6 Reducing the Burden on Users

Translucent caching must carefully balance the cost it imposes on the user with the benefits it offers. In designing the CodaConsole interface, I tried to reduce the burden placed on users wherever possible. The burden of translucent caching largely arises from occasions when the system requests advice. In particular, users are burdened by the need to define tasks and by the need to advise about network usage. If translucent caching requires its users to focus too much of their attention on managing the file cache, their productivity will suffer—a situation that is clearly not our goal. I employ three techniques to reduce the burden of translucent caching.

Two of these techniques reduce the burden required to define tasks and, more generally, to maintain the hoard database. One technique allows the system to automatically produce hoard profiles for programs in which the user has indicated an interest. It reduces the burden by lessening the amount of detail the user must provide. Another technique allows the system to offer advice to users regarding their task definitions. It reduces the burden by identifying mistakes in the user's hoard profile. The remaining technique reduces the number of occasions for which Venus requests advice. It models user patience in an effort to automate the process of determining whether or not to fetch an object over a weak network connection. In this chapter, I present each of these techniques.

6.1 Hoarding Programs Automatically

One burden imposed by the original Coda system was the need to produce hoard profiles for programs. Users created these profiles with the aid of a primitive tool called spy (as described in Section 2.3.3). The spy tool simply recorded all Coda references made during its execution. One problem with the tool arose because any process, not just the one on which the user was attempting to spy, could have made the observed references. Sorting out the references was the user's job. Another burden arose because, after generating the list of references, the user then had to post-process that list (using a procedure that assumed knowledge of Coda file identifiers) to create a profile acceptable to the hoard tool (see Section 2.3.2). Users managed the resulting profiles by hand, updating them occasionally as new requirements were discovered. Unfortunately, new discoveries were typically the result of a disconnected or weakly connected cache miss.

Thus, the first technique, used to reduce the burden that translucent caching places on users, focuses on the problem of identifying files on which programs depend. The goal is to automate the process of generating program profiles. It reduces the burden that hoarding, and thus translucent caching, places on the user because it takes over a task that was originally the user's responsibility. Further, it improves the accuracy of the generated program profile because it monitors the program continuously. Thus, the automatic monitor can identify references that the user might have missed because the program accesses the file only under certain, infrequent conditions. In this section, I will present the design and implementation of the *automatic program reference monitor*.

6.1.1 Design Requirements

The basic design for the automatic program reference monitor requires the system to observe every Coda reference made by various programs executed by individual users, and to do so continuously. The monitor should also distinguish references made to "program data" from those made to "user data", maintaining information only about program data. (Program data includes font files, style files, and mode files; user data includes source files.) This requirement allows program profiles to be used by many different tasks because it means that they do not depend upon an individual execution of that program.

Because the system is monitoring every reference users make, we must be careful to maintain the user's privacy. Users should not be given information about what files or programs another user accesses. For this reason, I limit the distribution of reference logs to only the user generating the original reference. Sharing of program profiles is not explicitly supported by the interface, though users can do so externally by giving other users access to the files that contain this information.

Further, because monitoring occurs on every file system request, we must be careful to maintain the system's efficiency. Whatever monitoring the system introduces, it must be fast because it occurs on the critical path of every file system request and will have a direct impact on the resulting system's performance.

6.1.2 Instrumentation

An important goal for the automatic program reference monitor was to support continuous monitoring. Continuous monitoring offers a significant advantage to the user. It allows the system to automatically detect the need for files that are referenced only infrequently, reducing the likelihood of cache misses. However, continuous monitoring risks serious performance degradation. For this reason, it became a primary consideration in the implementation. The system minimizes the burden such monitoring places on Venus by asking Venus to do as little work as possible and by requiring the Advice Monitor to pick up the burden.

Continuous monitoring requires Venus to inform the Advice Monitor of every file reference, specifying which program made the reference. The obvious design for such a

```
gnu-emacs 4471
<7f000259.246.23a> 4471 4471
<7f000259.136a.3ad> 4471 4471
gmake 4536
<7f000259.246.23a> 4536 4536
gnu-emacs 4471
<7f000259.246.23a> 4471 4471
.
.
.
```

Figure 6.1: Reference Log Fragment

This figure shows a fragment of a reference log. A reference log contains two different types of entries. One type of entry contains the name and process group identifier of the executable making references to the Coda file system. The first line of this log fragment shows an example of such an entry, in this case for `gnu-emacs`. The other type of entry contains the file identifier of a Coda file object as well as the process identifier and process group identifier of the process making the reference. As seen in this fragment, the `gnu-emacs` process (process group identifier 4471) referenced two file objects (to `<7f000259.246.23a>` and `<7f000259.136a.3ad>`) before being interrupted by the `gmake` process (process group identifier 4536).

system would extend the API by one additional call that would inform the Advice Monitor of these references. Upon receiving such a call, the Advice Monitor would record that the given program referenced the specified file. This design, although simple, imposes too great a burden on Venus. It adds a local RPC call to the critical path of every single file reference made by the user. For this reason, my system batches these requests and periodically ships a group of them to the Advice Monitor.

6.1.2.1 Venus Contribution

To support automatic program monitoring, Venus must record every reference and identify the program making that reference. Thus, when an Advice Monitor begins running, Venus opens a reference log. For each subsequent file reference, Venus outputs an entry containing the file identifier referenced as well as the PID (the Unix process identifier) and PGID (the Unix process group identifier) of the process making that reference. In addition, it must record the name of the program making the reference because that program may exit before post-processing begins. Periodically, Venus starts a new log and then signals the Advice Monitor.

A fragment of a reference log is shown in Figure 6.1. This log records that the `gnu-emacs` program (with PGID = 4471) referenced two files with Coda FIDS of <7f000259.246.23a> and <7f000259.136a.3ad>. Then, the `gmake` program (with PGID = 4536) referenced the file with FID <7f000259.246.23a>. After that, the `gnu-emacs` program began executing again and referenced <7f000259.246.23a> again.

Venus exploits locality of reference by maintaining a record of the most recently active program and its process group. It only outputs the name of the active program when the PGID of the current reference differs from that of the previous reference.

The beauty of this design is its simplicity. It exploits the standard I/O buffer, by using fprintf(), to avoid adding too much overhead to the critical path of servicing file references. Periodically (currently every 1000 references), a file reference must pay the cost of a local RPC call to inform the Advice Monitor that a reference log is available for post-processing. For all other file references, the overhead imposed by continuous program monitoring is the cost of a fprintf() to the reference log file and, if the PGID differs from the previous PGID, the cost of determining the name of the currently active program.

The overhead imposed by this monitoring as well as the monitoring required for providing advice to users, which is discussed in the next section, is small. To estimate this overhead, I performed a simple experiment that compares the time required to compile the Coda cache manager with and without the CodaConsole interface running, as calculated based upon the average of five compiles. This simple experiment showed that the overall overhead was less than 7%. (Without the interface, the compile took an average of 378 seconds; with it, the compile required 403 seconds.) Because the overhead was low, I did not perform a more involved experiment to determine the contributions of individual components of the interface.

6.1.2.2 Advice Monitor Contribution

The Advice Monitor's job begins once Venus signals that a reference log is available. Once this event occurs, the Advice Monitor wakes the ProgramLogHandler thread. This thread examines the log for references made by programs *under watch*. A program is said to be under watch if it is included in a task definition (see Section 4.10). Note that this includes programs contained in any task definition, including those not currently hoarded. The rationale for this decision is simply that the user has indicated an interest in this program and could hoard it at any time. We want to have built a good profile prior to the point at which the user needs it.

As described in the previous section, the reference log contains two types of entries. The first type identifies the active program. The second type identifies both the referenced file and the program responsible for making the reference. As the Advice Monitor post-processes a reference log, it maintains a record of the currently active program and whether it is under watch. If the currently active program is not under watch, then any references it made are skipped. If, however, the currently active program is under watch, then the Advice Monitor ensures that all subsequent references to *program data*, omitting duplicates, have been recorded in the program's profile.

The Advice Monitor distinguishes program data from *user data* using a simple heuristic that exploits the logical structure of the file system. This heuristic recognizes that users typically understand the layout of user and project volumes that they use, whereas they typically do not understand the layout of program volumes that they use. Thus, if the user includes a file or directory in a data definition (see Section 4.10), then the Advice Monitor considers all data stored in the volume owning that file or directory to be user data. Any references to volumes not considered user data are deemed program data and will be recorded in the program's profile.

This distinction between program data and user data is a key property of automatic program hoarding. It allows users to reuse program definitions in different tasks. For instance, if my Advice Monitor created a program profile for `latex` while I was writing a paper to submit to a conference, I can feel confident in reusing that program profile in a definition to support writing a workshop submission. The ability to reuse program profiles in subsequent task definitions reduces the burden placed on users of translucent caching.

This technique is flexible in handling different logical file system structures as well. For instance, suppose a different site places all program binaries in a volume named `s.local.bin`, all libraries in a volume named `s.local.lib`, and all font files in a volume named `s.local.fonts`. The heuristic would work equally well for this layout.

The limitation to this heuristic is its reliance upon a logical, volume-based structure. A file system that does not support volumes would obviously be unable to exploit this heuristic. However, as long as such a file system has a logical separation between user data and program data, then a similar heuristic should allow the system to distinguish between the different types of files. A file system that does not separate user data from program data in any logical way, however, will be unable to use this or any other heuristic to make this distinction. Its users will be prevented from reusing their program definitions, though they will still be able to exploit automatic program hoarding to build profiles for individual tasks.

6.2 Advising about Hoard Contents

A second technique, incorporated to help reduce the burden of user-assisted hoarding, allows the system to provide users with advice about their task definitions. The presentation of this advice was introduced in Section 4.6.2. Here, I will focus on the heuristics used to identify those files and directories that should be brought to the user's attention.

The heuristics take one of two forms. The first type identifies files that the user is not hoarding, but probably should be. Heuristics of this first type identify files that are accessed infrequently but repeatedly, files that the user has successfully accessed during a disconnected session but has not hoarded, and files that the user could not access during a disconnected session because the file was not cached. The second type identifies files that are currently hoarded, but probably should not be. These heuristics identify files that have not been accessed during recent disconnected sessions or that have been accessed infrequently during disconnected sessions.

In this section, I will begin by presenting each heuristic and the intuition behind it. I will then briefly describe how the system collects the information necessary to identify these files.

6.2.1 Heuristics

I will present the five heuristics used to identify files that the user may want to consider adding or removing from the task definitions. These heuristics are all conceptually simple, but identify classes of files that users have been known to forget.

Infrequent, but Repeated, Access—This heurisitic looks for files that are accessed infrequently, but repeatedly, over long periods of time. A good example of such a file is the `.xinitrc` file. This file is used every time the user starts X. After starting X, the user is unlikely to access this file again for days, or even weeks. This usage pattern makes the `.xinitrc` file a prime target for replacement, unless the user has included it in a task definition. Unfortunately, because it is accessed indirectly by a program, it is also an easy file for users to forget. Thus, this heuristic analyzes the user's reference stream specifically looking for files that are accessed infrequently, but repeatedly, over a long period of time. It extends the horizon by maintaining information about files that are replaced from the cache. This heuristic may identify objects identified by the *unlucky reference* heuristic (below) if the object experiences a cache miss, but can identify such an object even if it does not. Further, this heuristic will identify files whose priority may need to be increased as well as files that simply need to be included in a definition.

Unlucky Reference—This heuristic identifies files that experienced a cache miss. Although this is an obvious heuristic, it serves to remind users of files that they may want to add to their task definitions. It relies on locality of reference: if the user attempts to access a file, she is likely to access it again in the future. Specifically, this heuristic identifies all files which experienced cache misses, while the CodaConsole was running, since the last time the interface offered the user advice.

Lucky Disconnected Reference—This heuristic analyzes the user's reference stream looking for files that were accessed during a disconnected session, but for which the user has not included in a task definition. Such files were successfully accessed during disconnected operation by simple good fortune. Had other references caused the file to be replaced, the system would not have known to refetch the file in preparation for disconnected operation. This heuristic identifies all cached, but unhoarded, files accessed during any disconnected session. As above, this heuristic assumes locality of reference.

Infrequent Disconnected Use—This heuristic identifies files that are used only infrequently during disconnected sessions. Specifically, it identifies files that have been used in fewer than 50% of disconnected sessions, though this value should eventually become a user-configurable parameter (see Section 9.2.2.1). These files are ones included in a task definition, but for which it appears that inclusion is unnecessary. Such files may have been heavily used for a period of time, but are now no longer needed and thus can be removed. Their inclusion may be taking cache space away from files that have a lower priority, but that are used more frequently. This heuristic brings this anomaly to the user's attention.

Ancient Disconnected Use—This heuristic identifies files that have not been used during the two most recent disconnected sessions, though this number will eventually become a user-configurable parameter (see Section 9.2.2.1). As in the previous heuristic, these files have an augmented priority, but they appear not to need the increased priority. These files may belong to a project that has been finished, but whose tasks have not been removed. Once again, their increased priority may take cache space away from lower priority objects that are used more frequently.

6.2.2 Instrumentation

Each of the heuristics described in the previous section depends upon statistics gathered during use of the system. In this section, I will briefly describe how I instrumented Venus and the Advice Monitor to gather these statistics.

6.2.2.1 Replacement Statistics

The instrumentation necessary to collect the statistics supporting the *infrequent, but repeated access* heuristic is complicated by the fact that these objects may not be referenced again before they are replaced (e.g., the `.xinitrc` file). Thus, the data necessary to identify this class of files must persist across cache replacements. The basic strategy, then, is to maintain a small amount of information about previously cached objects.

The intuitive design to support this data collection would be to augment the data structures internal to Venus to allow Venus to maintain ghost entries for previously cached objects. Unfortunately, this design suffers from a fundamental problem. If Venus were to maintain this information, it would need to store it in recoverable virtual memory (RVM). Even though the amount of information stored for each previously cached file may be quite small, the number of previously cached files is likely to be large enough that the overall storage requirement would overwhelm Venus' modest RVM resources.

The actual design, similar to that employed by automatic program hoarding above, requires Venus to log data about each file system object that it replaces. This data contains the Coda file identifier of the object as well as indications of whether the status or the data (or both) were replaced. Periodically, Venus starts a new log and then signals the Advice Monitor that the old log is available for post-processing. The Advice Monitor's ReplacementLogHandler thread post-processes this data, building a database containing the pathnames of replaced objects and counts of the number of times their status and data have been replaced. Once an object's status or data has been replaced 10 times, the object is brought to the user's attention.

This revised design places the responsibility for collecting the data on Venus and the responsibility for maintaining the data on the Advice Monitor. It minimizing the amount of effort required to collect the data, since that effort is likely to be on the critical path for servicing file system requests. In fact, the overhead of this data collection is small, as was discussed previously in Section 6.1.2.1. The burden of maintaining the data falls to the Advice Monitor, which is the appropriate place from a software engineering perspective. The data is stored in a database that is stored on the local disk rather than in Venus' RVM space. Overall, the revised design is far superior to the intuitive design while remaining functionally equivalent.

6.2.2.2 Cache Miss Statistics

The instrumentation necessary to collect the statistics supporting the *unlucky reference* heuristic is straightforward. Each time Venus triggers a cache miss query during disconnected, weakly connected, or read disconnected operation (see Sections 4.6.1.1, 4.6.1.2, and 4.6.1.3, respectively), the interface keeps track of the pathname. On demand, the interface can generate a list of files that the user attempted to access, but for which Venus did not have cached data.

6.2.2.3 Disconnected Usage Statistics

The next three heuristics (*lucky disconnected reference*, *infrequent disconnected use*, and *ancient disconnected use*) rely on the user's disconnected behavior. To support these heuristics, I had to instrument Venus to collect disconnected usage statistics. Each time a volume transitions to disconnected operation, it records the current reference count in RVM. When a file is accessed during the disconnection, its last reference record is updated automatically. Upon reconnection, Venus first determines whether or not the disconnected session was *eligible*. Eligibility depends upon whether or not the user made any references during the period of time the volume was disconnected (i.e., if the current reference counter is higher than the disconnected reference counter). If the disconnected session is not eligible, then the user did not use any objects from this volume so the system does not collect statistics about this session. If the disconnected session is eligible, however, then Venus examines every cached object from the reconnected volume and compares its last reference record to the reference count at the point of disconnection. If the last reference count is higher than this value, the object was used during the disconnected session; otherwise it was not.

Informing the Advice Monitor of these statistics upon reconnection would involve a substantial amount of data. To avoid a substantial performance penalty, Venus produces these statistics only upon demand. When the Advice Monitor requests disconnected usage statistics, Venus generates a list of cached objects meeting certain disconnected usage criteria. These data are then used to identify objects for the *lucky disconnected reference, infrequent disconnected use*, and *ancient disconnected use* heuristics as described in Section 6.2.1.

6.3 Modeling User Patience

During the course of weakly connected operation, the system may encounter situations in which a file referenced by the user is not available in the cache. At times, that file may be important enough that the user is willing to wait a substantial amount of time for it to be fetched; at other times, the user may not be willing to wait long at all. Ideally, the system could distinguish those files for which a user is willing to wait from those for which she is not. I hypothesize that I can model this *user patience* and that this model depends upon just two things:

1. The importance of the file
2. The expected fetch time

If the system were able to predict the user's willingness to wait with reasonable confidence, it could reduce the frequency with which it requests advice during weak connectivity, thereby reducing the burden of translucent caching.

I call the maximum time that a user is willing to wait for a particular file the user's *patience threshold* (τ) for that file. This threshold depends critically upon how important the user perceives that file to be. For a very important file, the user is probably willing to wait many minutes. Because user perception of importance is the notion captured by the hoard priority, P, of an object, I posit that τ should be a function of P. At present, I am not aware of any data that could be the scientific basis for establishing the form of this relationship. Hence, I use a function based solely on intuition, but have performed some analysis to determine appropriate parameter values and have also structured the implementation to make it easy to substitute better functions as they are discovered.

6.3.1 Proposed Model

I conjecture that patience is similar to other human processes, such as vision, that exhibit logarithmic sensitivity [9]. This suggests a relationship of the form

$$\tau = \alpha + \beta e^{\gamma P}$$

where β and γ are scaling parameters and α represents a lower bound on patience. I chose parameter settings based on their ability to yield plausible patience values for files commonly found in the hoard profiles of Coda users. I use the values α=**6**, β=1, and γ=**0**.01, based upon the analysis presented in the following section.

Figure 6.2 illustrates the resulting model of user patience. Rather than expressing τ in terms of seconds, I have converted it into the size of the largest file that can be fetched in that time at a given bandwidth. For example, 60 seconds at a bandwidth of 64 Kb/s yields a maximum file size of 480 KB. Each curve in this figure shows τ as a function of P for a given bandwidth. In the region below this curve, cache misses are transparently handled and pre-approval is granted automatically during hoard walks.

This curve predicts that users would be willing to wait indefinitely for sufficiently large files. Clearly, this is not true. HCI researchers have found that delays beyond approximately 15 seconds are disruptive [48]. For this reason, any delay greater than 15 seconds is automatically presented to the user. Thus, the function employed by Venus is

$$\tau = \min(15, \alpha + \beta e^{\gamma P}) .$$

The user patience model is the source of adaptivity in cache miss handling. It maintains usability at all bandwidths by balancing two factors that intrude upon transparency. At very low bandwidths, the delays in fetching large files annoy users more than the need for interaction. As bandwidth rises, delays shrink and the interaction becomes more annoying. To preserve usability, the model must handle more cases transparently. In the limit, at strong connectivity, cache misses are fully transparent.

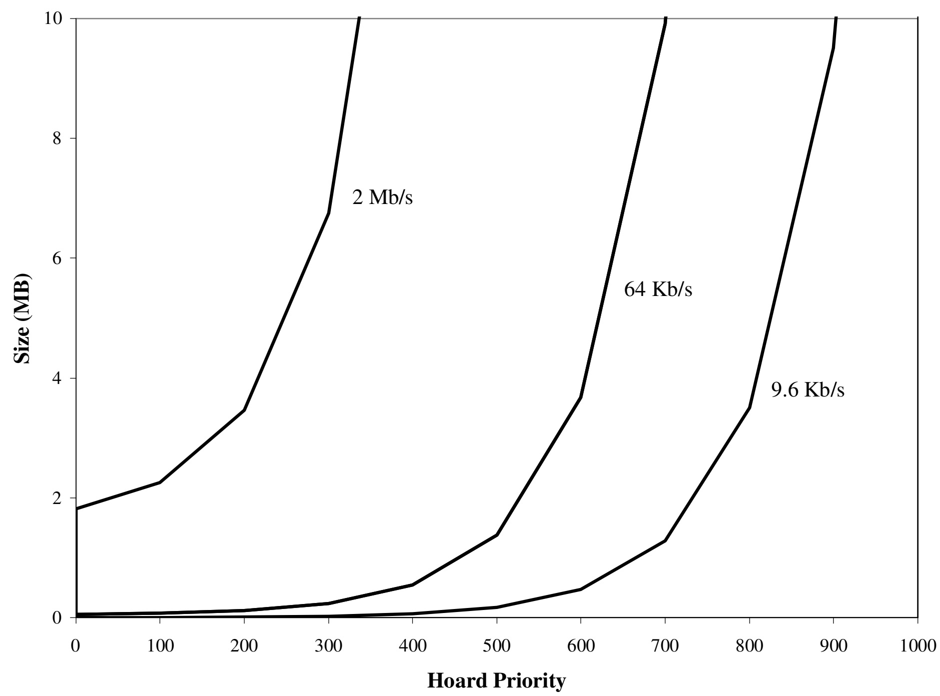


Figure 6.2: User Patience Model

This figure shows the user patience model. The X-axis shows the hoard priority of an object. The Y-axis shows the size of the object. Parameter values for this figure were $\alpha = 6$, $\beta = 1$, $\gamma = 0.01$. Each curve represents a different network bandwidth. The area below the curve represents those objects that will be fetched automatically; the area above the curve represents those objects that require user intervention before being fetched. For example, at 9.6 Kb/s, most high priority objects are fetched regardless of size, but only very small objects with low priority will be fetched automatically. At 2 Mb/s, however, even a 1-MB file with a priority of 0 will be fetched automatically.

6.3.2 Exploring the Parameters

The accuracy of the patience threshold depends upon its parameters (α, β, γ) and upon its argument (P). I begin this section with a brief discussion of the argument, and then examine each of the model's parameters.

6.3.2.1 Priority (P)

The argument to the function is the priority of the object to be fetched. The user indirectly assigns this priority when requesting that a task be hoarded using the window shown in Figure 4.15. These priorities range from 1 to N (the number of hoarded tasks), where 1 is the highest priority task. From these user-assigned priorities, the system must assign hoard priorities meaningful to Venus (in the range from 1 to 1000).

The obvious strategy for performing this translation would be to space out the hoard priorities evenly. This strategy has two important features: its ease of implementation and its ability to handle up to 1000 hoarded tasks. An unfortunate consequence, however, is that, as N increases, tasks become difficult for Venus to distinguish because the hoard priority of an object is but one component of its overall priority (see Section 2.2.2).

The strategy I used, instead, spreads high priority tasks at the cost of compressing low priority tasks. By spreading high priority tasks, I improve the odds that Venus can ultimately distinguish between files belonging to different high priority tasks, and thus allow Venus to make more intelligent caching decisions. This ability comes with a cost, however. By compressing low priority tasks, I decrease the odds that Venus can distinguish between files belonging to different low priority tasks.

In effect, this strategy increases Venus' ability to distinguish high priority tasks at the cost of decreasing Venus' ability to distinguish low priority tasks. This trade-off is important for two reasons. First, this trade-off assumes that the difference in importance between tasks prioritized 1st and 2nd by the user is generally higher than the difference in importance between tasks prioritized 31st and 32nd. Second, it increases the probability that all files belonging to a task with priority A are cached in preference to any files belonging to the task with priority A+1, for values of A near 1.[18] Because of Venus' prioritized cache management algorithm, objects with lower hoard priorities can, under certain circumstances, be cached in preference to objects with higher priorities.[19] Spreading the hoard priorities of the most important tasks makes this situation less likely to occur with these tasks.

Although this strategy reduces the number of unique tasks from 1000 to 88, this is still many more than a user is likely to need. The highest priority tasks are separated by a distance of 200; the lowest priority tasks are separated by a distance of only 1. To spread

[18] Because of the trade-off, this probability decreases as A increases.

[19] These circumstances require that the user has accessed the lower priority objects more recently. For this to happen, the reference priority component of the overall priority overwhelm the hoard priority component. Please refer to Section 2.2.2 for further details regarding Coda's prioritized cache management.

the assigned priorities, I employ the Fibonacci sequence. The number of elements spread by N points is Fib(i), as N decreases from 200 and i increases from 1. Thus, there is one gap of 200 priority units, one gap of 100 priority units, two gaps of 50 units, etc. If the user hoards more than 88 tasks, the additional tasks are all assigned the same low hoard priority and are, thus, indistinguishable from one another from the perspective of Venus. This distribution is summarized in Figure 6.3.

The use of the Fibonacci sequence is not critical to this strategy. The domain of the function (the user's priorities) must be the set of integers. The range of the function (Venus' hoard priorities) must be the set of integers from 0 to 1000. The mapping of user priorities to hoard priorities should be a nonincreasing, nonlinear function. The nonlinearity should spread lower values of the domain further apart than higher values of the domain, up to a point. Beyond that point, all values of the domain can map onto a single (low) value of the range. Any function with these characteristics should suffice. I use the Fibonacci sequence described above for its simplicity. Numerous alternatives, including exponential, inverse quadratic, and Zipf distributions, could work equally well.

6.3.2.2 Theoretical Lower Bound (α)

The first parameter I consider, α, represents a lower bound on patience. Even for an unimportant object, the user would prefer to tolerate a delay of up to α seconds rather than dealing with a cache miss; thus, this parameter specifies the amount of time the user is willing to wait for even the least important file. From a different perspective, α offers a way to prevent the interface from asking the user a question (e.g., the weak-miss query of Section 4.6.1.2) in situations where the system can fetch the object faster than the user can respond to the question. Thus, a candidate value for α is the minimum time required for a typical user to respond to the weak-miss query.

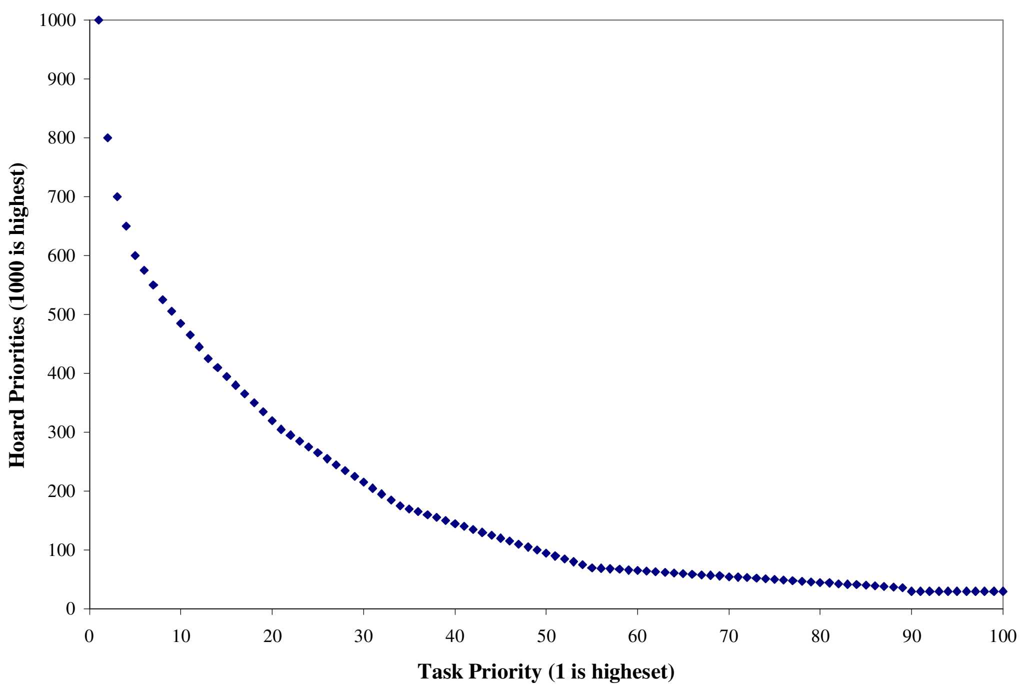
To determine the time required to respond to the weak-miss query, I use a Keystroke-Level Model (KLM), a form of GOMS analysis [7]. In GOMS modeling, one breaks down a task into a series of goals, operators, methods, and selection rules. For my models, I assume, among other things (see Figure 6.4), that the user is an expert. This assumption implies that the user has a method for responding to a weak-miss query, that the method is reasonably efficient, and that I need only define the actions specified in that method. The specific method used in my model is the one requiring the fewest operators. Once the operators necessary to complete the method have been identified, the time required to execute that method can be estimated by calculating the sum of the time required to apply each individual operator, as summarized in Figure 6.5.

For the model of the weak-miss query, I consider two pointing devices and two configuration options. The two pointing devices I consider are a mouse and a keyboard pointing device[20] common in laptop computers. The keyboard pointing device is of a small rubber cylinder (similar in size to the eraser on a pencil) that is situated between the

[20] IBM calls their device the TrackPoint.

Number of gaps	Units Separating Tasks
1	200
1	100
2	50
3	25
5	20
8	15
13	10
21	5
34	1
Total = 88	

(a)



(b)

Figure 6.3: Distribution of Task Priorities

This figure shows the distribution of task priorities. The table shown in view (a) defines how hoard priorities are assigned to hoarded tasks. The highest priority tasks are assigned the hoard priority 1000. The next most important task receives a priority of 800, or 200 units below the previous priority. The next task receives a priority of 700, or 100 units below the previous priority. The next two tasks are separated by 50 units and receive the priorities 650 and 600, and so on. More important tasks have more distinct hoard priorities. The chart shown in view (b) illustrates the shape of this distribution.

Assumptions

Expert user:
    knows how to use indicator lights
    knows where to look for *Advice Information* window to appear
    knows how to select event in *Advice Information* window

Weak-miss query is the only pending advice.

Weak Miss Event settings:
    no beeping
    no flashing

User is a touch typist.
User is an expert with the pointing device.
User must mentally prepare (**M**) to take action at beginning of method.

Figure 6.4: Assumptions Made in Building KLM Models.
This table shows the assumptions that I made in building the Keystroke-Level Models.

KLM Operator	Description	Time (ms)
H[mouse]	Home hand on device.	400
H[point]	Home finger on device.	75
B[button]	Press button (up or down).	100
M	Mentally prepare.	1200
P[location]	Point with mouse to target on display.	1100
R[type, machine]	System response time.	$SRT(t,m)$

Figure 6.5: Operators of the Keystroke-Level Model
The **H**, **M**, **P**, and **R** operators and their time estimates are defined in Chapter 8 of Card, Moran, and Newell [7], as modified by Kieras [21]. The **B** operator is defined by Kieras. Homing to a keyboard pointing device is assumed to be equivalent to moving a finger from one key to a neighboring key and is derived from Equation 2 of [16]. The time estimates for the **R** operator were measured from the interface mockup and appear in Figure 6.6, parameterized by the type of response time and the machine on which it was measured.

Type	Description	System Response Time (s)	
		Laptop	Desktop
A-cold	*Advice* Indicator ⇒	1.21	1.14
A-warm	*Advice Information* Window	0.79	0.68
B-cold	*Advice Information* Window ⇒	0.19	0.24
B-warm	*Weak-miss Query* Window	0.16	0.17

Figure 6.6: Measured System Response Times

The time measurements were obtained via the `time` facility provided by Tcl/Tk [56]. I performed 7 trials under each condition on each machine. The measurement reported represents the mean of 5 trials (high and low were thrown out). The cold measurements represent the time necessary to bring up the window for the first time. The warm measurements represent the time necessary to bring up the window for the second time. The measurements for response time B required that I modify the code slightly to return immediately after displaying the query window to the user. (Otherwise, the measurement would have included the delay caused by the user answering the question.) The measurements for the Desktop were taken on a DECstation 5000/200 running Mach 2.5. The measurements for the Laptop were taken on an IBM Thinkpad 701c (75 MHz i486-based machine) also running Mach 2.5. The interface was otherwise idle. The machine was running multi-user, on the network, and with the standard complement of system daemons. A single user was logged in and running X windows; that user was inactive except for the actions necessary for obtaining the measurements.

‘G’, ‘H’, and ‘B’ keys of a QWERTY keyboard. Two buttons, located below the space bar, serve the same purpose as the left and right mouse buttons. The two configuration options I consider are whether or not the user has configured the weak-miss query to pop up a window on the monitor. The models for the four cases appear in Figures 6.7–6.10.

Consider Method #1, shown in Figure 6.7. For this method, I assume the user is using a mouse and has configured the interface not to pop up the weak-miss query. This method contains a total of twelve steps. It is invoked when the user notices that the interface has turned the *Advice* indicator red. The user decides to investigate and begins by reaching for the mouse (I assume the user was interrupted from a typing activity so that her hands are initially hovered over the keyboard), pointing to the *Advice* indicator, and double-clicking the left button. The two mental operators normally required to point the mouse and to double-click a button have been anticipated by the home-to-mouse operator and have thus been eliminated [7].

Once the system has displayed the *Advice Information* window, the user points to the *Weak Miss* entry and double-clicks the left button. The mental operator required to point at the entry is eliminated because it could be anticipated and is therefore assumed to have occurred concurrently with the system response time. Further, the mental operator required to double-click the mouse was anticipated by the previous point operator.

Once the system displays the request, the user must make a decision[21], point to her choice, and then click the left button. The first mental operator cannot be eliminated nor can it overlap with the system response time, because the information necessary to make the decision is not available until the window has been displayed. The mental operator preceding the button click, however, is fully anticipated by the previous point operation.

Once the user has completed the request, she returns the cursor to her work area and moves her hands back to the keyboard. The mental operator preceding this pointing action is not eliminated because I assume she verifies that she has, indeed, completed all necessary advice activity. Note that the model does not consider the time required to close the *Advice Information* window because such an action is not strictly necessary. If the model were to include this action, the time estimate would increase by 1.3 seconds (**MPBB** inserted prior to the "Point to Previous Window" action with the mental operator in the following action eliminated).

Because the other three models (shown in Figures 6.8-6.10) are similar, I will summarize their differences from this first model, but I will not discuss each of them in detail. The only difference between the two pointing device models is the amount of time required to home to the pointing device. In the case of a mouse, the user must move a hand from the keyboard to the mouse. In the case of the keyboard-pointing device, the user must move

User Action	KLM Operator
Notice Indicator Light	**M**
Reach for Mouse	M**H**[mouse]
Point to Indicator Lights	M**P**[indicator]
Double-click Left Button	M**BBBB**[left]
System Response Time A	**R**[A-cold, desktop]
Point to *Advice Information* Window	**MP**[advice]
Double-click Left Button	M**BBBB**[left]
System Response Time B	**R**[B-cold, desktop]
Point to Command	**MP**[command]
Click Left Button	M**BB**[left]
Point to Previous Work Window	**MP**[work]
Reach for Keyboard	M**H**[keyboard]

$$T_{execute} = 3 \cdot t_M + 10 \cdot t_B + 2 \cdot t_{H_M} + 4 \cdot t_P + t_{R_1} + t_{R_2} = 11.18 \text{ seconds}$$

Figure 6.7: Method #1—Desktop without Popup Query

Operators shown in light gray were eliminated by Rule #1 of the Keystroke-Level Model. The one shown in dark gray occurred concurrently with the preceding system response time. (Although this **M** operator should theoretically take longer than the system response time, the system response time takes longer in practice.)

[21] I do not include the time necessary for the user to make this decision in my models because I assume that it is identical in each of the four cases.

User Action	KLM Operator
Notice Indicator Light	**M**
Reach for Mouse	M**H**[point]
Point to Indicator Lights	M**P**[indicator]
Double-click Left Button	M**BBBB**[left]
System Response Time A	**R**[A-cold, laptop]
Point to *Advice Information* Window	M**P**[advice]
Double-click Left Button	M**BBBB**[left]
System Response Time B	**R**[B-cold, laptop]
Point to Command	**MP**[command]
Click Left Button	M**BB**[left]
Point to Previous Work Window	**MP**[work]
Reach for Keyboard	M**H**[keyboard]
$T_{execute} = 3 \cdot t_M + 10 \cdot t_B + 2 \cdot t_{H_P} + 4 \cdot t_P + t_{R_1} + t_{R_2} = 10.46\,\text{second}s$	

Figure 6.8: Method #2—Laptop without Popup Query

User Action	KLM Operator
Task Interruption	**M**
Reach for Mouse	M**H**[mouse]
Point to Command	**MP**[command]
Click Left Button	M**BB**[left]
Point to Previous Work Window	**MP**[work]
Reach for Keyboard	M**H**[keyboard]
$T_{execute} = 3 \cdot t_M + 2 \cdot t_B + 2 \cdot t_{H_M} + 2 \cdot t_P = 6.8\,\text{second}s$	

Figure 6.9: Method #3—Desktop with Popup Query

User Action	KLM Operator
Task Interruption	**M**
Reach for Mouse	M**H**[point]
Point to Command	**MP**[command]
Click Left Button	M**BB**[left]
Point to Previous Work Window	**MP**[work]
Reach for Keyboard	M**H**[keyboard]
$T_{execute} = 3 \cdot t_M + 2 \cdot t_{H_P} + 2 \cdot t_B + 2 \cdot t_P = 6.08\,\text{second}s$	

Figure 6.10: Method #4—Laptop with Popup Query

her index finger from a keyboard key to the pointing device.[22] Homing a finger to the keyboard-pointing device is assumed to be equivalent to moving a finger from one key to a nearby one. The total time difference between these two conditions is approximately 0.7 seconds.

Note, however, that the assumption that the only difference between these pointing devices is their homing time is a simplification. Researchers comparing the performance of people using these devices to performance using mice have reported mixed results. An early study showed that, for pure pointing tasks, the pointing stick is 20-25% slower than a mouse [4, 43]. Research has also shown that, for intermixed pointing and keyboard tasks (such as the one under discussion), the keyboard pointing stick is actually faster than a mouse [43]. Unpublished studies show that, for more recent keyboard-pointing devices, experienced users come within 5% of mouse performance. My assumption that the main difference between the keyboard-pointing device and the mouse is in its homing time reflects the fact that the task in question is a mixed pointing and keyboard task, and that the user is experienced with the device in question.

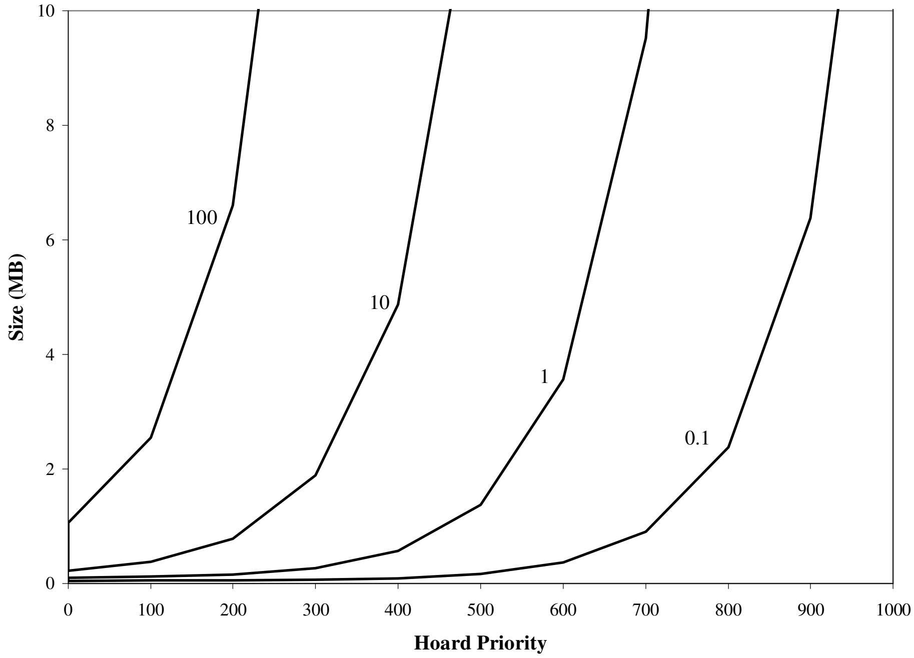
The differences between the two configuration options are more substantial. In the case of the no-popup option, the user must first open the *Advice Information* window and then open the advice request window. Neither of these activities is required of the popup option since the system automatically opens this window for the user. The models comparing these two options predict that this change in event configuration makes a difference of more than 4 seconds in the time required to complete the questionnaire.

The time required to answer the weak-miss query, depending upon pointing device and configuration options, ranges from 6 to 11 seconds. The models show that the time required changes only minimally based upon the choice of pointing device, but quite substantially based upon the configuration of events. Because of this disparity, the default value for α is 6 seconds, but the CodaConsole increases this value to 11 seconds when the user's event configuration is set to not pop up this query.

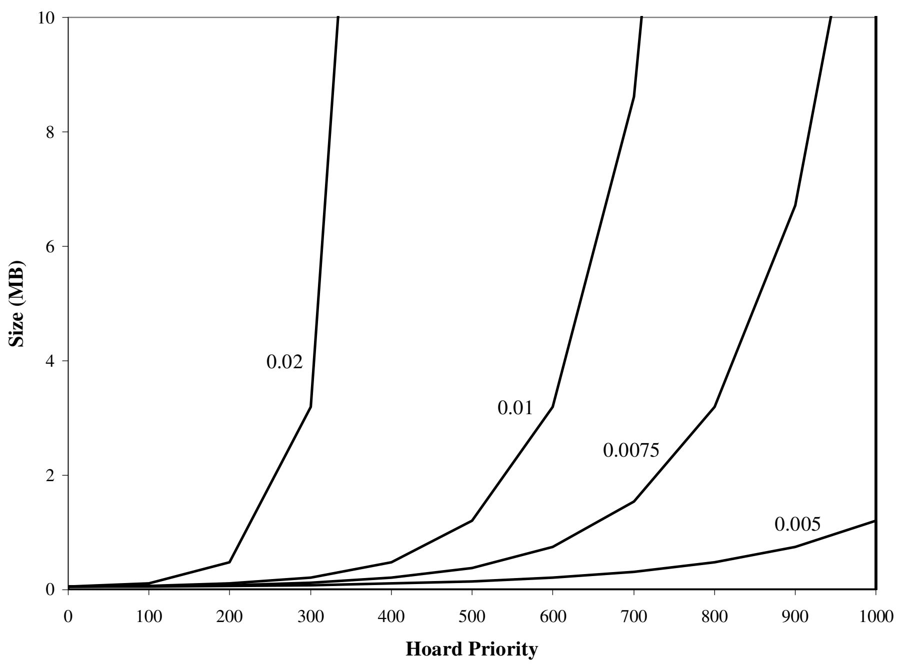
6.3.2.3 Sensitivity Analysis of β and γ

The other two parameters to the user patience model, β and γ, control the shape of the curve. Figure 6.11 shows how changes to these parameters affect the model. As is evident in these figures, decreasing the value of β from its default of 1 or decreasing the value of γ from its default of 0.01 flattens the curve. Similarly, increasing their values sharpens the curve. The default values for each of these parameters were chosen so that the model would yield plausible patience values for files and priorities commonly found in the hoard profiles of Coda users. To determine better estimates for those parameters, one would need to verify the model and collect data regarding the patience of users in practice (see Section 9.2.1).

[22] The user must also move her thumb to the primary button (e.g. the equivalent of the left mouse button). I assume this operation overlaps with the home to the pointing device and the subsequent movement of the cursor.



(a) Sensitivity to β



(b) Sensitivity to γ

Figure 6.11: Sensitivity to Changes in β and γ

These figures show the effect of changing the value of the β and γ parameters to the user patience model. The value of β in view (a) varies from 0.1 to 100; the values of α and γ remain constant at 6 and 0.01, respectively. The value of γ in view (b) varies from 0.005 to 0.02; the values of α and β remain constant at 6 and 1, respectively.

6.4 Summary

In this chapter, I have presented three techniques intended to reduce the burdens of translucent caching. The first technique, hoarding programs automatically, reduces the amount of effort users must expend building and maintaining hoard profiles for the programs they use. The second, advising about hoard contents, reduces the likelihood that users will experience failures of availability by identifying file system objects that the user may want to consider adding to a task definition. This technique also identifies objects that the user may want to consider removing from task definitions, thereby reducing the time and memory users must invest in hoarding. Finally, the third technique, modelling user patience, reduces the number of advice requests initiated by Venus. Reducing this number directly reduces the time and cognitive requirements placed on the user.