For example, a `makefile` investigator could potentially identify every file needed to build a particular program and create a cluster containing exactly these files.

3.3 Clustering Algorithm

Simply knowing the relationships among individual files solves only half the problem of predictive hoarding. These pairwise relationships must be converted into meaningful groupings of files into projects. To do so, we use a multidimensional clustering algorithm.

3.3.1 Requirements

Although clustering has been widely studied, relatively few known clustering algorithms are appropriate for the problem at hand. In particular, SEER needs the following characteristics:

Efficiency. SEER must cluster many thousands of files, so algorithms that require exponential time or $O(N^2)$ storage are not practical.[2] Since clustering must be done shortly before disconnection, the algorithm must take only seconds, or at worst a few minutes.

Partial Information. Because of space limitations, SEER does not store the distance between every pair of files, and there is no way to calculate this distance from the information that is kept, so the algorithm must be able to make its decisions based on limited data.

No Distance Metric. Although we call semantic distance a "distance measure," it is not a distance metric as required by many clustering algorithms [3]. In particular, it is asymmetric and does not satisfy the triangle inequality.

Overlapping Clusters. Perhaps the most troublesome characteristic of the problem is the need for files to be members of more than one cluster. A compiler, for example, may be used to compile programs for a number of different projects, and so should be a member of more than one cluster. Relatively few clustering algorithms allow points to be members of multiple clusters simultaneously; in fact, most algorithms assume that this characteristic would be undesirable.

No Objective Criterion. There is no numerical measure that can be used to characterize the "goodness" of a particular cluster assignment, eliminating algorithms that seek to optimize such a criterion.

[2] Optimal clustering is NP-hard [16].

Relationship	Action
$k_n \le x$	Clusters combined into one
$k_f \le x < k_n$	Files inserted, but clusters not combined
$x < k_f$	No action

Table 1: Summary of clustering algorithm (x is number of shared neighbors).

3.3.2 Agglomerative Algorithm

The algorithm we have developed is based on one originated by Jarvis and Patrick [10]. This algorithm is bottom-up, or *agglomerative*, starting with each data point assigned to an individual cluster and then combining clusters according to a shared-neighbors criterion. In the original formulation, the algorithm first calculates the n nearest neighbors to each point, where n is a parameter of the algorithm. After the n nearest neighbors of each point have been calculated, the Jarvis and Patrick algorithm compares the nearest-neighbor list for each pair of points. If two points have more than k of their n nearest neighbors in common, they are considered to be members of the same cluster, and their clusters are combined. The storage requirements are thus $O(N)$, while the time complexity is $O(N^2)$ since each point must be compared to every other point.

In our variation, we achieve $O(N)$ time complexity by avoiding the comparison of every possible pair of points to locate nearest neighbors. Instead, we use the existing table of n nearby files calculated by our semantic-distance heuristic. In addition, we use two thresholds, k_n (near) and k_f (far), where $k_n > k_f$.[3]

If two files share at least k_n neighbors, their clusters are combined into one, as in the Jarvis and Patrick algorithm. However, if the files share fewer than k_n but at least k_f neighbors, their clusters are not combined, but instead are overlapped. In the overlapping operation, each of the closely-related files is added to the other file's containing cluster. These options are summarized in Table 1, where x represents the number of shared neighbors.

For example, consider seven files, A, B, C, D, E, F, and G. The number of shared neighbors between each pair of these seven files is given in Table 2. In the table, a blank entry indicates that the file heading the row does not list the paired file as related; thus, even if they share neighbors, the clustering algorithm will not discover this fact. For simplicity, we list the other distances in terms of the thresholds: 0, k_f, or k_n. Thus, for example, file A lists B as a neighbor and shares k_n neighbors with it. A also lists C as related, but the two files share only k_f neighbors. None of the other four files are mentioned in A's relation list, so the algorithm will have no knowledge of neighbors shared with them.

[3] The idea of "near" exceeding "far" may seem counterintuitive, but is necessary because smaller thresholds are more lenient, so that the lower value of k_f allows more-distant relationships to be discovered.

	To:						
From:	A	B	C	D	E	F	G
A		k_n	k_f				
B			k_n				
C				k_f			
D					k_n		
E							
F							k_n
G				k_n			

Table 2: Example relationships among seven files.

In the first phase, our algorithm looks for files that share at least k_n neighbors, and combines their clusters. In our example, files A and B share k_n neighbors, so they become a two-file cluster. No other files are closely related to A, so the algorithm moves on to B. Since this file shares k_n neighbors with C, C is added to B's cluster. This step also clusters A with C, even though there is no direct relationship between the two files. Since neither B nor C share k_n or more neighbors with any other files, no other files are added to this cluster.

Continuing with files D through F, the same criteria are applied to combine D and E into one cluster, and to combine F and G into a second. At this point there are three clusters: $\{A, B, C\}$, $\{D, E\}$, and $\{F, G\}$. File G is then processed; noting that it shares at least k_n neighbors with D, the clusters containing these two files are combined into a single four-member cluster, $\{D, E, F, G\}$. Phase one is now complete.

In the second phase, the algorithm re-processes all files, looking for pairs that share fewer than k_n but at least k_f neighbors. There are two such pairs, $\{A, C\}$ and $\{C, D\}$. Since A and C are already in the same cluster, no further action is taken. For C and D, the algorithm adds each of these files to its counterpart's cluster, but does not combine the entire clusters. Thus, the final clusters are $\{A, B, C, D\}$ and $\{C, D, E, F, G\}$.

3.3.3 Incorporating Additional Information

The algorithm discussed in Section 3.3.2 is simple and effective, but does not support the additional distance measures discussed in Section 3.2. In the Jarvis and Patrick formulation, multiple measures could be handled by calculating the Euclidean distance between potential cluster measures. However, this calculation would require that all measures be available between all file pairs, which is not possible with the possibly limited information provided by external investigators. Thus, SEER uses a more *ad hoc* approach.

When extra information is available, the shared-neighbor count is incremented or decremented by the value of the additional information, optionally weighted by an administrator-chosen amount. For example, since a large directory distance (as defined in Section 3.2) tends to indicate a looser relationship, the directory distance is subtracted from the shared-neighbor count, causing widely-separated files to be less likely to cluster together. Conversely, an investigated relationship is additional evidence of closeness between files, so the strength of the relation as provided by the investigator is added to the shared-neighbor count to increase the likelihood of clustering.

Since it is the shared-neighbor count that is modified, the additional information does not modify the semantic distance, instead acting more directly on the clustering algorithm. This allows the tendency of two files to cluster together to be either enhanced or reduced, and also sidesteps the difficulties introduced by the asymmetry of semantic distance. In addition, modifying the shared-neighbor count allows the extra information to be given greater importance, which is appropriate because external investigators can use their application-specific knowledge to achieve more accuracy than is available through the more general-purpose algorithms of semantic distance.

An important point is that the investigated relationships are tested regardless of whether SEER has independently stored a semantic distance between the files. By setting the strength of a relation sufficiently high, an external investigator can force two or more files to be clustered together independently of other factors, so automated investigators can override the clustering algorithm if they choose.

4 Real-World Intrusions

The previous sections have presented an elegant framework for the design of 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.

4.1 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 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 inter-file relationships. In addition, because `find` accesses every file, it destroys any LRU history that might have been useful in hoarding decisions. This problem is even more severe in LRU-based systems such as CODA and LITTLE WORK.

As we gained experience with SEER, we learned that there were many programs with similar behavior, and we spent a considerable amount of time searching for the best solution to the problem. Approaches we 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 it as "meaningless").

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

3. Detect directory opens, and mark a process meaningless only while the directory is open.

4. 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 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 is almost as simple, but failed in practice because many meaningful programs read directories. For example, many text editors do so to implement filename completion.

The third solution 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, this assumption turned out to be false, so that this solution, too, fails in practice.

The fourth method, though more complex, has proven successful. Each time a process opens a directory, SEER counts 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. So, for example, `find` will tend to have a history of accessing every possible file, and thus would get marked as being a meaningless process, while an editor will (on average) access far fewer than the maximum and will remain meaningful.

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 `getcwd`'s behavior pattern and temporarily marks the process as being inside this function. During this period, all file references are ignored, even for purposes of inferring meaninglessness).

These heuristics have made it possible for SEER to make the right decision about the relevance of a process' references in most cases. However, we have retained the ability to hand-specify a few processes as being meaningless.[4] As in information retrieval, it is necessary to filter out certain irrelevant relationships, and as in that field, the current mechanisms are inelegant and could benefit from further refinement.

4.2 Shared Libraries

Certain files on a modern computer are so fundamental that nearly every program uses them. The most common example, though hardly the only one, is the shared library.

Shared libraries present a serious problem for a system that tries to infer inter-file relationships from the sequence of opens. Since every program's reference sequence includes the shared library, the library becomes a common link between otherwise unrelated files. For example, if S is the shared library, SEER might observe the sequences A, S, X and B, S, Y. S appears to be related to both X and Y, even though they are actually members of unrelated sequences. This eventually causes the clustering algorithm to combine all files into 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%) of all 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 important, it is always included in the hoard regardless of its last reference time. On the machine with the largest frequent-file list, 8 files fall into this class, representing 2.3 MB of disk space, or about 5% of that user's 50-MB hoard.

4.3 Critical Files

Every system has some files that are essential to system operation, such as files used in the bootstrap process or for personal startup and configuration. 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. The phenomenon of rare access to critical files is a fundamental problem with any completely automated hoarding system.

SEER addresses the problem in two ways. First, a system control file 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 applies a similar exclusion to any file whose name begins with a period (*e.g.*, `.login`). We have found that such files tend to be relatively small compared to the total hoard size, and that they usually contain important control and configuration information that the user cannot do without.

Although it is possible for the user to modify the system control file to list other files that he considers critical to successful operation, this has not been necessary in practice. Out of nine SEER users in our initial deployment, only one even learned how to list special files, and this was to correct an oversight by the system administrator. Nevertheless, we are unhappy with the necessity for explicit specification and plan to seek alternatives in our future research.

4.4 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 the hoard miss. This capability is important for two reasons. First, SEER needs to know of the miss so that it can add the file (and all other members of its project) to the hoard for future use. Second, because hoard misses are often devastating to the user, causing a change in the work being done, they provide the best statistics for measuring the success of SEER (see Section 5.1) and tuning the algorithms.

Depending on the underlying replication system, 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 a remote access 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, the error code, and the state of SEER's internal tables) 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 are "implied," occurring without a direct attempt to access the file. For example, a user might ask for a directory listing, note that the file is missing, and never attempt to open it. Again, this is dependent on the replication system, but because SEER is portable, it must deal with the possibility.

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 cannot access a file, he runs a simple program to record the miss in a log file and arrange for it to be hoarded in the future. This is a violation of our no-user-burden design, but is forced upon us by deficiencies in some replication systems. For research purposes, the program also records the time and date of the miss and a user-specified severity code, as follows:

0 The lack of the file has made the entire computer unusable, *e.g.*, a critical startup file is unavailable. In this case the miss cannot be recorded until a network connection is re-established.

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.

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

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 preloaded so that the file will be available in the future.

This manual recording of misses is subject to the vagaries of user behavior, since it is possible that a user might neglect to record a miss and thus perturb the statistics collected. It is for this reason that we designed the system so that the same user action both records the miss and arranges for the file to be hoarded at the next reconnection. By combining the gathering of statistics with a function necessary to the user, we were able to ensure that misses would not go unrecorded. In addition, regular personal interaction with users in our small-office environment allowed us to independently verify the low failure rate.[5]

As a backup to the manual miss reporting, SEER also includes an automated miss-detection system that notes when a user attempts to access a file that is known to exist but is absent from the hoard. This mechanism will sometimes detect misses that a user would consider unimportant, and it cannot detect "implied" misses, but it is still a useful feature.

4.5 Temporary Files and Directories

Many programs create temporary files to hold transient results. Because of their transient nature, semantic relationships between these files and more permanent ones 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.

The current implementation of SEER allows certain directories, *e.g.* `/tmp`, 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. Similar pattern-based detection methods could be used in other operating systems.

It would be much more elegant to detect temporary files automatically, but the current design cannot accomplish this because by the time a file can be recognized as temporary, it has already had the opportunity to displace more important files in the list of n related files that is kept for each file (see Section 3.1.3). We plan to pursue automated algorithms in the future.

[5] In early testing before statistical collection was began, the first machine deployed did experience a single severity-0 failure due to the lack of `.cshrc`; it was this failure that led us to install the UNIX-specific heuristic discussed in Section 4.3.

4.6 Non-Files

The LINUX filesystem supports a number of objects besides files, including directories, symbolic links, and more exotic objects such as device files and pseudo-filesystems. 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, or even to receive a login prompt.

With the exception of directories and possibly some pseudo-filesystems, these objects take almost no disk space. Because of the importance of these objects and their minimal space requirements, SEER always includes them in the hoard. Many of these objects are also omitted from semantic-distance and clustering calculations, since they often vary depending on extraneous factors (*e.g.*, `/dev/tty`*xx*). A control file, set up by the system administrator, specifies which objects are ignored.

Directories are the only objects that regularly require significant disk space. However, the underlying replication system may have its own needs regarding directories (for example, RUMOR might choose to store a directory so that its contained objects are accessible when disconnected). For this reason, SEER leaves hoarding decisions regarding directories up to the replication substrate. For space calculations, however, it makes the conservative assumption that all directories are hoarded.

4.7 Simultaneous Accesses

The formulations of semantic distance in Section 3.1 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 instance by reading e-mail while waiting for a compilation. These independent streams are intermixed when observed by SEER, and create incorrect and spurious file relationships if not properly handled.

We had originally hypothesized [13] that the data reductions discussed in Section 3.1.2 would provide a noise-filtering mechanism that would 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 [21]. SEER maintains a separate reference-history list for each process, and calculates semantic distances on a process-local basis. The file-open test mentioned in Definition 3 is also performed on a per-process basis. Reference histories are inherited from parent processes and merged when children exit, allowing SEER to detect extended relationships between files referenced by a process and by its parent.

4.8 Non-Open References

A real program can refer to a file in a variety of ways. Besides being opened and closed, 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. A few require more complex treatment:

Process Lifetimes. Executions and terminations of processes are treated as opens and closes, respectively.

File Deletion. Because many programs delete and immediately recreate files, SEER delays removal from its internal tables for a short period (measured in terms of total deletions) so valuable relationship information won't be lost if the file name is immediately reused.

Attribute Examination. Many programs examine file attributes to see whether a file exists or to discover whether it can be written. Usually, the file will be subsequently opened. It would be less correct to record this activity as two references to the file, since from the user's point of view there is only a single access. 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.

In general, SEER treats examination of an attribute as a simultaneous open/close pair. However, an examination immediately followed by an open is discarded as insignificant. In addition, certain more complex heuristics, discussed in Section 4.1, are applied in some cases.

4.9 Parameter Settings

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. Although space precludes a detailed discussion, we found it necessary to devote significant effort to searching the parameter space for the values that would produce good results for all users. The search methods and the parameters we used for our tests are detailed in [15].

4.10 Avoiding Deadlock

Since SEER issues its own system calls, deadlock can occur if these calls are themselves traced. 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), and deadlock can occur due to calls made by these processes. We solved this problem by not tracing most calls made by the superuser ("`root`"). This prevents SEER from being able to manage certain files needed by superuser activities (*e.g.*, programs invoked by `cron`). We are investigating alternative methods that will allow use to trace superuser operations and still avoid deadlock.

4.11 Tracing System Calls

We indicated in Section 2 that the observer watches the user's file accesses. Observation is implemented with a simple modification to the operating system kernel that allows system calls to be traced. In general, calls are traced after they complete so that SEER can observe their success or failure status. However, a few calls (on LINUX, only `exec` and `exit`) are traced before execution to capture important information that will be destroyed when the call completes.

5 Evaluating Success

5.1 Measurement Methodology

As discussed in [11], traditional measures of cache performance, such as miss rate, are inappropriate in a hoarding situation. In a traditional caching system, a miss causes a relatively minor performance penalty, and has no effect on the overall course of the computation. By contrast, a miss in a hoarding system is a very severe event, because there is usually no way to service the miss at a small cost in performance. Instead, a hoard miss generally causes the user to stop work on the current task and switch to a secondary one. In a trace-driven simulation, a hoard miss invalidates the trace because of this task-switching behavior.

5.1.1 Time to First Miss

An alternative measure, first suggested in [20], is the time to the first hoard miss, measured as either elapsed time or number of file references. This is attractive because it quantifies the amount of work the user was able to do before a hoard failure forced a change in activity. We implemented this measure in our live experiments, including measuring the severity of the miss as discussed in Section 4.4.

Our experiments were conducted by deploying SEER on nine 486-based laptops used in a software development environment. Each laptop was associated with a single user and served as the primary platform for that person. The measurement period varied from one to eight months, with most machines being examined over a 3-month interval. Three of the machines (A, B, and E) were used only occasionally in disconnected mode, while the remainder each generated 75 or more active disconnection periods. The number of observed disconnections is reported in Table 3. Four machines (B, C, E, and H) were not used extensively during either connected or disconnected mode, primarily due to outside commitments or the use of alternative operating systems. Traces of user activity recorded a low of about 40,000 operations for the least-used machines (C and H) to a high of about 326,000,000 operations for the most heavily used (G).

To measure the time to first miss, we combined several tools. A background daemon periodically pings a well-known site to detect disconnection durations. The output of this daemon was post-processed to remove disconnections or reconnections of less than 15 minutes. This eliminated brief disconnections in which hoard misses would not be bothersome, and brief reconnections made to transfer e-mail or service an important hoard miss. (The latter can occur only after a miss is recorded, since the miss must be recorded before the system can know that it needs to be serviced. By discarding the reconnection and thus combining the adjacent disconnections, the total number of disconnections is reduced and the average disconnection time increased, both of which perturb our statistics in a direction detrimental to SEER.)

A second daemon detects suspension periods. This is important because laptop computers are often placed into a power-saving mode when no work is being done. It would be incorrect to report a 16-hour overnight disconnection if the laptop were only in active use for 2 hours; this is especially true when calculating the time to first miss. By discarding suspensions, we ensured that our statistics considered only times when the machine was being actively used. Disconnections during which the machine was completely unused (*e.g.*, vacations) were also excluded from the statistics.

Finally, misses themselves were measured using the manual and automated methods discussed in Section 4.4.

However, preliminary analysis revealed a severe flaw in this measure. The time to first miss is very dependent on the relationship of the chosen hoard size to the user's working set. A user with a small working set will almost never experience a miss, while one whose configured hoard size is only slightly larger than his working set will suffer an abnormally high failure rate. This flaw cannot be rectified retroactively, since any traces collected will have been affected by the presence or absence of hoard misses and thus cannot be reanalyzed assuming a different hoard size.

A secondary, though still important, drawback is that it is difficult to compare hoarding methods using this real-world measure. To properly compare two proposed algorithms, one should ask the user to perform the same tasks twice, once with a hoard filled by each algorithm. This is clearly impossible. The best one could do would be to ask a user to live with each algorithm for a period of time, or to ask two different users to use the algorithms in parallel. Either approach would introduce so much uncontrolled variation that dozens or hundreds of experiments would be necessary to eliminate uncertainty.

5.1.2 Miss-Free Hoard Size

For these reasons, we have invented a new measure that can be used to quantify the difference between hoarding algorithms. This is the *miss-free hoard size*, which is defined as the size a hoard would have to be to ensure no misses. For example, under a strictly LRU algorithm, the miss-free hoard size can be calculated as follows:

1. Sort all files according to their last reference time *prior* to the current disconnection period, with the most recent file first.

2. Mark each file that was referenced during the current period.

3. Locate the last marked file in the list.

4. Sum the sizes of all files between this file and the beginning of the list.

Clearly, if the hoard size is at least as large as the sum, an LRU hoarding would have included all files that were referenced in the disconnection period. A similar approach can be applied to any hoarding algorithm to calculate the hoard size that would be needed to avoid misses.

The miss-free hoard size offers several advantages over other measures:

- It quantifies the difference between algorithms in a linear, fine-grained fashion.

- It is not sensitive to current parameter settings.

- It can be calculated using reference traces, so that simulation becomes a viable analysis tool.

- It reflects the behavior that the user desires: working as if connected, with no awareness that only a selected subset of files is present.

We have carried out extensive trace-driven simulations to measure the miss-free hoard size under various conditions; the results are reported in Section 5.2.1. For each of a group of laptop computers, we collected file reference traces, in both connected and disconnected mode, over a period of one or more months.

The question of whether to use connected or disconnected traces was a difficult one. Traces of connected operation include behaviors that cannot happen while disconnected, such as browsing the Web. Traces of disconnected operation will at least occasionally include a period covering a hoard miss, which may change user behavior in some fashion. This will usually have the effect of placing more stress on the hoarding system, since the attempted access to the missed file will have been recorded in the trace, and the attention shift forced by the miss will now require the system to hoard both the old and the new projects. Even in the absence of misses, the user may have avoided some activity because he was aware that it was not hoarded.

Because of the complexity of these considerations, we chose to use complete traces, covering both disconnected and connected operation. Since our users experienced relatively few misses, and the majority were at insignificant severities, we believe that the disconnected traces were generally valid even when they covered periods with misses. The connected portions of the traces were included because we believe that although some activities might not occur disconnected, the general file-access patterns of these activities are still representative of typical applications and thus serve as a reasonable test of a hoarding system.

We then replayed the traces into the correlator running in a simulation mode. The simulation made use of actual file sizes whenever possible; when the size of a file was not available, the size was randomly assigned 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. To the extent that this distribution does not reflect actual file sizes, it will slightly distort the hoard-size and working-set statistics.

Each trace was replayed under up to four sets of conditions. We simulated disconnection durations of both 24 hours and 7 days, with each simulated disconnection separated by an infinitesimal reconnection during which the simulated user performed no work while the hoard was recomputed. On three machines (B, F, and G), we evaluated the impact of external investigators by simulating both with and without the information they supplied. For each combination of conditions, the simulation was repeated several times with different random seeds to reduce the variation introduced by randomly assigning file sizes. The individual experiments were done in random order to avoid possibly introducing outside trends.

Each simulation generated comparative results for SEER's cluster-based management scheme, a strict LRU scheme, and three schemes inspired by the formula used in CODA. However, the latter three schemes performed more poorly than LRU, due the lack of the ongoing hand management that they were designed to expect. (We did not have the resources to apply such hand tuning to our simulations.) Because these algorithms were not tested under conditions appropriate to their design, we chose not to report results for them.

5.2 Results

We now have approximately 35 man-months of experience using SEER in a live setting, with excellent results. We have also conducted extensive simulations. In live use, SEER has performed even better than expected; several users experienced no hoard misses at all, no one suffered a significant percentage of failed disconnections, and there were no severity-0 failures. In simulations, SEER's clustering algorithm essentially always outperforms LRU-style methods, and is usually so close to the optimum that we at first suspected an error in our measurement procedures.

Our only disappointment has been analytical, rather than experimental. The clusters produced by SEER often have contents that are surprising to us, either by including apparently unrelated files or by separating a single project into a few clusters rather than the single grouping that would correctly represent it. However, this problem can be mitigated by the use of external investigators. In any case, this discrepancy has not affected the success of our live and simulated experiments, so it is possible that it is only a theoretical difficulty that will never bother real users.

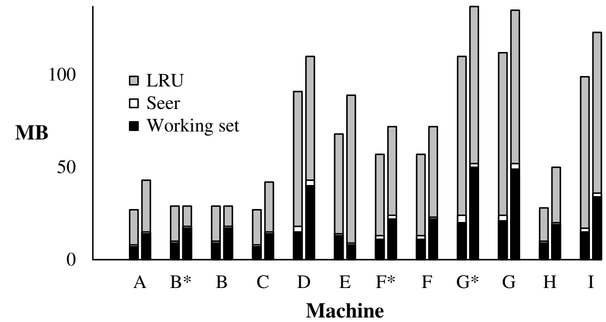

Figure 2: Mean working sets and miss-free hoard sizes for two managers. The left-hand bar of each pair represents daily disconnections, while the right-hand bar gives weekly values. Starred labels represent the use of external investigators.

5.2.1 Simulations

Figure 2 shows the miss-free hoard sizes graphically. Each pair of stacked bars represents a single machine; the left-hand bar of each pair is for daily disconnections and the right-hand bar is for weekly activity. For three machines (B, F, and G), the effect of using external investigators is shown by a bar pair marked with an asterisk. The lowest element of each stack represents the mean working set for the machine and period; the center element is the additional space required by SEER's clustering algorithm to remain miss-free, and the upper element shows the additional space needed by the LRU algorithm. 99% confidence intervals were within ± 2 MB about the mean for all measurements except the LRU hoard space, which always fell within $\pm$ 5 MB about the mean.

It is remarkable in Figure 2 that the clustering algorithm consistently requires space only slightly greater than the working set, which represents the needs of an optimum algorithm. By contrast, the LRU approach frequently uses space several times greater. This shows that SEER can be successfully used to hoard files in near-optimal space, so that power users who normally operate with nearly-full disks can work disconnected without inconvenience.

An interesting and unexpected result is that the external investigators did not make a significant difference in the required hoard size. In every case, the 99% confidence durations show that the use of external investigators had no statistically meaningful effect. In future research, we plan to examine this anomaly to see whether we can devise parameter settings that will make external investigation more useful.

Figure 3 gives another view of the same material. Here, instead of showing means, we give detailed data for a single machine and simulated disconnection period. This graph shows the weekly working set sizes for the most heavily-used

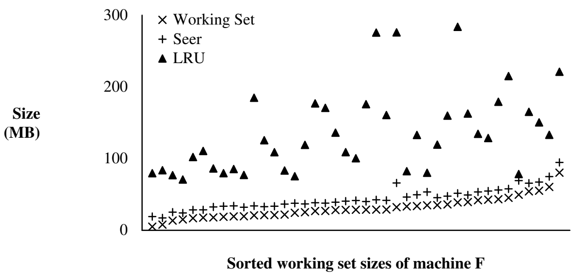

Figure 3: Performance of two hoard managers vs. working set sizes for simulated weekly disconnections of machine F (sorted by working set size; X axis represents sort order).

machine (F), plus the miss-free hoard size needed by the clustering and LRU managers for each week. To aid visualization, the X axis is sorted by working-set size. Each X value represents a particular week, but consecutive values do *not* represent consecutive weeks. Instead, the X values reflect the ordering of the disconnection periods after sorting. Again, we see that SEER's clustering manager requires a hoard size only slightly larger than the working set, while the LRU manager often requires significantly more space.

Another result of the simulations is that the working sets are relatively small. This is somewhat surprising in the face of the frequently made observation that disks tend to be full. This is an indication of the wastage on most systems: only a small fraction of all files are actually needed by the user on any given day. We also note that the advent of multimedia, voice recognition, and similar features can be expected to increase everyday disk requirements, placing added pressure on hoarding systems and making the superior performance of SEER even more important.

5.2.2 Live Usage

Table 3 gives statistics on the general disconnection behavior of actual users, including the number of observed disconnections (which reflects the machine's usage level during the measurement period), the mean ($\bar{x}$), median ($x_{0.5}$), and standard deviation (σ) of the disconnection duration, and maximum duration. (The minimum duration tends to be nearly constant approximately at 0.25 hours because of the 15-minute minimum disconnection time mentioned in Section 5.1.1.)

Table 4 summarizes statistics on failed disconnections, defined as those in which there was at least one hoard miss. For each machine, the table gives the hoard size used in megabytes, the absolute number of failures at each severity level, the number of failures at any severity, and the automatically detected failure count. To save space, all-zero rows have been omitted from Table 4. Two apparent anomalies in this table require further explanation. First, the number of failures at any severity can be smaller than the row sum if a particular disconnection experienced failures at multiple severities. Second, interviews with users and examination of

	Hoard	Failures						
User	Size	0	1	2	3	4	Any Sev.	Auto
A	50	0	0	0	0	0	0	2
C	50	0	0	0	0	0	0	1
D	50	0	0	0	0	0	0	5
E	50	0	0	0	0	0	0	1
F	50	0	3	6	11	9	24	2
G	98	0	0	0	0	0	0	3
I	50	0	1	0	0	0	1	5

Table 4: Summary of failed disconnections at various severities.

		Hours				
User	Sev.	$\bar{x}$	$x_{0.5}$	σ	Min	Max
A	Auto	1.8	—	2.3	0.21	3.4
C	Auto	1.6	—	—	1.6	1.6
D	Auto	0.9	1.0	0.5	≈ 0	1.3
E	Auto	11.0	—	—	11.0	11.0
F	1	10.6	—	16.3	≈ 0	29.4
	2	6.6	0.9	9.1	≈ 0	21.5
	3	3.4	0.5	4.9	0.1	12.9
	4	6.2	0.5	11.2	0.1	29.3
	Auto	20.4	—	28.4	0.3	40.5
G	Auto	0.5	—	0.3	0.2	0.8
I	1	1.0	—	—	1.0	1.0
	Auto	0.9	0.6	0.6	0.1	1.8

Table 5: Hours until first miss for failed disconnections.

traces have shown that automatically detected failures are not always failures from the user's point of view, which is why they tend to exceed the user-reported count.

Most users experienced very few failures. Only the most heavily used machine (F) suffered a significant number of failed disconnections (13% of the total disconnections), and the majority of those failures were at the unobtrusive severity levels 3 and 4. We should also emphasize that we deliberately chose unrealistically small hoard sizes to stress the system; in a real environment there would have been no failed disconnections at all. This reduced hoard size was the primary cause of the misses observed for machine F. Post-analysis of the data revealed that this machine's working set often exceeded 50 MB, so that no hoarding system could have performed miss-free with the configured hoard size. We have since increased the hoard size to 100 MB for this machine, and the miss rate is now comparable to that experienced by the other users.

Table 5 summarizes the time until the first miss, in hours, for the failed disconnections listed in Table 4. Here, the mean ($\bar{x}$), median ($x_{0.5}$), standard deviation (σ), and range are given. The median is omitted when there are fewer than 4 samples. This table also omits rows for all severity levels that had a zero miss count, and for machines that had no misses.

	Days	No. of	Disconnection Duration (Hours)				
User	Measured	Disconnections	Total	$\bar{x}$	$x_{0.5}$	σ	Max
A	111	38	424	11.16	3.24	15.82	71.89
B	79	10	431	43.20	0.57	127.19	404.94
C	113	75	745	9.94	1.12	40.87	348.20
D	118	90	271	3.01	1.38	4.46	26.50
E	71	25	47	1.87	0.81	2.54	12.08
F	252	184	1711	9.30	2.00	16.33	90.62
G	132	107	862	8.06	1.47	38.29	390.60
H	113	75	763	10.17	1.12	41.09	348.20
I	123	116	274	2.36	0.78	4.26	27.68

Table 3: Disconnection statistics.

Such rows would merely report the disconnection-time statistics for those machines; interested readers may refer to [15] for more information.

It is clear from these tables that the users of SEER did not suffer greatly due to hoard misses. Misses were rare, although when they did occur, they often occurred relatively soon after disconnection (as shown by the median values in Table 5). However, when these values are compared with the median disconnection times given in Table 3, we can see that misses generally occurred well into the disconnection, and that users normally continued to work after the miss occurred (shown both by the fact that the time to first miss is far less than 100% of the disconnection period and by the severity levels of the misses). It is also worth reiterating that no user experienced a severity-0 miss (computer unusable).

We also calculated the time-to-first-miss statistics across all disconnections, both successful and failed. Under these conditions, the time to first miss becomes essentially equal to the mean disconnection time. Again, this provides evidence to suggest that hoard misses were not bothersome to our users.

It is worth noting that intelligent user behavior is an important factor in the success of SEER. This same factor was previously observed with CODA [11, Section 5.2.2]. Before the advent of mobile computing, a traveling businessperson would load his briefcase with documents he expected to work on. While on an airplane, he would not attempt to work on a project that he knew was not in the briefcase. In a similar manner, users of SEER tend to be at least peripherally aware of the hoard contents, and do not attempt to work on projects that they know are unavailable. Instead, they plan ahead to some extent, devoting themselves to hoarded projects and later, while connected, attacking the unhoarded ones, which has the side effect of then causing them to become hoarded.

5.3 Implementation and Performance Impact

SEER is implemented primarily in C++, with a few auxiliary shell and PERL scripts that help to support external investigators and interface to the underlying replication system. All told, the system comprises approximately 47,500 lines of code.

The cost of running SEER is twofold: CPU and memory requirements. The CPU cost of tracking system calls is minor, about 35 μs on a 133-MHz PENTIUM® processor [15], and the system calls traced are infrequent ones such as `open`, making tracing inexpensive. Hoarding decisions are significantly more costly, requiring about 2 minutes of CPU time to form the clusters, but this is a relatively rare event that can be delayed until a chosen time, so our users have not found it troublesome.

The primary impact of SEER is in its memory usage, which was deliberately left unoptimized to simplify the research. The database of known files is stored in virtual memory, requiring about 1 KB for each of the approximately 20,000 files tracked on behalf of a typical user. However, we believe that a few straightforward improvements could cut this memory requirement by 50% or more. In addition, it would be relatively simple to modify the system to store the database on disk, rather than in virtual memory, since only a small fraction of the information is active at any given time. We postponed these optimizations because it was clear that they would not contribute directly to the research and could be added at a later date.

6 Related Work

There have been a number of previous systems supporting disconnected operation; however, it is difficult to compare them to SEER because no quantitative results have been published.

6.1 Early Systems

Disconnected operation was first developed in the early 1990's. Early systems used an LRU mechanism to load the hoard [1, 9], or left the problem to unspecified external mechanisms [8]. Some of these systems were actually used for disconnected operation, but no data on the performance of hoarding has ever been published. Our own experience suggests that LRU is usually an adequate approach, so that users would find these systems acceptable. It is only when an attention shift occurs that LRU fails significantly, because the user must individually reference each file involved in the shift.

This is in contrast to SEER's clustering approach, where an attention shift will quickly cause all members of a project to be loaded into the hoard.

6.2 CODA

The CODA system [11] enhanced simple LRU by allowing the user to specify an offset to be applied to the LRU age of a particular file, as a means of indicating its importance. A global bound arranged that for older files, the offset controlled the hoarding decision regardless of the original reference order. In practice, CODA users do not concern themselves with these details; instead they simply assign a "hoarding priority" to each file or group of files based on their perceived importance relative to other files.

When an attention shift occurred, users would change projects by loading a new set of priorities, called a "hoard profile," for that project. According to [20], separate hoard profiles were normally used for applications and data; a user would choose a subset of possible profiles depending on the expected activity. Hoard profiles for applications could potentially be created by a system administrator, but the user was burdened with both the specification of profiles for his own data and with the task of choosing the proper subset of profiles that would reflect the work he planned to do. Mahadev Satyanarayanan has commented [19] that this approach is similar to programming in assembly language: it provides excellent control over what happens, but is tedious and requires great expertise.

There are very few published results on the hoarding behavior of CODA. Although both [11] and [20] give quantitative information, the data presented relates to the size of working sets and the performance of the replication system. The only discussion of hoarding success is couched in general terms. For example, from [20, Section 5.2.2]:

> Many disconnected sessions experienced by our users, including many sections of extended duration, involved no cache misses whatsoever.

and

> 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.

6.3 SPY UTILITY

To date, the only other attempt to automate the hoarding process is Tait *et al.*'s SPY UTILITY [21]. Like SEER, this system tracks process execution trees and infers the contents of projects based on file accesses. It differs in that it restricts itself to loading unions of access trees, rather than attempting to create project clusters at a higher semantic level. This mechanism is much more limited. There is no facility for providing multidimensional semantic information, as SEER does via the external investigators discussed in Section 3.3.3. The system allows for certain other types of user input, but these are not integrated with the process-tree information.

Unfortunately, there is even less published data for SPY UTILITY than for CODA. The primary description appeared soon after the system was deployed, without quantitative results, and no subsequent data has been made available to date.

7 Future Work

SEER is running successfully in our workgroup. In the future, we would like to collect performance data for a larger user community. We plan to conduct further studies with the CODA user base and to port SEER to the WINDOWS environment. The latter port will make SEER available to business and management users, who often have very different behavior than computer scientists [14]. As part of this porting effort, we plan to analyze the performance of SEER in other settings and to compare this to our current data.

There are also significant opportunities for further development of the underlying mechanisms. The clustering algorithms, in particular, are more parameter-sensitive than one would like, and provide fruitful soil for study of more stable methodologies.

In addition, the predictive and inferential methods pioneered by SEER hold promise for other applications, such as Web caching, network file systems, and directory reorganization. We are currently investigating ways to apply our work to these and similar areas.

8 Conclusion

SEER has shown that fully-automated predictive hoarding is feasible, though the engineering challenges involved are daunting. The system is capable of supporting disconnected operation for lengthy periods with only occasional hoard misses, giving the user the illusion that the network is still present even in the complete absence of communication. This level of automation enables the entire *virtual-networking* paradigm of mobile operation [2].

An especially important contribution of SEER is the freedom from manual user configuration. While previous systems required the hoard contents to be specified partially or entirely by hand, SEER is able to infer project contents and make its hoarding decisions without intruding on the user's work. Such automated operation is critical with modern systems, since there is no practical way for the user to identify all files that will be required during disconnection.

Acknowledgments

We would like to express our thanks to the members of UCLA's File Mobility Group, whose willingness to try an unproven hoarding system made it possible to evaluate our software in a real-world context, and to Peter Reiher for innumerable discussions and suggestions on the design of the system. We would also like to acknowledge the contributions of the anonymous referees, and especially the patience and help of our shepherd, Karin Petersen, whose guidance was instrumental in achieving a coherent, readable result.

References

[1] 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*, pages 12–21, October 1990.

[2] Rajive Bagrodia, Wesley W. Chu, Leonard Kleinrock, and Gerald Popek. Vision, issues, and architecture for nomadic computing. *IEEE Personal Communications Magazine*, 2(6):14–27, December 1995.

[3] 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.

[4] James Griffioen and Randy Appleton. Performance measurements of automatic prefetching. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, September 1995.

[5] 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.

[6] 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.

[7] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71. University of California, Los Angeles, USENIX, June 1990.

[8] 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*, pages 2–5. University of California, Los Angeles, IEEE, November 1992.

[9] L. B. Huston and Peter Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 1–10. USENIX, 1993.

[10] 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.

[11] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[12] Thomas M. Kroeger and Darrell D. E. Long. Predicting file system actions from prior events. In *USENIX Conference Proceedings*, pages 319–328, San Diego, California, January 1996. USENIX.

[13] Geoffrey H. Kuenning. The design of the SEER predictive caching system. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[14] 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*, pages 291–306. USENIX, June 1994.

[15] Geoffrey Houston Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation*. PhD thesis, University of California, Los Angeles, Los Angeles, CA, May 1997. Also available as UCLA CSD Technical Report UCLA-CSD-970015.

[16] Mirko Křivánek. *Algorithmic and Geometric Aspects of Cluster Analysis*. Academia Praha, Prague, 1991.

[17] 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*, pages 183–195. University of California, Los Angeles, USENIX, June 1994.

[18] Peter Reiher, Jerry Popek, Michial Gunter, John Salomone, and David Ratner. Peer-to-peer reconciliation based replication for mobile computers. In *Proceedings of the ECOOP Workshop on Mobility and Replication*, July 1996.

[19] Mahadev Satyanarayanan, January 1997. Personal communication.

[20] 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*, pages 11–28, Cambridge, MA, August 1993. USENIX.

[21] Carl D. Tait, Hui Lei, Swarup Acharya, and Henry Chang. Intelligent file hoarding for mobile computers. In *Proceedings of MobiCom '95: The First International Conference on Mobile Computing and Networking*, pages 119–125, Berkeley, CA, November 1995.