# Profile-Driven Cache Management

**Mitch Cherniack**
Brandeis University
Waltham, MA 02454
mfc@cs.brandeis.edu

**Eduardo F. Galvez**
Brandeis University
Waltham, MA 02454
eddie@cs.brandeis.edu

**Michael J. Franklin**
University of California
Berkeley, CA 94720
franklin@cs.berkeley.edu

**Stan Zdonik**
Brown University
Providence, RI 02912
sbz@cs.brown.edu

## Abstract

*Modern distributed information systems cope with disconnection and limited bandwidth by using caches. In communication-constrained situations, traditional demand-driven approaches are inadequate. Instead, caches must be preloaded in order to mitigate the absence of connectivity or the paucity of bandwidth. In this paper, we propose to use application-level knowledge expressed as* **profiles** *to manage the contents of caches. We propose a simple, but rich profile language that permits high-level expression of a user's data needs for the purpose of expressing desirable contents of a cache. We consider techniques for prefetching a cache on the basis of profiles expressed in our framework, both for basic and preemptive prefetching, the latter referring to the case where staging a cache can be interrupted at any point without prior warning. We examine the effectiveness of three profile processing techniques, and show that the rich expressivity of our profile language does not prevent a fairly simple greedy algorithm from being an effective processing technique. We also show that for a large shared cache, multiple clients' profiles can be combined into a single superprofile that is representative of them all, but that when the number of clients with profiles is significantly large, a randomized approach is more scalable than a greedy approach. We believe that profiles, as described in this paper, are an enabling technology that could spawn a rich new area of research beyond cache management into network data management in general.*[1]

## 1. Introduction

Modern distributed information systems cope with disconnection and limited bandwidth by caching. In communication-constrained situations, traditional demand-driven approaches are inadequate. Instead, caches must be preloaded to mitigate the absence of connectivity or the paucity of bandwidth. In this paper, we propose to use application-level knowledge expressed as *profiles* to manage the contents of caches.

User profiles are used by many applications to specify information to deliver to users (e.g., AvantGo). We extend this notion to provide data management hints for preloading and prestaging caches in distributed environments. For us, a profile is not a demand for data, but a pragma or hint of what might be useful to prefetch closer to the user.

There are many scenarios in which such hints can be useful. If a client machine that is connected to a data source by a low-bandwidth link, a client-side cache could be used to hide access latency by using slack bandwidth to bring useful data closer to the point of use. Disconnected use is a more extreme example in which prefetching must be done while a device is connected to allow access at times when it's not. Another example involves a cluster of users connected to a common node that is itself connected to one or more data sources across a thin pipe. The latency introduced by the thin pipe can be reduced by intelligent use of a cache. In all of the above scenarios, the system must choose subsets of the items of interest to prefetch. A simple listing of items, as provided in other profiling schemes, is insufficient since it provides no information about relative value and interdependencies among the data items.

In this paper, we present a profile specification scheme that allows us to identify items of interest, but also allows us to express their *relative importance* through weighting. Furthermore, our scheme supports the specification of *data dependencies* (e.g., we are able to say that directions to a hotel from the airport are only useful in the presence of the corresponding airline and hotel reservations). Finally, we add the ability to specify *thresholds*, as in up to three restaurant recommendations are useful while any more are not. All three of these extensions to profile formulation reflect a global view of how a selected item relates to other selected items. Extended profiles such as these will allow us to fill our cache with a much more accurate set of objects. This paper makes three fundamental points towards this end:

**Specification:** We propose a simple language that separates the expression of interesting objects from their utility value. Utilities can be expressed with primitives that handle the three forms of dependency given above.

**Algorithms:** We present and compare some heuristic techniques for filling a cache based on our extended profiles.

**Profile Composition:** We discuss how to combine several profiles into one *superprofile* that subsumes its constituents (as is required when a cache is shared by multiple users).

[1]This work was supported in part by the National Science Foundation under NSF Grant number IIS00-86057.

This study will show that profiles with dependencies and thresholds are easily expressed, and that a simple greedy algorithm does a good job processing profiles to select items to prefetch into a cache. We look at basic cache prefetching, as well as preemptive cache prefetching, which takes into account the possibility that prefetching may get interrupted prematurely. We then show for shared caches with very large "superprofiles", that a more scalable approach is to use a randomized algorithm (simulated annealing).

## 2. Profile-Driven Cache Prefetching

Two applications of profile-driven cache prefetching are data recharging and thin pipe environments. Both applications need to allocate limited cache space to data objects.

### 2.1. Example Applications

**Data Recharging:** Data recharging [9] is best understood in the context of mobile computing environments. Mobile computers have two fundamental renewable resources: power and data. The goal of data recharging is to make recharging data from the data grid as simple and robust as recharging power from the power grid. Based on a user profile, data recharging middleware gathers data of interest to a user, and when the user next connects to the network, this data is delivered to the user's mobile device. In this case, the profile would reflect some aspect of the user's context (e.g., location, workflow) and data delivered would depend on available resources on the mobile device (e.g., memory, applications). The order in which items are delivered also matters if unplanned disconnection is possible.

In the mobile environment, computers are often memory-limited. Thus, given a large amount of potentially interesting data, a data recharging system must make decisions about what subsets of this data are to be allocated to the limited memory resource.

**Thin-Pipe Environments:** In this situation, the client is connected to the network through a thin pipe. A thin pipe can be a result of a bandwidth constrained connection or a heavily overloaded server. The essential characteristic here is that using the pipe introduces very high latency. Inserting a cache at the client end attempts to hide this latency. An item whose access is extremely likely is prefetched in order to eliminate the latency of its first access. Prefetching is inherently predictive, and profiles support our prediction.

### 2.2 Shared Caches

In many networked environments, caches are shared among multiple users and cache contents must be managed in a way that maximizes the benefit for the user community as a whole. The applications described above have shared cache analogs. In data recharging, we assume that users can connect to well-defined charging stations. These charging stations would maintain a shared cache that would service users who are likely to connect there in the future. For thin pipes, a shared cache is placed between the connection and a user community. This cache is used to mitigate the latency effects of the thin pipe.

```
PROFILE Traveler
 DOMAIN
  RC = related:www.hertz.com
  Sh = "shuttle schedules" AND "airport" AND "Boston"
  Di = "directions to downtown Boston" AND "airport"
  Ho = "Hotel" AND "downtown Boston"
  Re = "Restaurant Review" AND "downtown Boston"
 UTILITY
  U (Re) = 1
  U (RC [#Di > 0]) = UPTO (2, 2, 0)
  U (Sh) = UPTO (1, 3, 0)
  U (Di [#Ho > 0]) = UPTO (1, 1, 0)
  U (Ho [#Di > 0 OR #Sh > 0]) = UPTO (1, 2, 0)
END
```

Figure 1: A Profile For Data Recharging

### 2.3. A Profile Language and Example Profile

To illustrate our profile language, we present an example profile in Figure 1. Observe that the profile specification is broken into two parts: the *Domain clause* (`DOMAIN`) defines and names sets of objects of interest (*domain sets*); and the *Utility clause* (`UTILITY`) specifies the relative values of objects contained in each domain set. The Utility clause distinguishes our notion of profiles from those typically found in publish/subscribe systems. We will explain the intuitive meaning of these profiles here as an informal means of introducing our profile language. A complete formal semantics of our profile language can be found in [10].

Profile `Traveler` might be used to drive data recharging for a traveler about to travel to Boston. Suppose the traveler wants to stay downtown. She needs to get from the airport to a downtown hotel, either by rental car or a shuttle. For a shuttle, she needs a schedule for a company offering shuttle service. For a rental car, she needs rate information for one or more rental car companies and driving directions from the airport to downtown. Even if she takes a shuttle downtown, directions serve some use as they tell her a bit about how to get about the city. She also needs information about downtown hotels, but only if she has received a shuttle schedule or directions telling her how to get to them Finally, she would like to see reviews for nearby restaurants.

The Domain clause of `Traveler` identifies 5 domain sets specified by expressions resembling inputs to a search engine.[2] As with a query, declarative specifications of domain sets free profile authors from having to locate the data

[2] As we describe elsewhere [10], *any* query-like language can be used to express domains in our profile language framework. We have chosen search engine inputs for this example for simplicity.

that interests them (the traveler may not care whether directions to a downtown hotel are generated by Mapquest [16] or found on a hotel web page), and makes the profile processor responsible for finding this data. In this profile, `RC` is a set of rental car company web pages that (presumably) offer details about rates and policies, `Sh` is a set of shuttle schedules for shuttles heading to downtown from the airport, `Di` is a set of web pages, text files etc. that give directions from the airport to downtown, `Ho` is a set of web pages for hotels located in downtown Boston, and `Re` is a set of reviews for restaurants also located in downtown Boston. The Utility clause of `Traveler` specifies 5 utility "equations" (one for each domain set) that capture the data values and dependencies previously described. Every restaurant review has a value of 1. The value of rental car web pages (`RC` objects) is *dependent* on the presence of `Di` objects in the same cache: the condition, "`[#Di > 0]`", is true if the number of `Di` objects in the cache (`#Di`) is greater than 0. This reflects the dependency of the value of rental car data on having driving directions. The value of `RC` objects also depends on how many of them appear in the cache. "`UPTO (2, 2, 0)`" specifies that the first two `RC` objects in the cache carry a value of 2, and that any more found in the cache have no value. This reflects the preference that having 2 rental car web pages in the cache is useful (so that rates and policies can be compared), but that any more than this is unnecessary. `UPTO` $(u, v, w)$ is a *threshold operator*: up to $u$ objects have a value of $v$ each, and every object beyond $u$ has a value of $w$. `Sh`, `Di` and `Ho` objects are defined similarly to `RC` objects: up to 1 shuttle schedule has value 3; up to 1 set of directions to downtown has value 1 provided that a hotel web page is in the cache; and up to 1 hotel web page has value 2, provided that either a set of directions or a shuttle schedule are also in the cache. Thus a cache consisting of two rental car web pages, one set of directions and a hotel web page would have overall value of $7\ (2+2+1+2)$, and a cache consisting of one shuttle schedule, one rental car web page and two restaurant reviews would have an overall cache value of $5\ (3+0+2)$.

The `Traveler` profile demonstrates two desirable features of profile languages: *data dependencies* and *thresholds*. In Figure 2 we show three ways that `Traveler` might be expressed in a profile language lacking these features. $T_1$ sets utility values for each domain equal to their *initial value* in `Traveler`. $T_2$ values domains according to a *ranking* of the maximum values that those domains can take according to `Traveler`. For example, `Sh` is the most valuable domain (as an object in `Sh` can have value 3), and therefore is given the highest value of 5. Finally, $T_3$ clusters dependent objects in `Traveler` into single conglomerate objects. As dependencies exist between `Ho`, `Sh`, `Di` and `RC`, they are combined into a single domain of megaobjects, with the idea that an object of one simple domain (e.g., `Ho`) can only be placed in the cache if it is accompanied by an object of each other domain in the conglomeration. The size and value of these megaobjects is their combined size and value according to `Traveler`.

| (a) Initial Values | (b) Ranking | (c) Clustering |
|---|---|---|
| <pre>PROFILE T1<br>DOMAIN<br>...<br>UTILITY<br> U (Re) = 1<br> U (RC) = 2<br> U (Sh) = 3<br> U (Di) = 1<br> U (Ho) = 2<br>END</pre> | <pre>PROFILE T2<br>DOMAIN<br>...<br>UTILITY<br> U (Re) = 2<br> U (RC) = 4<br> U (Sh) = 5<br> U (Di) = 2<br> U (Ho) = 4<br>END</pre> | <pre>PROFILE T3<br>DOMAIN<br>...<br>UTILITY<br> U ({Ho,Di,Sh,RC}) = 8<br> U (Re) = 1<br>END</pre> |

Figure 2: `Traveler` Without Dependencies, Thresholds

Table 1 shows the optimally valued cache of 5 objects according to the `Traveler` profile of Figure 1 and the three alternative profiles of Figure 2, and the "real" values of these caches according to the `Traveler` profile that includes dependencies and thresholds. Observe that both $T_1$ and $T_2$ would fill 5 object caches with what they assess to be the most valuable objects (shuttle schedules), leaving both caches with a real overall value of 3. Both profiles fail to produce a better cache because they are unaware of the threshold that limits the number of shuttle schedules that have value. Even if $T_2$ had ranked `Re` with the highest ranking (restaurant reviews are the only objects that have value no matter how many of them are placed in the cache), the value of its 5 object caches would only be 5. $T_3$'s strategy of clustering dependent objects into a single conglomerate object works reasonably well for a 5 object cache, but note that this approach can never result in a cache with differing numbers of `Ho`, `Di`, `Sh` and `RC` objects, as is needed to produce the optimally valued cache (which has 2 `RC` objects, and no more than 1 of any other).

Data dependencies and thresholds are just some of the desirable features defined in our profile language. Other features demonstrated by the `Traveler` profile include the separate specifications of the objects of interest, and the relative worth of these objects, and QoS-style weighted utility values (as opposed to priority lists). This list is not exhaustive; a comprehensive list of profile language desiderata is presented elsewhere [9], as is a formal specification of our profile language framework [10].

In practice, we expect profiles to describe many more domain sets with more complex utility value expressions than was demonstrated with the `Traveler` profile. We consider it unlikely that most users will write profiles manually (the same could be said for SQL). Instead, we expect that a profile-generation system [7] with good interfaces could support users to this end. Such a system could rely on libraries of parameterized profiles that are built and extended

| Profile | Best 5 Object Cache | Value using `Traveler` |
|---|---|---|
| `Traveler` | 1 `Sh`, 1 `Ho`, 1 `Di`, 2 `RC` | 10 |
| $\texttt{T}_1$ | 5 `Sh` | 3 |
| $\texttt{T}_2$ | 5 `Sh` | 3 |
| $\texttt{T}_3$ | 1 {`Ho,Di,Sh,RC`}, 1 `Re` | 9 |

Table 1: Best 5 Object Caches of Profiles of Figure 2

by experts. The key idea is that a *declarative* profile language is needed to facilitate this process. Finally, we recognize that profile-driven cache management will have to reconcile the data needs of thousands of users, as specified with thousands of profiles. We consider techniques for processing multiple profiles in Section 3.3.

## 3. Profile Processing

We consider three profile-driven prefetching problems: *Non-Preemptive Prefetching*, *Preemptive Prefetching*, and *Prefetching for Shared Caches*. Non-preemptive prefetching is standard prefetching where a cache is filled with a set of objects that are likely to be accessed by a client. Preemptive prefetching allows prefetching to be interrupted prior to completion. This is relevant for data recharging, where a client can "unplug" his device from a network connection at any time without warning. For preemptive prefetching, what matters is not only the cache's final contents but the order in which objects are placed in the cache, as objects placed into the cache early are the most likely to be present in a partially filled cache resulting from preemption. Prefetching for shared caches requires reconciling multiple (and potentially competing) client profiles to prefetch a cache that best meets the needs of all clients.

### 3.1. Non-Preemptive Cache Prefetching

Non-Preemptive Cache Prefetching (NCP) uses profiles to determine how to prefetch caches to achieve maximum utility value. Assume for any profile $p$, that its *value function* $f_p$, maps any set of objects (i.e., a cache) to its value according to $p$. Then the Non-Preemptive Cache Prefetching problem (NCP) is defined as follows:

**Definition 3.1 (NCP)** *Given a finite set of objects ("candidate object set") $O = \{o_1, \ldots, o_n\}$ such that $s(o_i)$ is the size of object $o_i$, a profile $p$, and a cache capacity $C$, determine subset, $O' \subseteq O$, that satisfies the constraint,*

$$\sum_{o \in O'} s(o) \leq C$$

*and that maximizes $f_p(O')$.*

NCP is a variation of the knapsack problem [20] where an object's *size* and *utility* is analagous to its *weight* and *value* using classic terminology. Unlike the knapsack problem, the value function for NCP ($f_p$) must map *sets of objects* (rather than individual objects) to values because the value of an individual object can vary according to its *context* (i.e., the other objects that have been put into the cache) due to data dependencies and thresholds. We examined the effectiveness of three standard optimization heuristics: a greedy technique, a randomized technique (simulated annealing), and branch-and-bound.

| Possible Cache Resulting From Greedy | | Cache Utility |
|---|---|---|
| (a) | $\{sh_1, ho_1, di_1, rc_1, rc_2\}$ | 10 |
| (b) | $\{sh_1, ho_1, re_1, di_1, rc_1\}$ | 9 |
| (c) | $\{sh_1, ho_1, re_1, re_2, di_1\}$ | 8 |
| (d) | $\{sh_1, ho_1, re_1, re_2, re_3\}$ | 8 |

Figure 3: Possible Caches from Applying `GREEDY`

***A Greedy Algorithm* (`GREEDY`):** The greedy algorithm for NCP (`GREEDY`) selects one object at a time to add to a cache, that adds the most value per byte to the cache of previously chosen objects. In the case of ties, `GREEDY` selects the smallest object. To demonstrate, consider the application of `GREEDY` to the `Traveler` profile in the context of data recharging. Suppose that the set of candidate objects, is the set of equally-sized objects,

$$O = \{re_1, re_2, re_3, rc_1, rc_2, sh_1, sh_2, d_1, d_2, h_1, h_2\}$$

such that $re_1$, $re_2$ and $re_3$ are restaurant reviews, $rc_1$ and $rc_2$ are rental car web pages, $sh_1$ and $sh_2$ are shuttle schedules, $d_1$ and $d_2$ are directions to Downtown and $h_1$ and $h_2$ are hotel web pages. For its first object, `GREEDY` would choose a shuttle schedule (e.g., $sh_1$) as this object adds maximal value (3) to an empty cache. Then `GREEDY` would choose a hotel web page (e.g., $ho_1$), as this object adds maximal value (2) to a cache consisting only of a shuttle schedule. `GREEDY` has a choice for the third chosen object, as both restaurant reviews and directions add 1 to the existing cache. If it chooses the latter, it will finish with the optimal cache shown in Figure 3a. If it chooses the former, then it will make the same choice with the same consequences for its fourth object and therefore finish with a suboptimal cache of value 9 (Figure 3b) or 8 (Figures 3c and 3d).

***Simulated Annealing* (SA):** Simulated Annealing (SA) is a randomized algorithm that statistically simulates the slow cooling of a physical system [22]. The algorithm works by choosing an initial state (the "current state") within a state space consisting of all possible solutions, and then performing a random walk beginning at this state. The walk consists of choosing a random neighbor, and proceeding to make this neighbor the current state if it has greater value than the current state, or with some non-zero probability that gradually decreases over time. Applied to profile-driven cache prefetching, states denote sets of candidate objects that together can fit in the given cache. The value of a state is the value of its associated set of objects according to the given profile. A move in the state space corresponds to the act of replacing objects in the associated set, with one or more drawn from the candidate object set (that fit in the cache that remains when the initial objects are removed).

The key parameters that affect the results of running simulated annealing are the initial temperature, $T$, and the conditions for decreasing the temperature. The latter can be based on a timeout, or on a degree of convergence (i.e., stopping when the difference in value between a state and its chosen neighbor is below some threshold). Because the difference in value between one cache state and its neighbor is likely to be relatively low (given that they will differ in just a few out of several hundred objects), we based changes in temperature on timeouts. We explored three different choices of initial temperature and timeouts, each of which required SA to run for roughly 5 minutes before settling on a cache's contents. Of these, we found the values produced to be close to the same for all profiles, but that setting the initial temperature to be 5, and the timeout to be 1 minute worked best in most cases. Therefore, for our experiments in Section 4, we run simulated annealing with these temperature and timeout settings.

***Branch-and-Bound* (BnB):** Branch-and-bound (BnB) searches a tree of all possible solutions in a depth-first manner, but selectively pruning the search tree whenever it is determined that an unvisited subtree cannot possibly contain a solution better than the best seen thus far. Applied to profile-driven cache prefetching, every path from the root to a leaf node in the search tree represents one possible cache of objects chosen from the candidate object set. Each node denotes a (domain, count) pair, $(D, i)$, and every path through that node represents a solution that includes the $i$ smallest objects in the candidate set that belong to domain $D$ (and no others from domain, $D$). Each level of the tree contains only nodes for some given domain,

$$(D, k), (D, k-1), \ldots, (D, 1), (D, 0)$$

such that $k$ is the largest number of objects from $D$ that can fit in the cache. While searching this tree of solutions, the branch-and-bound algorithm we implemented prunes an unvisited subtree if all paths from the root to leaf nodes of the subtree specify object sets that are guaranteed not to fit in the cache. For example, let the set of domains defined for a given profile be $\{D_1, \ldots, D_m\}$, and let the path from the root to the current node in the search tree be

$$(D_1, j_1), \ldots, (D_k, j_k).$$

If the total size of the set consisting of the $j_1$ smallest objects in $D_1$, the $j_2$ smallest objects in $D_2$, and so on up to the $j_k$ smallest objects in $D_k$ is larger than the cache capacity, then there is no need to visit any of the nodes in any subtree of $(D_k, j_k)$. A second pruning trick is based on cache value bounds. Let $D$ be any domain in the set, $\{D_{k+1}, \ldots, D_m\}$. Let $f(D, j_1, \ldots, j_k)$ be the maximum number of objects from domain $D$ that could fit in the cache containing the $j_1$ smallest objects of $D_1$, the $j_2$ smallest objects of $D_2$ etc. If the value of the cache denoted by the path,

$$\begin{aligned} P \quad = \quad & (D_1, j_1), \ldots, (D_k, j_k), \\ & (D_{k+1}, f(D_{k+1}, j_1, \ldots, j_k)), \ldots, (D_n, f(D_n, j_1, \ldots, j_k)) \end{aligned}$$

is less than the best cache value seen thus far, then the entire subtree rooted at $(D_k, j_k)$ can be pruned, as no cache represented by a path emanating from this node can exceed the value of the cache denoted by $P$.

Given the size of the search tree involved for non-trivial profiles (the number of nodes will be the product of the number of objects that can fit in a cache for each domain), branch-and-bound searching quickly becomes infeasible. To make it feasible, we terminate searching after 10 minutes. To increase the likelihood of an interrupted search finding a good solution, we use a trick from [14] whereby domains are sorted in the order generated by GREEDY, and the search tree is constructed so that the highest level of the tree contains nodes representing the most valuable domains. The search is then conducted in a postorder, rather than preorder traversal of the tree. This biases the search to favor caches with more objects from more valuable domains.[3]

## 3.2. Preemptive Cache Prefetching

Preemptive Cache Prefetching (PCP) resembles NCP, but adds the possibility that cache prefetching can be preempted prior to filling the cache. For example, for data recharging to truly resemble battery recharging, it must be possible to "unplug" a device prior to its being fully charged (i.e., before its cache is full) and still end up with useful data in the cache. This implies that not only is the set of objects chosen to fill a cache important, but so too is the *sequence* in which these objects are put in the cache given that this determines what are the potential partially-filled caches resulting from premature disconnection.

To evaluate PCP strategies, we defined a "goodness" function ($g_{p,pr}$) that scores the sequence of objects chosen by any algorithm for placement in a cache. This function is defined in terms of a profile, *p*, and a *preemption probability*, *pr*, which is the probability that cache prefetching will be interrupted prior to completion. If *pr* is 0, then prefetching always results in a full cache and PCP reduces to NCP (i.e., only the utility value of the final contents of the cache

[3] Note that BnB can only be applied to *monotonic* profiles: profiles for which the value of a cache can never decrease as a result of adding on object. We believe that non-monotonic profiles are unlikely in practice, and therefore all experiments described in Section 4 assume monotonic profiles.

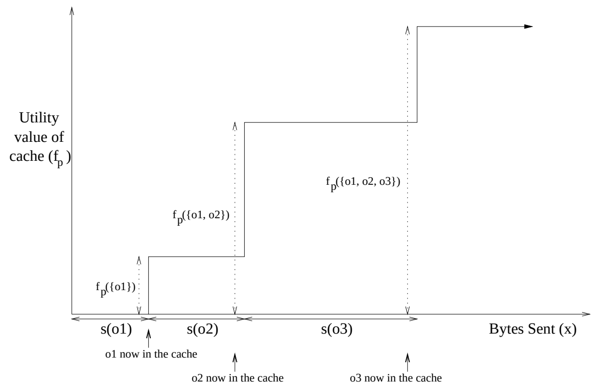


Figure 4: Bytes Sent vs Overall Cache Utility

matters in measuring the goodness of the result). On the other hand, if $pr > 0$, then goodness measures must also consider the utility value for partially filled caches.

**Definition 3.2** *[Sequence Goodness] Given a profile, $p$, a preemption probability, $pr$, and some ordering of objects, $S = \ll o_1, \ldots, o_n \gg$, the goodness of $S$ wrt $p$ and $pr$ is:*

$$g_{p,pr}(S) = e_1 + e_2$$

*such that:*

$$e_1 = (1 - pr)\ (f_p(O')),\ \textit{and}$$

$$e_2 = pr\ \left(\frac{(\sum_{i=2}^{n} s(o_i) \cdot f_p(\{o_1, \ldots, o_{i-1}\}))}{s(o_1) + \ldots + s(o_n)}\right).$$

Intuitively, $e_1$ is the value of a full cache ($f_p(O')$) times the probability that cache prefetching is not preempted ($1 - pr$), and $e_2$ is the probabilistically weighted average of values for all non-full caches, assuming that preemption is equally likely to occur after any number of bytes have been transmitted. Figure 4 illustrates how $e_2$ is calculated given the sequence of objects $\ll o_1, o_2, o_3 \gg$. The graph shown plots the value of the cache after each byte of an object is sent. Note that the curve shown in Figure 4 is stepwise linear, and further, every line has either 0 or infinite slope. This is because the value of a cache does not change while a given object is in the process of being delivered to the cache, but increases by the added value of the object once the entire object is contained in the cache. Thus, the value of the cache will be 0 until the 1st object is delivered (for bytes $1 \ldots s(o_1)$), $f_p(\{o_1\})$ until the 2nd object is delivered (for bytes $(s(o_1) + 1) \ldots (s(o_1) + s(o_2))$), $f_p(\{o_1, o_2\})$ until the 3rd object is delivered (for bytes $(s(o_1) + s(o_2) + 1) \ldots (s(o_1) + s(o_2) + s(o_3))$), and so on. Generalized over all objects that can appear in a non-full cache ($\{o_1, \ldots, o_{n-1}\}$), and weighted by the probability that prefetching will be preempted, we get $e_2$.

**Definition 3.3 (PCP)** *Given a finite set of objects $O$, such that for any object $o \in O$, $s(o)$ is its size, a profile $p$, a cache capacity $C$, and a preemption probability $pr$, determine the sequence, $S = \ll o_1, \ldots, o_n \gg$, (that is a permutation of any subset of $O$, $O' = \{o_1, \ldots, o_n\}$) that satisfies the constraint,*

$$\sum_{i=1}^{n} s(o_i) \leq C,$$

*and that maximizes the value of $g_{p,pr}(O')$.*

In Section 4.3, we compare 3 heuristic algorithms that approximate solutions to PCP. These algorithms are ordered versions of the algorithms used for NCP, in that they simply take the results that the algorithms produce for NCP and order them according to the `GREEDY` algorithm.[4]

### 3.3. Prefetching for Shared Caches

For a shared cache, there can be multiple clients (each with data requirements specified in a profile) competing to influence the contents of the cache. The profile processing algorithms described in the previous sections all assume a single profile as input, and therefore must be adapted to determine the contents of a shared cache. One way to accomplish this is to merge $n$ profiles into one *superprofile* that is representative of them all. This would require first *normalizing* individual profiles so that utility values are comparable between profiles. There are many statistical techniques that can be used to normalize utility values (e.g., z-scores [18]), and therefore we assume that profiles have already been normalized for the purpose of this discussion.

In merging individual normalized profiles, we assume that the value of an object that is of interest to multiple clients is the *sum* of values it takes in those clients' profiles.[5] This means that a superprofile can be constructed simply by appending the domain and utility equations of the individual profiles that comprise it. The value of a given object, then, would be the sum of all values it takes from utility equations in the superprofiles. This approach nicely handles the case where two profiles define domains that are either equal or intersecting, but that are named or defined differently. For example, if a profile expresses interest in Truck Rental web sites, then the Hertz web page containing information about renting cars and trucks would have a utility value which is the sum of values it takes in this profile and the `Traveler` profile (according to domain, `RC`) for any superprofile that combines them. The downside to this approach is the size of the superprofile that results - especially when the number of client profiles from which it is built is very large. We explore how well cache prefetching algorithms scale to handle superprofiles (and comprably scaled candidate object sets and cache sizes) in Section 4.4.

[4] Obviously, `GREEDY` is already ordered and need not be changed.

[5] It would be equally acceptable to *average* the values that the object takes from different profiles, but this amounts to dividing the sums by the same constant (the total number of profiles that constitute the superprofile).

| *Parameter* | *Value* | | | | |
|---|---|---|---|---|---|
| Cache Size | 64 MB | | | | |
| # Profiles | 96 | | | | |
| Domains / Profile | 20 | | | | |
| Object Size Distribution | 30% | 34% | 28% | 4% | 4% |
| | 47 KB | 95 KB | 138 KB | 207 KB | 268 KB |
| Avg Object Size | 104 KB | | | | |
| Candidate Set Cardinality | 12800 | | | | |

Table 2: Parameters for NCP & PCP

## 4. Experiments

Our experiments compare the effectiveness of `GREEDY`, `SA` and `BnB` for NCP and PCP. The algorithms were written in Java 1.3.1, and run on Linux-based Pentium III and IV workstations with 512MB of RAM.

### 4.1. Experiment Environment

**Common Parameters:** Table 2 gives the parameter settings that were constant for the unshared cache, NCP and PCP experiments we present in Section 4.2 and 4.3. We fixed the cache size to be 64 MB, as this is a typical memory size for current PDA's: the most obvious choice of device for data recharging. Each experiment ran over 96 randomly generated profiles (as described below), and each of these profiles included 20 domain definitions and accompanying utility equations. Every experiment assumed a candidate object set consisting of roughly 12800 objects (and distributed in size as specified in the table).[6] The size distribution was determined by downloading the 50 most popular web site homepages (including images and any necessary files for correct display) as rated by Media Metrix [17] for December 2001. We divided the range of sizes for these pages into five equally sized intervals, and for each interval calculated the average homepage size as well as the percentage of homepages that fell in that interval. The cardinality of the candidate object was set so that each domain set in the profile was represented and assigned an equal number of objects that fit the distribution shown in Table 2. The total size of all objects for each domain exceeded the size of cache, so that algorithms were free to choose objects based on their policy rather than object availability. Given the average size of objects of 104 KB, this amounted to a candidate set cardinality of about 12800.

**Random Profile Generation:** For these experiments, 96 monotonic profiles were randomly generated, each with 20 domains and 20 utility equations. For each profile, 20% of its utility equations are *flat* (i.e., of the form `U (A) = k`) and 80% are *non-flat* equations, which have the form,

$$\mathtt{U}\,(\mathtt{A}\,[\#\mathtt{B}\,\geq\,x])\;=\;\mathtt{UPTO}\,(u,\;v,\;w).$$

[6] Of course, this object set was modelled and not explicitly created as it would not have fit in 512 MB of memory.

Non-flat utility equations subsume utility equations with no dependencies ($x$ could be set to be 0) as well as utility equations with no thresholds ($u$ could also be set to be 0).[7]

The values of $x$, $u$, $v$ and $w$ in non-flat utility equations determine how objects of a given domain type change in value as more get added to a cache, as categorized below:

*Immediate vs Delayed Gratification:* The value of $x$ determines how quickly `A` objects acquire value as `B` objects get added to a cache. If $x$ is very small (e.g., 0 or 1), the utility equation for `A` objects exhibits *immediate gratification*, whereas larger values of $x$ (e.g., 10-15% of the size of the cache) result in utility equations with *delayed gratification*.

*Appreciating vs Depreciating:* The relative values of $v$ and $w$ determine whether or not the value of an `A` object tends to appreciate or depreciate in value as the cache contains more objects. If $\frac{v}{w} > 1$, the utility equation for `A` objects is *depreciating*. If $\frac{v}{w} < 1$, the utility equation is *appreciating*. An appreciating utility equation might be used to give value to an object that gains importance as it becomes part of a larger collection, such as emails reporting system problems (several reports of the same problem suggest that the problem is system-wide).

*Quickly vs. Slowly Accelerating:* Finally, the value of $u$ indicates whether a utility equation appreciates/depreciates in value quickly or slowly. If $u$ is very small (e.g., 1), the utility equation for `A` is quickly accelerating. If $u$ is large (e.g., 50), then the utility equation for `A` is slowly accelerating.

The 96 randomly generated profiles fall into 24 different profile "classes" of 4 profiles each, with each class distinguished by their preset values for $x$, $u$ and $\frac{v}{w}$ in their non-flat equations. Specifically, for any given class, $x$ is fixed to be either $0$, $5$, or $25$, $u$ is fixed to be either $5$ or $25$ and $v$ and $w$ are fixed to be randomly generated numbers between 0 and 100 such that $\frac{v}{w}$ is either $\frac{1}{10}$, $\frac{1}{2}$, $2$, or $10$.

### 4.2. NCP: Unshared Caches

We ran each algorithm described in Section 3 on each of the 96 randomly generated profiles and normalized the results by specifying each algorithm's score as a percentage of the score generated by `GREEDY`. Then for each profile class, we averaged the normalized scores for the four profiles from that class. The results are shown in Table 3. Note that because of normalization, an algorithm does better than `GREEDY` when its score is more than 100.

From Table 3, we can see that `GREEDY` produced the highest value caches for 14 of the 24 profile classes. `SA` produced the highest valued caches for the remaining 10 classes. Therefore, the simplest and fastest algorithm[8] was

[7] Note that ensuring that conditions are of the form, "$\#\mathtt{B} \geq x$" ensures that profiles containing these equations are monotonic.

[8] With profiles of this size, `GREEDY` completed in a few seconds, whereas `SA` ran for 5 minutes and `BnB` ran for 10 minutes.

| $x$ | $u$ | SA | BnB |
|---|---|---|---|
| 0 | 5 | 48.3 | 10.5 |
| 0 | 25 | 71.6 | 13.4 |
| 5 | 5 | 54 | 78.8 |
| 5 | 25 | 62.5 | 68.6 |
| 25 | 5 | 45.5 | 81.3 |
| 25 | 25 | 69.7 | 72.2 |

(a) Depr: $\frac{v}{w} = 10$

| $x$ | $u$ | SA | BnB |
|---|---|---|---|
| 0 | 5 | 79.1 | 52.5 |
| 0 | 25 | 84.4 | 37.7 |
| 5 | 5 | 82.8 | 72.3 |
| 5 | 25 | 90.6 | 63.1 |
| 25 | 5 | 87.5 | 80.9 |
| 25 | 25 | 96.9 | 74.2 |

(b) Depr: $\frac{v}{w} = 2$

| $x$ | $u$ | SA | BnB |
|---|---|---|---|
| 0 | 5 | 108 | 51.5 |
| 0 | 25 | 93.2 | 59.5 |
| 5 | 5 | 131.7 | 63 |
| 5 | 25 | 121.8 | 66.6 |
| 25 | 5 | 124.2 | 62.5 |
| 25 | 25 | 104.5 | 68.9 |

(c) Appr: $\frac{v}{w} = 0.5$

| $x$ | $u$ | SA | BnB |
|---|---|---|---|
| 0 | 5 | 128.1 | 72.7 |
| 0 | 25 | 94.9 | 79.2 |
| 5 | 5 | 207.4 | 78.6 |
| 5 | 25 | 107.2 | 83.3 |
| 25 | 5 | 128.9 | 70.3 |
| 25 | 25 | 105.2 | 77 |

(d) Appr: $\frac{v}{w} = 0.1$

Table 3: Normalized NCP Scores: Depreciating and Appreciating Utility Equations

also the most effective, dispelling the notion that expressive profiles require complex processing to be effectively used. As can be seen in Table 3, SA beat GREEDY when the profiles processed were predominantly composed of *appreciating* utility equations (Table 3c and d). In fact, SA beat GREEDY in 10 of the 12 profile classes of this form. SA did especially well when $u$ was small, producing the best cache for all 6 classes of appreciating profiles with $u = 5$.

SA's relative success with appreciating profiles can be explained as follows. GREEDY, by nature, performs no lookahead and thus it will never choose to place an object from a low-valued domain in the cache, even if objects from that domain can become more valuable as the cache gets filled with more objects. But for appreciating profiles, the best caches will often be those that contain objects from domains that initially have low value, but that increase in value as more of them are put into the cache. Whereas GREEDY will never choose these low-valued objects, with some nonzero probability, SA will select a random cache that includes enough of these low-valued objects that they take on a higher value. When $u$ is small, the number of objects required to be in the cache before these objects increase in value is also small, and the probability of randomly choosing a cache with $u$ of these objects increases. Hence, SA consistently produces better quality caches than GREEDY when $u$ is small and utility equations are predominantly appreciating. It should be pointed out that GREEDY's poor performance with profiles with appreciating utility equations also had to do with the way we chose random values for $v$ and $w$. For both appreciating and depreciating utility equations, we first chose a random value between 50 and 100, and this became the value of what was supposed to be the larger of $v$ and $w$ ($v$ for depreciating equations and $w$ for appreciating equations). The value of the smaller variable was then calculated using the ratio, $\frac{v}{w}$, set by the profile class. Therefore, for appreciating utility equations such that $\frac{v}{w} = 0.1$, $v$ is never greater than 10, and for appreciating utility equations such that $\frac{v}{w} = 0.5$, $v$ is never greater than 50. Given that values for flat utility equations were random values between 0 and 100, with high probability the values for these objects were greater than $v$, and therefore, these objects were far more likely to be chosen by GREEDY.

It is also interesting to note the poor performance of Branch-and-bound (BnB), having the lowest scores for 20 of the 24 profile classes. BnB does poorly at profile-driven prefetching because it doesn't scale: the search tree that the algorithm has to traverse has a height of 20 (one level per domain), and a degree of 640 (one per object in the candidate set belonging to any given domain). Therefore, the search tree consists of $640^{20}$ nodes! After 10 minutes (at which point we cut off its search), BnB was never left with a cache with objects from more than 3 domains. Therefore, we did not include BnB in our algorithm analysis for superprofiles (Section 4.4).

## 4.3. PCP: Unshared Caches

Table 4 shows "goodness" calculations (Definition 3.2) for GREEDY, SA and BnB over all 24 profile classes assuming preemption probabilities of $pr = 0.9$, (likely preemption), $pr = 0.5$, (equally likely preemption and completion), and $pr = 0.1$, (unlikely preemption). As with the NCP results, we have normalized the goodness values by expressing them as percentages of the goodness values achieved by GREEDY. The table shows, for each profile class, the average of these normalized values over the the four profiles belonging to that class. Once again, GREEDY produces better caches for more profile classes than SA and BnB, producing the best caches for $17/24$ profile classes when the probability of preemption is high, for $15/24$ profile classes when the probability of preemption is moderate, and for $14/24$ profile classes when the probability of preemption is low.

By comparing tables 3 and 4, it becomes clear that the algorithm that does best for NCP usually does best for PCP when given the same parameter settings. The only four cases where this is not the case are listed below:

$$
\begin{array}{lllll}
(1) & x = 25, & u = 25, & \frac{v}{w} = 0.1, & \text{and } pr = 0.9 \\
(2) & x = 25, & u = 25, & \frac{v}{w} = 0.1, & \text{and } pr = 0.5 \\
(3) & x = 25, & u = 25, & \frac{v}{w} = 0.5, & \text{and } pr = 0.9 \\
(4) & x = 5, & u = 25, & \frac{v}{w} = 0.1, & \text{and } pr = 0.9
\end{array}
$$

where for each, SA has produced the best result for NCP and GREEDY has produced the best result for PCP. Figure 5 illustrates what is happening in these cases. The graph in this figure shows how GREEDY, SA and BnB give value to a cache over "time"[9] for the case where $x = 25$, $u = 25$, and $\frac{v}{w} = 0.1$. Note how SA produces the highest value cache once the cache is filled, but that GREEDY usually produces the best partially filled caches,[10] thereby confirming the anomalous results.

The explanation for cases (1)-(3) is as follows: for each of these cases, $x = 25, u = 25$ and the profiles are appreciating. For appreciating profiles, when $u$ and $x$ are large, it likely will take a while (i.e., many objects placed in the cache) before a random algorithm produces a cache with sufficient numbers of objects of domains with appreciating utility values to give high value to those objects. Therefore, what is common to these cases is that SA produces a cache that has relatively low value for a long time, and then increases in value suddenly, overtaking the value of the cache produced by GREEDY by the time the cache is full. Because the value of the cache produced by SA was low for so long, the area under the curve is less for SA than for GREEDY. Case 4 is more subtle. Note that for $x = 5$, $u = 25$, $\frac{v}{w} = 0.1$ and $pr = 0.9$, SA beats GREEDY for NCP but loses to GREEDY for PCP. On the other hand, if $x = 25$ (and all other parameters are the same), SA beats GREEDY for both NCP and PCP. When $x = 5$ instead of $x = 25$, it takes fewer objects in the cache to give high value to those with appreciating utility values. Thus, one might think that with $\frac{v}{w} = 0.1$, an early and sharp increase in object value would lead SA to beat GREEDY for PCP. But in fact the opposite is true: GREEDY beats SA for PCP in this case, but loses to SA when $\frac{v}{w} = 0.5$ (i.e., when the increase in object value occurs at the same time, but to a far lesser degree). The explanation for this phenomenon again lies with the way we generate random values for $v$ and $w$. Always, our technique is to first generate a random value between 50 and 100, assign it to the variable that is supposed to be larger (for appreciating equations, $w$), and set the value of the other variable according to the value of $\frac{v}{w}$ associated with the profile class. Note that when $\frac{v}{w} = 0.1$, this means that $v$ will always be less than or equal to 10, but when $\frac{v}{w} = 0.5$, $v$ can be as high as 50. Therefore, when $\frac{v}{w} = 0.1$, the initial values of the cache produced by SA are **so low**, that when $pr$ is large (thus favoring early values of the cache in calculating goodness), even a sharp increase in object value is not enough to make SA beat GREEDY for PCP. On the other hand, when $\frac{v}{w} = 0.5$, initial values of the cache are large enough that SA beats GREEDY at PCP.

The key lessons learned from our experiments for NCP and PCP is that despite the apparent complexity of profiles that include data dependencies and thresholds, a fairly simple and fast algorithm (GREEDY) works quite well in processing such profiles to decide what to prefetch into a cache. Further, the cases where SA produces a better cache than GREEDY (profiles with predominantly appreciating utility equations) are arguably, unlikely to occur in practice.

[9] Time is measured by "MB in the cache" to equalize data transfer rates.
[10] The corresponding graphs for cases 3 and 4 exhibit the same property.

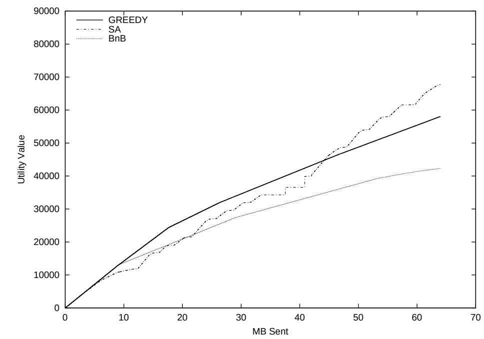

Figure 5: Partial Cache Values ($x = 25, u = 25, \frac{v}{w} = 0.1$)

## 4.4. NCP: Shared Caches

As we pointed out in Section 3.3, prefetching algorithms that assume a single profile and an unshared cache can be adapted to work with multiple profiles and a shared cache by merging the multiple profiles into a single superprofile that is representative of them all. The merged profile can then be used as input to each of the prefetching algorithms described in Section 3.

One complication brought on by profile combination is the fact that different profiles may define domains that intersect or that are even equivalent, even though they are defined differently. In other words, for superprofiles we must assume that any given object can belong to multiple domains.[11] This means that GREEDY must be adapted so that it always chooses an object that gives the most added value to the cache according to **all** of the utility equations that give it value. (SA, because it chooses objects randomly, need not be altered.)

We refer to the greedy algorithm that allows for objects to belong to multiple domains as GREEDY$_{sup}$. GREEDY$_{sup}$ maintains 3 data structures as it uses an input superprofile to determines the contents of a cache:

- $B$: a Boolean *membership* matrix that specifies all of the domains to which each object belongs. Thus, $B$ is an $n \times m$ matrix such that $n$ is the size of the candidate object set, $m$ is the number of domains defined by the superprofile, and $B[i, j] = 1$ iff object, $o_i$, belongs to domain, $D_j$;

- $V$: an *added value* vector that specifies for each of the $m$ domains, the change in value that would result from adding an object from the domain to the current cache. Note that unlike the membership matrix which can be computed statically, the added value vector must be recomputed every time an object is added to the cache; and

[11] For single profiles it is more reasonable to assume disjoint domains.

| $(x, u)$ | pr = 0.9 SA | pr = 0.9 BnB | pr = 0.5 SA | pr = 0.5 BnB | pr = 0.1 SA | pr = 0.1 BnB |
|---|---|---|---|---|---|---|
| (0, 5) | 60.3 | 12.5 | 53.7 | 11.7 | 49.3 | 11.2 |
| (0, 25) | 84.0 | 15.8 | 78.3 | 14.9 | 74.4 | 14.3 |
| (5, 5) | 61.7 | 82.7 | 58.0 | 80.5 | 55.5 | 79.1 |
| (5, 25) | 70.5 | 73.8 | 66.0 | 71.2 | 63.0 | 69.5 |
| (25, 5) | 53.6 | 85.7 | 49.6 | 83.5 | 47.0 | 82.1 |
| (25, 25) | 72.4 | 77.5 | 71.2 | 74.7 | 70.4 | 72.9 |

Depr: $\frac{v}{w} = 10$

| $(x, u)$ | pr = 0.9 SA | pr = 0.9 BnB | pr = 0.5 SA | pr = 0.5 BnB | pr = 0.1 SA | pr = 0.1 BnB |
|---|---|---|---|---|---|---|
| (0, 5) | 84.0 | 56.7 | 81.3 | 53.8 | 79.5 | 51.9 |
| (0, 25) | 90.3 | 42.1 | 87.1 | 39.8 | 85.0 | 38.3 |
| (5, 5) | 81.6 | 78.2 | 82.4 | 74.9 | 82.8 | 72.8 |
| (5, 25) | 88.3 | 68.3 | 90.2 | 65.0 | 91.5 | 62.8 |
| (25, 5) | 85.5 | 88.7 | 87.8 | 84.7 | 89.3 | 82.2 |
| (25, 25) | 95.9 | 82.5 | 98.1 | 78.9 | 99.6 | 76.7 |

Depr: $\frac{v}{w} = 2$

| $(x, u)$ | pr = 0.9 SA | pr = 0.9 BnB | pr = 0.5 SA | pr = 0.5 BnB | pr = 0.1 SA | pr = 0.1 BnB |
|---|---|---|---|---|---|---|
| (0, 5) | 113.5 | 61.1 | 110.3 | 55.3 | 108.4 | 52.0 |
| (0, 25) | 96.2 | 68.0 | 94.5 | 63.0 | 93.5 | 59.9 |
| (5, 5) | 121.7 | 69.2 | 128.2 | 65.1 | 132.2 | 62.5 |
| (5, 25) | 111.6 | 75.0 | 118.8 | 70.6 | 123.5 | 67.9 |
| (25, 5) | 116.6 | 72.8 | 121.5 | 66.6 | 124.5 | 63.0 |
| (25, 25) | 98.1 | 77.0 | 102.6 | 72.9 | 105.5 | 70.3 |

Appr: $\frac{v}{w} = 0.5$

| $(x, u)$ | pr = 0.9 SA | pr = 0.9 BnB | pr = 0.5 SA | pr = 0.5 BnB | pr = 0.1 SA | pr = 0.1 BnB |
|---|---|---|---|---|---|---|
| (0, 5) | 118.8 | 77.4 | 124.9 | 74.9 | 128.9 | 73.2 |
| (0, 25) | 91.2 | 84.4 | 93.9 | 81.8 | 95.7 | 80.2 |
| (5, 5) | 181.5 | 82.0 | 200.5 | 79.7 | 213.0 | 78.2 |
| (5, 25) | 95.8 | 87.4 | 103.1 | 85.4 | 108.0 | 84.1 |
| (25, 5) | 114.1 | 75.6 | 124.0 | 73.1 | 130.4 | 71.4 |
| (25, 25) | 91.6 | 81.3 | 99.8 | 79.1 | 105.2 | 77.6 |

Appr: $\frac{v}{w} = 0.1$

Table 4: Normalized PCP Scores: Depreciating and Appreciating Utility Equations

- $S$: a *size* vector that specifies the size of each of the $n$ objects in the candidate object set.

`GREEDY`$_{sup}$ begins by initializing the membership matrix, added value vector (relative to an empty cache) and size vector. It then chooses one object at a time to add to the cache, by following the steps below:

1. First, it computes the $n$ element vector, $C_1 = B \cdot V$. For any object, $o_i$, $C_1[i]$ is the value that object would add to the current cache if it were inserted (without replacing any existing object in the cache).

2. Next, it computes the $n$ element vector, $C_2 = C_1/S$. For any object, $o_i$, $C_2[i]$ is the value per byte that object would add to the current cache.

3. Next, it finds the maximum value in $C_2$, $C_2[i]$, and designates object $o_i$ as the next object to add to the current cache.

4. Next, it updates the added value vector to reflect domain values relative to the new cache.

5. Finally, it removes column $i$ from $B$ to reflect that object, $o_i$, is no longer eligible to be added to the cache.

This process is repeated until the candidate object set contains no more objects that can fit in the remaining cache and that add value to the cache. Note that `GREEDY`$_{sup}$ is far more expensive than `GREEDY` (which simply cycled through the *added value* vector looking for the domain which added the most value to the cache), as it requires a matrix-vector multiplication be performed for every object chosen. We use some tricks to reduce the cost of `GREEDY`$_{sup}$. For one, we partition the set of candidate objects into *classes*, where objects in a given class belong to the same set of domains. This allows us to reduce the size of one dimension of the Boolean matrix (from the number of objects in the candidate object set to the number of classes). Also, every time an object is added to the cache, we keep track of which domains have their values affected. For example, in the case of the `Traveler` profile of Figure 1, adding a `Di` object to an empty cache only changes the values for domains `RC`, `Di` and `Ho`. Any class whose objects do not belong to any of the affected domains need not be included in the matrix calculation (objects of such classes would add the same value to the new cache as they did to the previous cache). While these tricks reduced the cost of `GREEDY`$_{sup}$ somewhat, our experiments showed that this new version of `GREEDY`$_{sup}$ still does not scale. We ran `GREEDY`$_{sup}$ assuming superprofiles of increasing size (with candidate object sets, classes and cache sizes increasing proportionally in the size of the superprofile) and recorded the time `GREEDY`$_{sup}$ required to determine a cache. We also measured the value of the cache constructed by `GREEDY`$_{sup}$ and compared it to that produced by `SA` which was allowed to run for 30 minutes (by setting its temperature to 30 and its timeout to 1 minute). The results are shown in Table 5. Each row in the table shows a run of `GREEDY`$_{sup}$ with a superprofile of size, $n$, where $n$ denotes the number of single profiles merged to construct the superprofile. Single profiles are chosen ran-

| Profiles in Superprofile | $\mathsf{GREEDY}_{sup}$ | SA | SA Normalized Cache Value |
|---|---|---|---|
| 25 | 1.1 sec | 30 min | 73.5 |
| 50 | 13.4 min | 30 min | 71 |
| 75 | 73 min | 30 min | 85 |
| 100 | 210 min | 30 min | 84 |

Table 5: Execution of $\mathsf{GREEDY}_{sup}$ and SA on Superprofiles

domly from the 96 randomly generated profiles described in Section 4.1. All other parameters were determined as functions of $n$. The number of object classes was set to $40n$ (i.e., twice the number of domains), under the assumption that domains in a superprofile can usually be clustered into disjoint sets of related topics, and that objects from the candidate set that belonged to multiple domains would belong to domains from a single cluster. As the number of disjoint clusters is bounded by the number of domains, we set the number of classes to be twice the number of domains (also accounting for objects that belonged to a single domain). Each object class was assumed to contain 5000 objects, distributed in size according to the distribution shown in Figure 2. The cache size was set to be to the maximum of $100n$ MB and $8$ GB. 100 MB per client is widely considered to be a useful metric for determining the size of a shared web proxy cache [13]. $8$ GB is the size of the shared web proxy cache employed at Brandeis (serving 3000 clients). And the average number of domains that classes included was set to 2. Given that there are $20n$ domains in all, this means that for any class, $i$ and Domain, $D_j$, that $B[i,j] = 1$ with probability, $\frac{1}{10n}$.

As Table 5 shows, $\mathsf{GREEDY}_{sup}$ scaled to $n = 75$ before its execution time took more than 3 hours. On the other hand, SA (which always ran in 30 minutes) produced competitive results with $\mathsf{GREEDY}_{sup}$, suggesting that a randomized approach is better suited for large scale cache prefetching problems. This raises the possibility of two-tiered prefetching: data recharging might use GREEDY to decide what to prefetch into an individual device cache, but SA to manage the shared cache from which multiple data recharging clients draw.

## 5. Related Work

Our notion of profiles is unique in that it combines a language for specifying predicates over data items with a rich language for specifications of the user's preferences, priorities, and requirements. User profiles for Web-based applications (such as Yahoo, PointCast [21], etc.) are typically fairly simple, allowing the user to specify particular categories of information (*channels*) that they are interested in receiving. This approach to building information dissemination systems (*publish/subscribe* [19]) typically uses a *walled garden* approach in which the data that can be delivered to the user is restricted to that found on specific content sites. Most systems allow simple, channel-specific predicates to identify channels of interest to users (e.g., to specify particular companies for stock prices). The Grand Central Station system [15] developed at IBM Almaden provides a more general form of predicates over its channels, and is therefore closer to our notion of the domain specification portion of a profile.

User profiles for text-based data have been extensively investigated in the context of Information-Filtering and Selective Dissemination of Information research [12]. The systems in these areas use techniques from the Information Retrieval (IR) world for filtering unstructured text-based documents [4]. In general, IR profile systems use either a Boolean model or a similarity based model. In the Boolean model a user profile is constructed by combining keywords with boolean operators (e.g., And, Or, Not), and an "exact match" semantics is used — a document either satisfies the predicate or not. Similarity-based models use a "fuzzy match" semantics in which the profiles and documents are assigned a similarity value. A document whose similarity to a profile is above a certain threshold is said to match that profile. The Stanford Information Filtering Tool (SIFT) [23], is a well-known content-based text filtering system for Internet News articles. With the advent of XML, filtering of web-documents based on structure as well as content has become more feasible. The XFilter system [3] is a recent example of such a filtering system. XFilter, however, has no notion of variable utility as exists in our profile model.

One recent effort that has investigated the representation of variable utility is the user preference framework proposed by Agrawal and Wimmers [1]. This model allows users to specify numeric weights for entities or sets of entities and provides a well-defined mechanism for combining sets of preferences. This model, however, provides no notion of thresholds or dependencies and this work did not address the performance and scalabablity of algorithms for exploiting the preference information.

All of the work discussed above was focused on improving the relevance of data returned by search or delivered to users via push-based dissemination. Profile information has also been used in a limited way to direct the management of caches. Early work in this area was the "quasi-caching" system of Alonso et al.[2], in which specifications of user requirements in terms of data quality were used to reduce the amount of data sent from a server to update client caches. More recently, a cache maintenance technique that exploits user specifications of preferences in terms of the tradeoff between latency and recency were presented in [5]. Due to the emphasis on cache consistency in this work, the language for describing user preferences in these systems is limited to specifying quality constraints. Our notion of profiles is focused on priorities and dependencies amongst items, but can be extended to incorporate such quality constraints. An alternative approach to maintaining the recency of cache contents is the notion of "data freshening" [11]. Rather than determining update priorities based on user preferences, this work determines the refresh rates based on how often the underlying data elements change. Recent work [6] has used user profiles to achieve more effective freshening policies.

## 6. Conclusions

We propose a simple profile language that permits high-level expression of a user's data needs for the purpose of expressing desirable contents of a cache. We consider techniques for cache prefetching on the basis of profiles expressed in our framework, both for basic and preemptive prefetching, the latter referring to the case where staging a cache can be interrupted at any point without prior warning. We examine the effectiveness of three techniques in particular: a greedy approach, a randomized approach (simulated annealing), and branch-and-bound, and show that the greedy approach is fast and effective for moderately sized profiles, but that the randomized approach scales better for shared caches with extremely large superprofiles. We view this work as a migration of data management ideas and techniques, characterized by their use of data and application semantics to drive the management of shared resources, into the wide-area network setting. We believe that profiles, as described in this paper, are an enabling technology that could spawn a rich new area of research beyond cache management into network data management in general. The use of rich, declaratively specified profiles to drive data management policy promises benefits similar to the profound effect that declarative queries had on database system technology. This paper is a first step in that direction.


## Acknowledgements

We would like to thank David Brooks and Ajay Daptardar for implementing the algorithms presented in this paper.


## References


[1] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 297–306. ACM, 2000.

[2] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *TODS*, 15(3):359–384, 1990.

[3] M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 53–64. Morgan Kaufmann, 2000.

[4] N. J. Belkin and W. B. Croft. Information filtering and information retrieval: Two sides of the same coin? *Communications of the ACM, December 1992*, 35(12):29–38, December 1992.

[5] L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the web. In *VLDB 2002, Proceedings of the 28th International Conference on Very Large Databases, August, 2002, Hong Kong, China*, 2002.

[6] D. Carney, S. Lee, and S. Zdonik. Scalable application-aware data freshening. In *Proceedings of 18th International Conference on Data Engineering (ICDE)*, 2002. To appear.

[7] U. Cetintemel, M. J. Franklin, and C. L. Giles. Self-adaptive user profiles for large-scale data delivery. In *Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*, pages 622–633. IEEE Computer Society, 2000.

[8] W. Chen, J. F. Naughton, and P. A. Bernstein, editors. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29. ACM, 2000.

[9] M. Cherniack, M. J. Franklin, and S. Zdonik. Expressing user profiles for data recharging. *IEEE Personal Communications: Special Issue on Pervasive Computing*, August 2001.

[10] M. Cherniack, M. J. Franklin, and S. Zdonik. Profile-driven data management. Technical report, Brandeis University Department of Computer Science, December 2001.

[11] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In Chen et al. [8], pages 117–128.

[12] P. W. Foltz and S. T. Dumias. Personalized information delivery: an analysis of information filtering methods. *Communications of the ACM, December 1992*, 35(12):51–60, December 1992.

[13] Isaserver faq. http://isaserver.org.

[14] J. J.Burg, J. Ainsworth, B. Casto, and S.-D. Lang. Experiments with the 'oregon trail knapsack problem'.

[15] Q. Lu, M. Eichstaedt, and D. A. Ford. Efficient profile matching for large scale webcasting. In *7th International WWW Conference, April 1998*, 1998.

[16] http://www.mapquest.com.

[17] http://www.mediametrix.com.

[18] D. A. Menasc and V. A. Almeida. *Capacity Planning for Web Performance*. Prentice Hall, 1998.

[19] B. M. Oki, M. Pflgl, A. Siegel, and D. Skeen. The information bus - an architecture for extensible distributed systems. *Symposium on Operating Systems Principles (SOSP)*, pages 58–68, 1993.

[20] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[21] S. Ramakrishnan and V. Dayal. The pointcast network. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, page 520. ACM Press, 1998.

[22] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, 1991.

[23] T. W. Yan and H. Garcia-Molina. The sift information dissemination system. *TODS*, 24(4):529–565, 1999.