UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# Adaptive Caching by Experts

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Robert B. Gramacy

March 2003

The Thesis of Robert B. Gramacy is
approved:

Professor Manfred K. Warmuth, Chair

Professor Scott A. Brandt

Professor David P. Helmbold

Frank Talamantes
Vice Provost & Dean of Graduate Studies

# Contents

# List of Figures

# List of Tables

# Adaptive Caching by Experts

*Robert B. Gramacy*

## ABSTRACT

We are constructing caching policies that have 15-22% lower miss rates than the best of twelve baseline policies over a large variety of request streams. This represents an improvement of 45–70% over Least Recently Used, the most commonly implemented policy. We achieve this not by designing a specific new policy, but by using on-line Machine Learning algorithms to develop a master policy, which dynamically combines the recommendations of a pool of standard policies based on their observed success. The framework outlined in this paper is a paradigm shift for the design of caching strategies. Our approach is attractive because it is simple, adaptive, scalable, and gives impressive results. A thorough experimental evaluation of our techniques is given, as well as a discussion of what makes caching an interesting on-line learning problem.

## Acknowledgments

Thanks to Manfred K. Warmuth and Scott A. Brandt for focus, encouragement, patience, helpful suggestions, some great ideas, and financial support. Dave P. Helmbold helped out tremendously by providing an efficient recurrence for BestShifting$(K)$, and other such insights which alluded us. Thanks to Ismail Ari, Jonathan Panttaja, and Clayton Bjorland, (the Evolutionary Game Theory Group) for helping to get the project started. Leah Johnson helped a lot with editing and prose, and her love and patience is always much appreciated. I can’t forget to thank my family. Where would I be without them? Finally, my thanks goes out to the UCSC Storage Systems group (Ahmed Amer, Ismail Ari, Scott A. Brandt, Darrel D. E. Long, and Ethan L. Miller) for great data, a healthy pool of baseline policies, constructive comments, and criticism.

# 1. Introduction

Caching is ubiquitous in operating systems. It is useful whenever we have a small, fast main memory and a larger, slower secondary memory. For example: In file system caching the secondary memory is a hard drive or a networked storage server, while in web caching the secondary memory is the Internet. The goal of caching is to keep within the smaller memory data objects (files, web pages, etc.) from the larger memory which are likely to be accessed again in the near future. Since the future request stream is not generally known, heuristics, called *caching policies*, are used to decide which objects should be discarded as new objects are retained. If a requested object already resides in the cache then we call it a *hit*, corresponding to a low-latency data access. Otherwise, we call it a *miss*, corresponding to a high-latency data access, as the data is fetched from the slower secondary memory into the faster cache memory. In the case of a miss, room must be made in the cache memory for the new object. To accomplish this, a caching policy discards objects from the cache which it thinks will cause the fewest or least expensive future misses.

## 1.1 Caching Policies

In this work we consider twelve baseline policies including seven common policies (RAND, FIFO, LIFO, LRU, MRU, LFU, and MFU), and five more recently developed and very successful policies (SIZE and GDS [9], GD* [17], GDSF and LFUDA [2]). These algorithms employ a variety of directly observable criteria including recency of access, frequency of access, size of the objects, cost of fetching the objects from secondary memory, and various combinations of these. Table 1.1 roughly groups these twelve policies by the criteria they exploit.

For our discussion in this paper, any collection of caching policies can be used, and their replacement criteria can be of any sort. We do require that a caching policy fulfill

| **criteria** | **algorithm** |
|---|---|
| – | RAND |
| arrival order | FIFO, LIFO |
| arrival time | LRU, MRU, GDS, GDSF, LFUDA, GD* |
| request freq | LFU, MFU, GDSF, LFUDA |
| object size | SIZE, GDS, GD*, |
| retrieval cost | GDS, GDSF, GD* |

Table 1.1: A characterization of cache replacement policies used in this paper.

one obligation: that it maintain priorities over the objects it caches. Low priority objects are the next to be discarded, and high priority objects should be last. Policies do not have to be deterministic. That is, objects can be discarded randomly. However, priorities should indicate which objects are preferred, and which objects are expendable. A completely random policy would assign the same priority to every object.

## 1.2 Choosing A Policy

The primary difficulty in selecting the best policy lies in the fact that each of these policies may work well in different situations or at different times due to variations in workload, system architecture, request size, type of processing, CPU speed, relative speeds of the different memories, load on the communication network, etc. Thus the difficult question is: In a given situation, which policy should govern the cache? For example, the request stream from disk accesses on a PC is quite different from the request stream produced by web-proxy accesses via a browser, or that of a file server on a local network. The relative performance of the twelve policies may vary greatly depending on the application. Furthermore, the characteristics of a single request stream can vary temporally for a fixed application. For example, a file server can behave quite differently during the middle of the night while making tape archives in order to backup data, whereas during the day its purpose is to serve file requests to and from other machines and/or users. Because of their differing decision criteria, different policies perform better given different workload characteristics. The request streams become even more difficult to characterize when there is a hierarchy or a network of caches handling a variety of file-type requests. In these cases, choosing a fixed policy for each cache in advance is doomed to be sub-optimal.

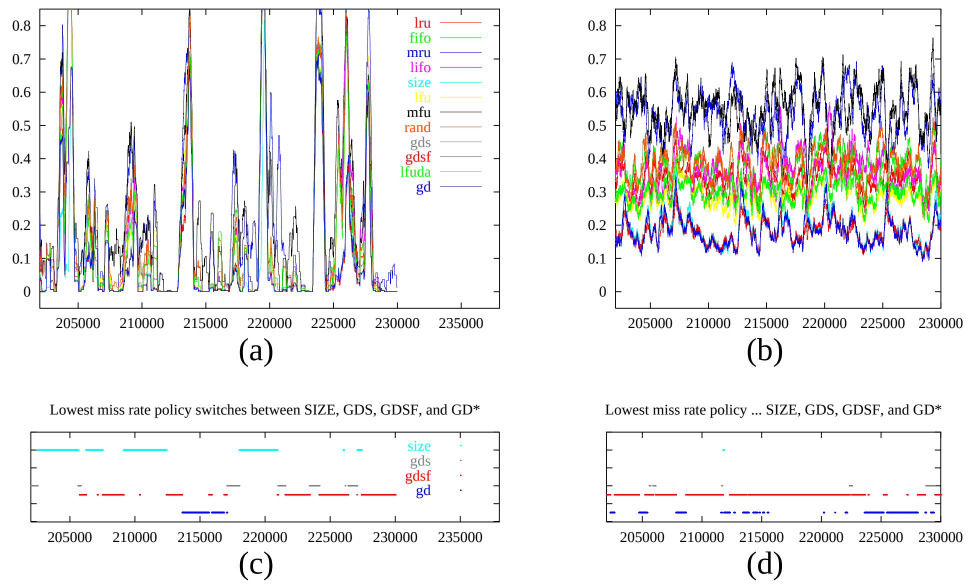

Figure 1.1: Miss rates ($y$ axis) of (a) the twelve fixed policies (calculated w.r.t. a window of 300 requests) over 30,000 requests ($x$ axis), (b) the same policies on a random permutation of the data set, (c) and (d) the policies with the lowest miss rates in the figures above. (Snapshot of UMo dataset, see Section 6.1.)

The usual answer to the question of which policy to employ is either to select one that works well on average, or to select one that provides the best performance on some important subset of the workload. However, these strategies have two inherent costs. First, the selection (and perhaps tuning) of the single policy to be used in any given situation is usually done by hand. This may be both difficult and error-prone, especially in complex system architectures with unknown and/or time-varying workloads. Second, the policy with the best common case performance may in fact be worse than what is achievable by another policy at any particular moment. Figure 1.1(a) shows the hit rate of the twelve policies described above on a representative portion of one of our data sets (UMo, described later in Section 6.1) and Figure 1.1(b) shows the hit rate of the same policies on a random permutation of the request stream. While cluttered, these figures show that the miss rates on the permuted data set are quite different, and typically higher than those of the original data set. Figures 1.1(c) and (d) show which policy is best at each instant of time for the data segment and its permuted counterpart. It is clear from these (representative) figures that the best policy changes over time.

To avoid the perils associated with trying to hand-pick a single policy, one would like to be able to automatically and dynamically select the best policy for any situation. In other words, one wants a cache replacement policy which is “adaptive”. In our Storage Systems Research Group, we have identified the need for such a solution in the context of complex network architectures and time-varying workloads and suggested a preliminary framework in which a solution could operate [1] (see Section 2.1.3). However, this preliminary work lacked concrete algorithmic solutions to the adaptation problem. This paper presents a complete adaptive caching framework, together with off-line comparators, competitive policies designed explicitly for varying workload characteristics, and experimental results.

Rather than develop a new caching policy (well-plowed ground, to say the least), this paper uses a *master policy* to dynamically determine the success rate of all baseline policies and vary both its replacement criteria and fetching behavior based on their relative performance on the workload. We show that with no additional fetches, this policy works about as well as the best fixed policy, chosen *a posteriori*. We define a *refetch* as a fetch of a previously seen object that is favored by the current policy but was discarded from the real cache at some time prior. Refetching is like prefetching [23, 8, 13], except that the refetched objects have recently been discarded. With refetching, the master policy can outperform the best fixed policy. In particular, when all required objects are refetched continuously, it has a 15-23% lower miss rate than the best fixed policy, and almost the same performance as the best possible partition into segments and assignments of governing policies. For reference, when compared with LRU, this policy has a 45-70% lower miss rate. To achieve the same miss rate as our master policy, an LRU cache of five to six times larger would be needed. Disregarding misses on objects never seen before (*compulsory* misses), the performance improvements are even greater. Because refetches are potentially costly, it is important to note that they can be done in the background. Our preliminary experiments show this to be both feasible and effective, capturing most of the advantage of continuous refetching.

The aim of a caching policy is usually twofold: (1) to reduce the end-user latency for requested objects, and (2) to keep the total number of system I/O's low. Refetching can increase the amount of I/O's performed by the system. However, with wise choices of which objects to refetch and when, it is still possible to obtain a reduction in end-user latency (miss rate) below that of the best fixed policy, while keeping the total number of I/O's low. In fact, the total number of I/O's can often also be kept below that of the best fixed policy. Our master policies always have far fewer I/O's than LRU.

# 2. Related Work

Like many on-line algorithms, the more recently developed caching policies strive for "adaptivity". However, researchers often disagree about what adaptive really means, and how it can be measured. Rather than survey the vast literature of attempts at developing more "adaptive" caching policies, we refer the reader to our references. The newer of the twelve policies used in this paper (see Section 1) juggle multiple *aging* criteria which trade off recency, frequency, size, etc. in order to make replacements. While none of these policies' parameters are ever explicitly altered, their profound improvement over traditional static policies, like LRU and LFU, suggests that their criteria are in some sense more "adaptive". Megiddo, et. al. [21] survey many of the prominent policies for adaptive *paging*[1], adding their own into the mix. In contrast to many previous caching and paging policies, theirs *does* involve a parameter that is explicitly adapted. Instead, this section focuses on introducing the *Expert Framework*, which our paper exploits, and whose degree of adaptivity can, in a certain sense, be quantified.

Informally, *experts* are decision making or prediction automata. They can be predictors, algorithms, heuristics, protocols, etc. Their actions can be static or dynamic in time, and/or functions of state or particular inputs. Most often, experts are heuristics for predicting responses, reacting to events, and/or altering systems. An expert's success is measured by *loss*. Loss is a quantification of the discrepancy between the actions, or predictions of an expert, and what is currently judged to be an optimal response. In the *Expert Framework* [16, 11, 20, 10, 6] a *master* algorithm enlists the advice of experts, and observes their losses, in order to to make its own predictions, decisions, or actions. The master algorithm is evaluated by the same loss functions. Ideally, its loss is smaller, or close to, the loss of its best experts.

To our knowledge Avrim Blum [3] (1996) was the first author to suggest using

[1] Paging and caching are quite similar, except that all objects cached (or *pages*) have uniform size.

caching policies (actually, page-replacement policies) as experts to develop a master paging strategy. In this work Blum also identified the need for modifications to the Expert Framework for the situation where the experts, like caching policies, make decisions which irreparably alter the state of a system.

Despite what to us seems like an obvious application domain, very few people have applied the Expert Framework to operating systems problems. The few that are known to us are introduced in this section, following a brief overview of the assumptions made by the framework, how it operates, and what can be proven about it. The first application helps determine appropriate spin-down times for hard disks in mobile computers. Informed choices about when to spin down the hard disk can dramatically increase battery life, and save energy. The second application is paging, and is of largely theoretical interest. Finally, we briefly mention our preliminary work in applying the Expert Framework to caching: showing how experts can be used to find the best policy for nodes in a distributed cache, and how to track the best policy when it changes over time.

## 2.1 Expert Framework and Online Learning

Littlestone and Warmuth [20] introduced the Expert Framework to the on-line learning community with their presentation of the WEIGHTED MAJORITY algorithm for finding the best predictor from a pool of possible candidates. Cesa-Bianchi, et. al. [11] gave an in-depth treatment of the case when the expert's loss is measured using *absolute loss*. The most powerful aspect of the Expert Framework is that it makes no statistical assumptions about the data. Instead, the loss of the master algorithm is usually bounded by a function of the loss of the experts in its pool.

The basic setup is as follows. First, a comparison class of predictors is chosen. These predictors are called *experts*, $\mathbf{e} = \{e_1, \ldots, e_N\}$. Usually experts are themselves learning algorithms, but they don't have to be. The framework is very flexible in this respect. Learning in the Expert Framework proceeds in trials $t = 1, \ldots, T$. At each trial the experts receive an instance/label pair $(x_t, y_t) \in X \times Y$, make a prediction $e_n(x_t)$, and incur a *loss* $L_{t,n} = L_t(e_n(x_t), y_t) \in [0, 1]$. The loss function measures the discrepancy between the expert's prediction and the true label. A *master* algorithm combines the predictions of experts to make its own prediction. The master maintains a weight vector $\mathbf{w} = \{w_1, \ldots, w_N\} \in \mathcal{P}^N$, where $\mathcal{P}^N$ is the probability simplex in $N$ dimensions. The weight $w_n$ represents the master algorithm's belief in the $n$th expert. After each trial, the master algorithm updates its belief in each expert $e_n$ by modifying $w_n$ in response to $L_{t,n}$. The most straightforward way of doing this is by applying the following *multiplicative update*:

$$w_{t+1,n} = \frac{w_t * \beta^{L_{t,n}}}{\text{normaliz.}}, \qquad \text{where } \beta \in (0, 1). \tag{2.1}$$

In the more recent literature, (2.1) is written with $\beta = e^{-\eta}$, where $\eta$ is referred to as the learning rate. A master algorithm using this update is sometimes referred to as the STATIC EXPERT algorithm [16]. Usually, $\mathbf{e}(x_t) \in \mathbf{R}^N$ and the master predicts by taking an inner product $\hat{y}_t = \mathbf{w} \cdot \mathbf{e}(x_t)$, and itself incurs loss $L_{t,A} = L_t(\hat{y}_t, y_t) \in [0, 1]$. Vovk [19, 24] discusses fancier prediction functions that lead to better loss bounds.

Algorithms employing the update given in (2.1) have bounds which relate the loss of the master algorithm to the loss of the best expert. Let $L_{1\ldots T,A}$ and $L_{1\ldots T,n}$, for $n = 1, \ldots, N$, be the cumulative loss of the master algorithm, and the experts $e_n$ respectively. Then, for any sequence $S$ of instance/label pairs $(x_1, y_1), \ldots (x_t, y_T)$,

$$L_{1\ldots T,A} \leq \min_n \{L_{1\ldots T,n} + c_L \log n\}, \tag{2.2}$$

where $c_L$ is a constant that depends on the loss function [14, 19]. In other words, the loss of the master algorithm is bounded by the loss of the best expert, plus the term term $c_L \log n$, which can be recognized of as the *minimum description length* required to encode the best expert. Vovk [24] generalized the theory to incorporate other loss functions.

Bounds like (2.2) are great when the best expert is the most sensible comparator. But what happens when the best expert changes over time? Imagine an off-line algorithm that can *partition* the sequence into sections, and choose the best expert in each section. “Tracking” the best expert over time requires a weight update which could help a previously poor expert, with low weight, to recover weight quickly when it starts predicting well. Multiplicative updates like those in (2.1) are a double-edged sword: driving the weights of experts which are currently poor to zero so quickly that it becomes very difficult for them to recover if they start doing well.

Herbster and Warmuth [16], and more recently Bousquet and Warmuth [6], developed a second set of updates in order to help prevent poor experts’ weights from becoming too small. After an intermediate LOSS UPDATE, identical to that of (2.1), these updates, dubbed SHARE (or MIXING) UPDATES, force experts with a large amount of weight to share (or mix) a small amount of their weight with poor experts.

$$w^m_{t,n} = \frac{w_{t,n} e^{-\eta L_{t,n}}}{\text{norm.}}, \qquad \mathbf{w}_{t+1} = \sum_{q=0}^{t} \gamma_{t+1,q} \mathbf{w}^m_q, \qquad \text{where } \sum_{q=0}^{t} \gamma_{t+1,q} = 1.$$

Mixing portfolios $\gamma_{t+1}$ prescribe how current and past weights ($\{\mathbf{w}^m_1, \ldots, \mathbf{w}^m_t\}$) contribute to the weight vector used in the next trial ($\mathbf{w}_{t+1}$). Typically, $\gamma_{t+1}$ constructs $\mathbf{w}_{t+1}$ by mixing a fixed share ($\alpha \ll 1$) of the first $t-1$ intermediate weight vectors with larger share ($1-\alpha$) of $\mathbf{w}^m_t$, from the most recent LOSS UPDATE.

Some example mixing schemes, discussed extensively by Bousquet and Warmuth [6], are outlined below. Figure 2.1 shows pictorially how $\gamma$ mixes past intermediate weights in these updates.[2]

1. FIXED SHARE TO START VECTOR mixes in a small amount of the (uniform) initial weight vector with $\mathbf{w}^m_t$ from the most recent LOSS UPDATE, to help

[2]picture borrowed from Bousquet & Warmuth [6].

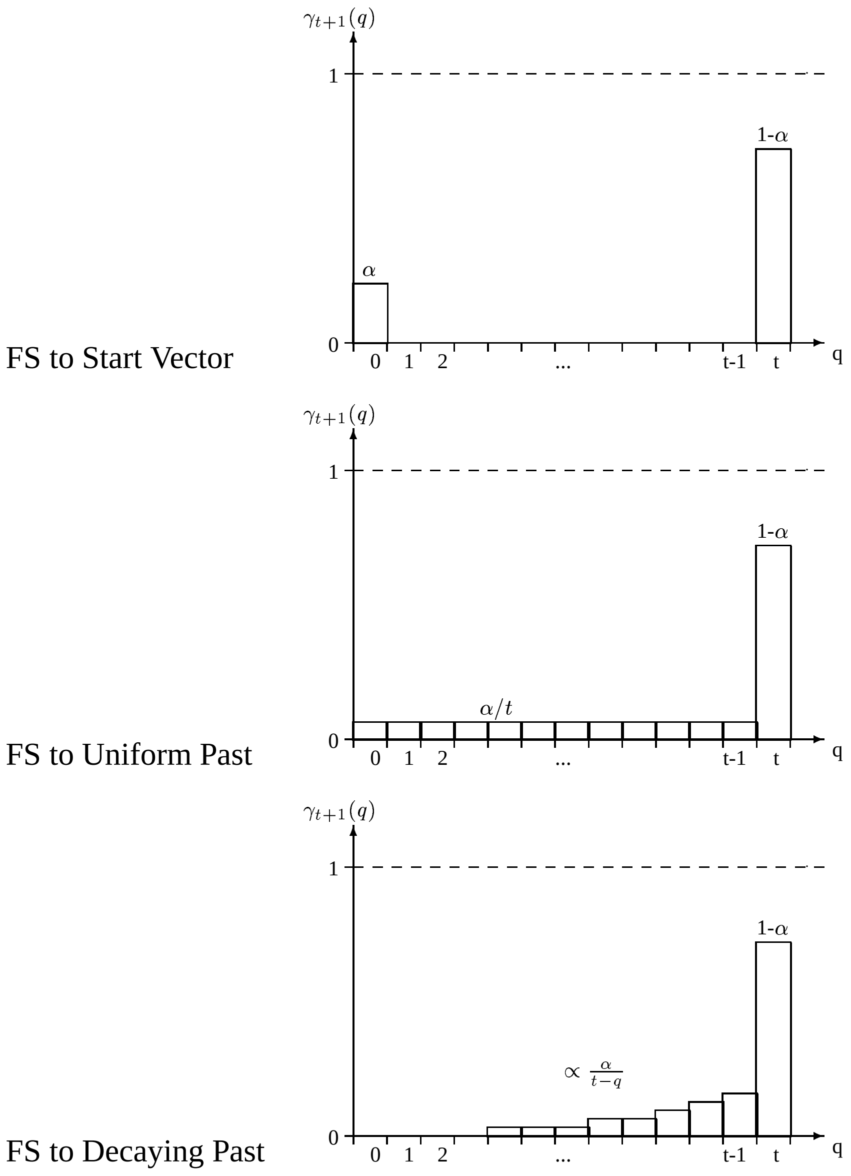

Figure 2.1: Three Fixed Share (FS) share updates: FS to Start Vector, FS to Uniform Past, and FS to Decaying Past. The diagrams describe how intermediate weights $w^m_{t,n}$ are mixed in order to determine $w_{t+1,n}$.

prevent experts' weights from becoming too small.

2. Fixed Share to Uniform Past mixes in the past average of intermediate weight vectors. This share update is ideal for the situation where there is only a

small sub-pool of helpful experts. Currently poor experts which were good in the past maintain a slightly higher "dormant" weight than those which have proven less useful over time.

3. FIXED SHARE TO DECAYING PAST also mixes in the weight vectors of previous trials, but is more generous about sharing weight to experts who have faired well recently.

The loss of master algorithms which employ mixing updates (like those described above) can be bounded by a function of the loss of the optimal partition of experts. Just as before, these bounds usually include terms representing the minimum description length of an encoding of the partition (and experts used):

$$L_{1\ldots T,A} \leq \min_P \{L_{1\ldots T,P}\} + O(\text{\# of bits to encode } P) \tag{2.3}$$

where $L_{1\ldots T,P}$ is the loss of partition $P$.

Some recent examples of applications of the Expert Framework to operating systems problems follow.

### 2.1.1 Experts to Spin Down Hard Disks

Mobile computers can conserve battery power by spinning down the hard disk when it is not in use. On many systems, inactivity for a fixed amount of time determines whether or not the disk should spin down. Fixed timeouts on the order of 30 seconds to 10 minutes are typically chosen by the user of the device. Helmbold, et. al. [15] propose a more adaptive approach which uses many, harmonically or exponential spaced, fixed timeouts as experts $e_n$, for $n = 1, \ldots, N$. The algorithm proceeds in trials $t = 1, \ldots, T$. At each trial, $t$, an idle time $i_t$ is provided to the master. For each idle period, $i_t$, the energy used by the fixed timeout $e_n$ is computed as

$$\text{Energy}(e_n, t) = \begin{cases} i_t & \text{if } i_t \leq e_n \\ e_n + \text{spin-down cost} & \text{otherwise.} \end{cases}$$

Supposing the algorithm knew the length of the idle time at the beginning of a burst of inactivity, the energy used by an optimal strategy would be

$$\text{Optimal}(t) = \begin{cases} i_t & \text{if } i_t \leq \text{spindown-cost} \\ \text{spin-down cost} & \text{otherwise.} \end{cases}$$

To keep the units simple, spindown-cost is measured in seconds of (spun-up) disk idling. Using the above, the loss of each expert is then computed as

$$\text{Loss}(e_n, t) = \frac{\text{Energy}(e_n, t) - \text{optimal}}{\text{spin-down cost}}.$$

A variant of Herbster and Warmuth's VARIABLE SHARE algorithm [16] is used to update the experts' weights, and the actual timeout used by the master algorithm is the usual weighted linear combination (inner product) of the experts' timeouts: $\sum_{n=1}^{N} w_n e_n$. It is reported that this approach results in the mobile computer using as little as 88% of the energy consumed by the best possible (fixed) time-out chosen in hindsight.

This application of the Expert Framework is perhaps the most promising and successful one to date. It is also the first application (known to us) of the Expert Framework to an operating systems problem.

### 2.1.2 Experts for Paging

Blum, et. al. [4] applied updates from the Expert Framework to *paging*. Their policy is designed to exploit a situation quite similar to our own. Underlying everything is the assumption that the workload is partitioned into *phases*. Each phase consists of a "working set" of pages which receive a majority of the requests. With this setup they consider an off-line comparator called $r$-unfair. The comparator $r$-unfair knows the future, and therefore knows the phase boundaries. At the beginning of each phase, $r$-unfair fetches only the most frequently requested pages (the working set) into the cache. During a phase, requests to pages not in the current working set are "rented" (at a loss of $1/r$) rather than fetched into the cache (which incurs a loss of 1).

An on-line policy proceeds similarly, but cannot anticipate phase changes. At the start of each phase it creates $k!$ experts, one for each permutation of the pages in the working set *learned* in the previous phase.[3] Each expert then keeps its own view of the cache, and executes a MARKING algorithm [5]. The expert's initial permutation determines the order in which unmarked pages are discarded. Requests to new pages are rented (this time at a loss of 1) until they become marked (requested $r$ times in the phase). Only marked pages are actually fetched into the cache. When all $k$ pages in the cache become marked a new phase begins. Within each phase the RANDOMIZED WEIGHTED MAJORITY algorithm [20] is used to update a distribution of weights, one weight for each expert. The weights are used to construct a probability distribution $P_j$, over all pages $j$. $P_j$ is computed by taking the ratio of the sum of weights corresponding to the experts which retained $j$, divided by the total weight of all experts.

$$P_j = \frac{\sum_{n:j \in e_n} w_n}{\sum_n w_n} \tag{2.4}$$

$P$ determines the order in which unmarked pages are released from the cache in order to make room for marked pages that need to be fetched. By proceeding in this way the expected number of page faults will be the same as the expected loss to the RANDOMIZED WEIGHTED MAJORITY algorithm.

[3]At the beginning of the first phase the previous working set is empty.

Blum, et. al. show that this on-line policy has a competitive ratio[4] of $O(r + \log n)$ with respect to $r$-unfair. While an impressive result, Blum, et. al, fail to address the computational burden and memory required to maintain $k!$ experts, and each of the corresponding $k!$ views of the $k$ pages in the cache. Their paper is largely a theoretical work, and provides no experiments or simulations demonstrating the advantage of their approach in practice.

### 2.1.3 Experts for Caching

Recently, the Machine Learning and Storage Systems groups at UC Santa Cruz embarked on a joint venture to apply on-line learning to storage related problems, including caching. In a preliminary work, Ari, et. al. [1], discuss a framework for Adaptive Caching using Multiple Experts (or ACME) in order to manage replacements within distributed caches[5]. The topology of a distributed cache is both geographic and hierarchical. Employing the same caching strategy at adjacent nodes, or between parent and child nodes, is usually sub-optimal [25, 7]. For example, a cache which only serves requests missed by a parent cache executing LRU will likely observe a request stream which is recency-saturated. Implementing LRU at the child level is a poor choice. Even with complete knowledge of the request stream, choosing the best policy for each node can be a daunting task for an administrator. ACME was designed to help nodes choose their own policy based on the requests they observe.

Extending this research, Gramacy, et.al. [12], demonstrated how one could shift back and forth between policies in order to exploit temporal changes in the workload's characteristics. Turning ACME on its head by taking the experts to be the policies' caches, instead of the policies themselves, we were able to use the Expert Framework to produce a master policy whose miss rate was *lower* (experimentally) than that of

[4]*competitive ratio:* worse case loss of the on-line policy over the optimal comparator.

[5]*distributed caches*: a network of autonomous caches (nodes) whose workload depends on their geographic location, and the policies executed by nearby nodes (e.g. web proxies, $n$-level caches).

the best policy in the pool. Our key observation was that experts acquire high weight because they retained objects currently favored by the workload. This thesis expands on and solidifies the ideas expounded in that work.

# 3. Off-line Comparators

Our goal is to design a caching policy which "adapts" its replacement criteria based on which policies in its pool are currently best. Of course, implicit in the desire for something adaptive is a tacit desire for something good. Many caching policies claim to be "adaptive", but seldom is "adaptive" clearly defined. Thus, we use the term "adaptive" only informally. When we want to be precise, we use off-line comparators to judge the performance of our on-line algorithms, as is commonly done in the on-line learning community [20, 10, 18]. In this paper, we use three off-line comparators: BestFixed, BestShifting$(K)$, and BestRefetching$(R)$. These are described below.

In addition to these off-line comparators we also compare to LRU (our easiest comparator). A successfully adaptive policy should do at least as well as LRU. We also compare against the rate of *compulsory* misses (our hardest comparator). The *compulsory* "policy" only misses requests for objects it has never seen before. Its miss rate of is the same as that of any policy which governs an infinitely large cache, and is a lower bound on the miss rate of the optimal replacement policy on a cache of any size.

## 3.1 BestFixed and BestShifting$(K)$

BestFixed is the *a posteriori* selected policy with the lowest miss rate on the entire request stream, chosen from the pool of twelve (or $N$) policies. Computing this comparator is straightforward.

BestShifting$(K)$ considers all possible partitions of the request stream into at most $K$ segments along with the best policy for each segment. BestShifting$(K)$ then chooses the partition yielding the lowest miss rate over the entire dataset.

BestShifting$(K)$ can be computed using dynamic programming. Let $B(t, k, i)$ be the total number of misses accrued by BestShifting$(k)$, for $k \geq 0$, after observing $t \geq 0$

requests, where the current policy (on the segment including the $t$th request) is policy $i \in \{1, \ldots, n\}$. $B(t, k, i)$ can be computed with the following recurrence:

$$B(t,k,i) = \begin{cases} 0 & t = 0, k \geq 1 \\ B(t-1,k,i) + \text{miss}_{t,i} & t > 0, k = 1 \\ \min\{B(t-1,k,i) + \text{miss}_{t,i}, \quad \text{(or)} & t > 0, k > 1 \\ \qquad \min_{j \neq i}\{B(t-1,k-1,j) + \text{miss}_{t,i}\}\} & \end{cases} \tag{3.1}$$

where $\text{miss}_{t,i} \in \{0, 1\}$ indicates whether the $t$th object would have been retained ($\text{miss}_{t,i} = 0$) or discarded ($\text{miss}_{t,i} = 1$) by the $i$th policy.

Using dynamic programming *memoization* to fill in a table $B[T, K, i]$ takes time in $O(TKN)$, where $T$ is the total number of requests in the workload, and $K$ is the largest $k$ of interest. Having filled in the table $B[\cdot, \cdot, \cdot]$ we can extract

$$\text{BestShifting}(K) = \min_i B[T, K, i]. \tag{3.2}$$

Moreover, it is also the case that

$$\text{BestFixed} = \text{BestShifting}(1). \tag{3.3}$$

Another interesting comparator arises when we consider BestShifting$(K)$, as $K \to \infty$. We call this comparator *AllVC*:

$$\text{AllVC} \doteq \lim_{k \to \infty} \text{BestShifting}(k), \tag{3.4}$$

This is an interesting benchmark whereby a miss is incurred only if the object would have been missed by all of the $N$ baseline policies.

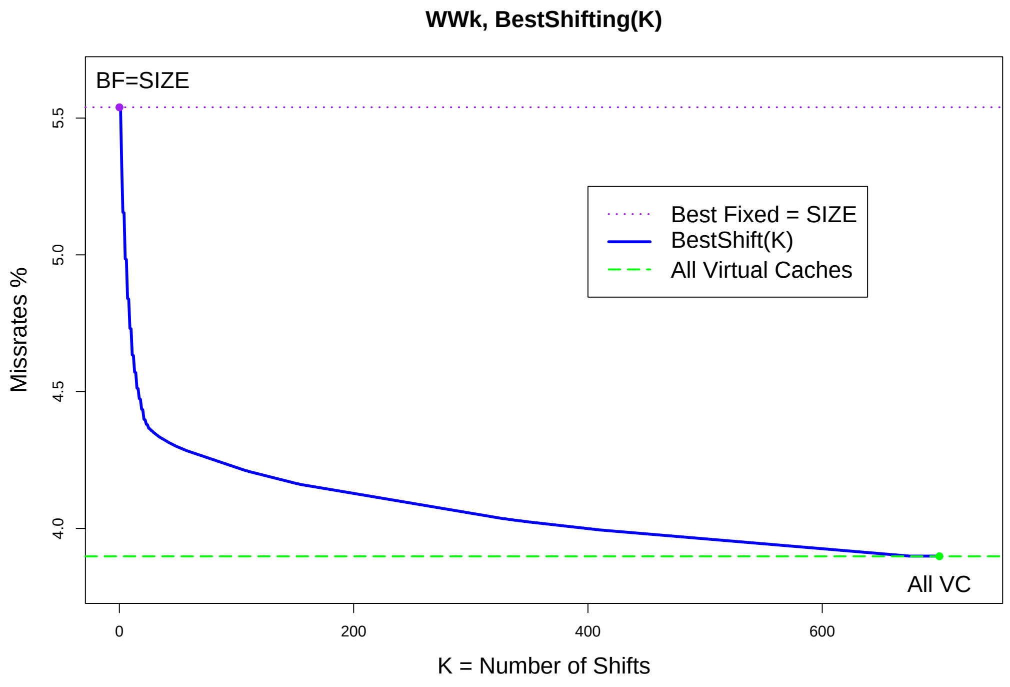

Figure 3.1: Off-line comparators for a typical dataset (WWk, 138,000 requests, see Section 6.1). Miss rates are plotted as a function $K$, the number of policy switches or segments. BestFixed incurs 7,665 misses, AllVC incurs 5,396. 2% (or 2,970 misses) are compulsory.

Figure 3.1 shows BestShifting$(K)$, BestFixed, and AllVC graphically for a typical dataset of about $138,000$ requests. Here, miss rates are plotted as a function $K$, the number of policy switches or segments. The BestFixed policy is SIZE. Notice that a small number of shifts ($K \approx 50$) gives a *shifting policy* with a miss rate $\approx 20\%$ less than BestFixed. More shifts ($K > 50$) gives only marginal improvement.

We seek to develop a policy which switches its decision criteria to effectively exploit significant changes in the workload's characteristics. Informally, we consider a policy to be "adaptive" when its miss rate is lower than BestFixed and close to BestShifting$(K)$, for modest $K$.

### 3.1.1 How BestShifting$(K)$ Cheats

Even though it has knowledge of the future, BestShifting$(K)$ is still restricted to follow the same protocol as the type of online policy we seek to develop; namely that it can only solicit advice from a pre-specified pool of policies in order to facilitate requests. Still, there are two ways in which BestShifting$(K)$, as specified above, "cheats".

The first cheat is that BestShifting$(K)$ uses its knowledge of the future to take advantage of baseline policies which experience and intuition suggests are poor choices in all but a few rare scenarios. Policies like MRU, MFU, and LIFO, prefer releasing objects that where recently accessed. We included them in our pool of policies on the off-chance that they might prove unexpectedly useful. It turns out that BestShifting$(K)$ does indeed find them useful, but not for the purpose we intended. For larger $K$, after many of the major shifts in the characteristics of the workload have been assigned (more sensible) policies, we found more and more short segments governed by one of MRU, MFU, or LIFO.

Fortunately, employing MRU, MFU, and LIFO does not help BestShifting$(K)$ all that much. Figure 3.2 shows that removing these policies leaves the miss rate largely unaffected. When $K$ is small they are not being used at all.

The second cheat involves the state of the cache when BestShifting$(K)$ switches policies. Its recurrence, as given above, makes the powerful assumption that the operation of switching policies comes at a negligible cost. In (3.4), miss$_{t,i}$ indicates whether policy $i$ would have missed object $t$ if it had governed the cache all along. That is, the recurrence prescribes not only a change of governing policy, but also a swap of cache contents. This situation is an artifact of our attempt to keep BestShifting$(K)$ simple, and tractable. Realistically, a new governing policy should work with a cache state which resulted from management by previous governing policies.

Our experiments suggest that much of the advantage gained by changing governing policies comes not because a new policy is expected to make wise replacements in the future, but rather that its past choices make it preferred for the current requests.

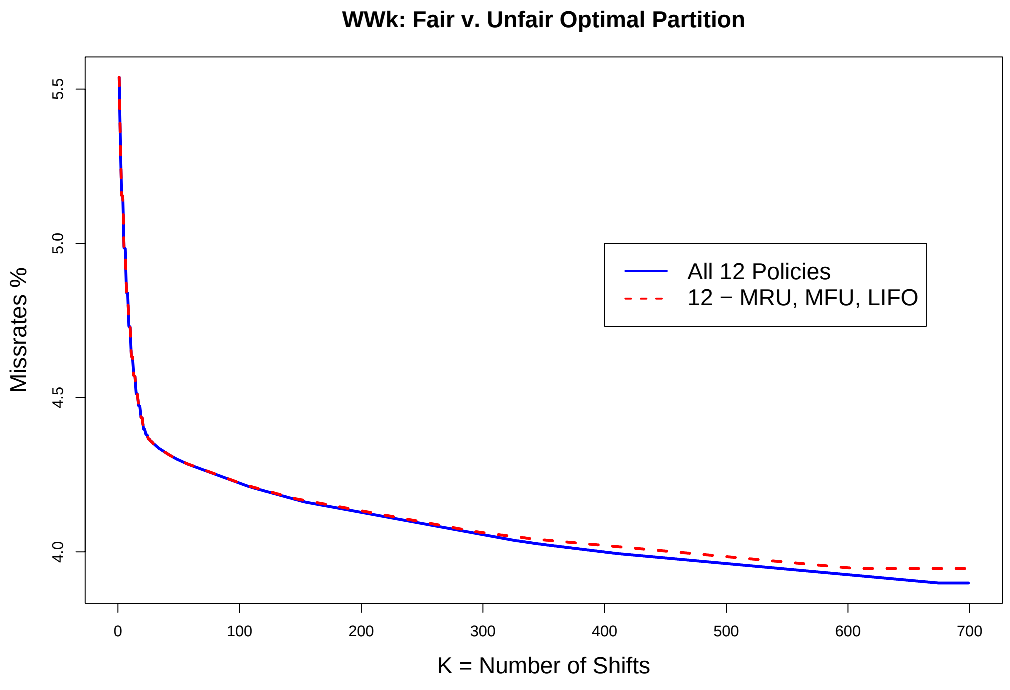

Figure 3.2: Comparing BestShifting$(K)$'s for different sub-pools of policies to judge "fairness". Notice that including policies like MRU and MFU which are notoriously bad strategies online actually help BestShifting$(K)$, but only a little bit.

Therefore, even though BestShifting$(K)$ "cheats" by *rolling over* its cache contents at each policy shift, it models the extreme situation where the best objects to keep in (or bring into) the cache are those cached by the best policy. This idea is what we will try to exploit when developing an on-line master policy. BestShifting$(K)$ is an ideal comparator for judging how successful we are at accomplishing this goal.

### 3.1.2 BestRefetching$(R)$

One way to better understand the amount by which BestShifting$(K)$ uses rollover to cheat is to keep track of the number of *refetches* required to facilitate each cache shift. Realistically, objects discarded by a previous governing policy, which are cached by the new governing policy, would have to be refetched. Rather than plotting $K$ on the x-axis, as in Figure 3.1, BestShifting$(K)$ could be plotted versus an $R$-axis which shows the number of refetches required to accomplish $K$ shifts. Adding an extra $R$-table in the dynamic programming (3.1), and extra inputs depicting the refetch-distance between all caches at all times, suffices to record the number of refetches used by $B[\cdot,\cdot,\cdot]$.

Alternatively, one can derive a recurrence (and thus another comparator) which deals with refetches more naturally– not just as an afterthought. With this in mind, we define BestRefetching$(R)$ to be the minimum number of misses incurred a partition of policies into segments which results in $R$ refetches, regardless of the total number segments used.

Computing BestRefetching$(R)$ is is similar to computing BestShifting$(K)$. Let ref$_t(i,j)$ be the number of refetches required to move from the cache of policy $i$ to policy $j$, at time $t$. Now let $B'(t,r,i)$ be the minimum number of misses incurred by a shifting policy for the first $t$ requests, ending with the governing policy $i$, and using at most $r$ refetches to accomplish an arbitrary amount of shifting between governing policies. Then, in the spirit of (3.1) ...

$$B'(t,r,i) = \begin{cases} 0 & t=0, r\geq 0, \\ \infty & r<0, \\ B'(t-1,r,i)+\text{miss}_{t,i} & t>0, r=0, \\ \min\{B'(t-1,r,i)+\text{miss}_{t,i}, \quad \text{(or)} & t>0, r\geq 1. \\ \quad \min_{j\neq i}\{B'(t-1,r-\text{ref}_t(j,i),j)+\text{miss}_{t,i}\}\} & \end{cases} \tag{3.5}$$

As before, (3.5) can be computed by using dynamic programming, and we have that

$$\text{BestRefetching}(R) = \min_i B'[T,R,i]. \tag{3.6}$$

Again, we have

$$\text{BestFixed} = \text{BestRefetching}(0) \quad \text{and} \quad \text{AllVC} \doteq \lim_{r\to\infty} \text{BestRefetching}(r). \tag{3.7}$$

The memoization table $B'[\cdot,\cdot,\cdot]$ takes time in $O(N^2RT)$ to fill in. The reason for the $N^2$ term instead of $N$, like in the $O(NKT)$ time for filling in $B[\cdot,k,\cdot]$ to compute BestShifting$(K)$ via (3.1), is as follows. The term $r - \text{ref}_t(j,i)$ in (3.5) might cause $\min_{j\neq i}$ to be different for each choice of $i$. Therefore, each $B'[t,r,i]$ requires $O(N^2)$ to compute because the $\min_{j\neq i}$ needs to be re-evaluated for all $i = 1,\ldots,N$. In other words, the choice of which policy to let govern the current request depends on how previous requests are partitioned into segments and governing policies, given a particular refetch allowance. Instead of "$r - \text{ref}_t(j,i)$", which depends on both $i$ and $j$, the recurrence for computing BestShifting$(K)$ given in (3.1) has simply "$k-1$", which doesn't depend on either $i$ or $j$. As a result, $\min_{j\neq i}$ is the same for all $B'[\cdot,\cdot,i]$, and thus needs only be evaluated $N$ times to compute $B[t,k,\cdot]$.

Asymptotic behavior aside, we typically see that the number of refetches, $R$, induced by a particular $K$ is orders of magnitude larger than $K$. Each shift in policy can cause anywhere between 10 and 200 refetches. So computing BestRefetching$(R)$ requires vastly more resources than computing BestShifting$(K)$, in terms of both time and space. In our experiments, computing BestRefetching$(R)$ consumes about 100 times the memory of BestShifting$(K)$, and takes 1200 times longer to compute. Figure 3.3 compares BestShifting$(K)$ and BestRefetching$(R)$, by plotting their miss rates as a function of refetches. Refetches are plotted on the x-axis as a percentage of the total number of requests. At the two extremes (no refetches or shifts, and saturating refetches or shifts) both methods give the same results:

$$\text{BestShifting}(1) = \text{BestRefetching(0)} = \text{BestFixed}$$

and

$$\text{AllVC} = \lim_{k\to\infty} \text{BestShifting}(k) = \lim_{r\to\infty} \text{BestRefetching}(r).$$

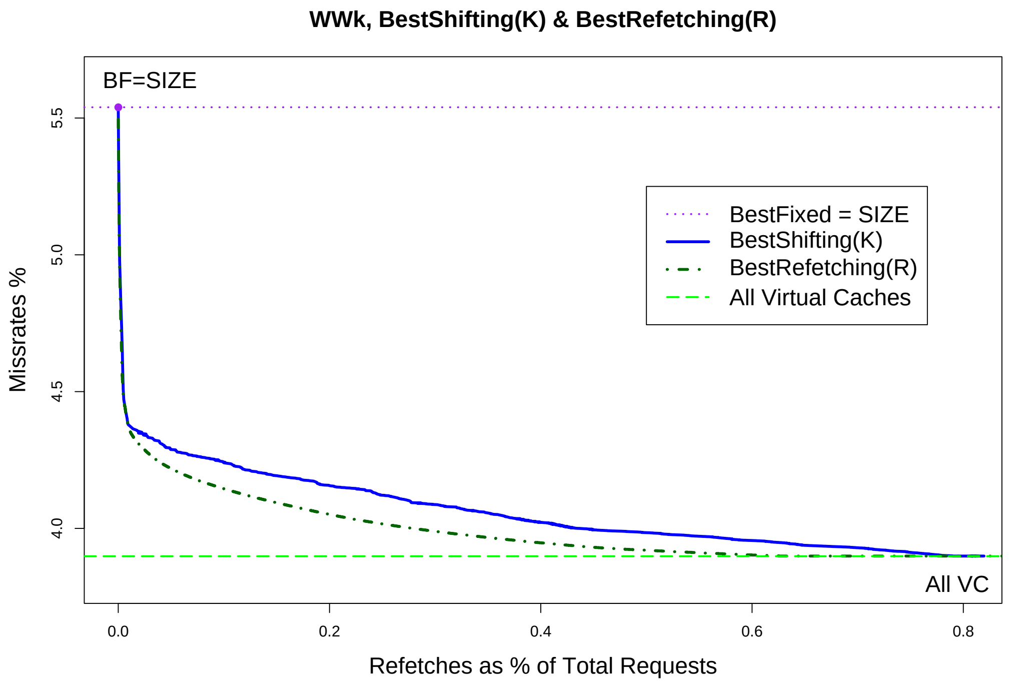

Figure 3.3: Shows BestShifting$(K)$ with miss rates plotted as a function of refetches (x-axis), computed to facilitate each of the $K$ shifts, and BestRefetching$(R)$ for comparison.

However, BestShifting$(R)$ is not restricted by the number of policy shifts it uses. This allows it to achieve lower miss rates with less refetching.

## 3.2 Regions of Goodness

Later, when we experiment with our on-line master policies, we will want to get a sense of how good they are. Here, we highlight several (compound) ways in which an on-line adaptive algorithm can be good, as measured against the comparators outlined previously.

Caching policies are usually developed with the following goal: to reduce user latencies resulting from memory requests. Often, this is accomplished by reducing the number of fetches which result from misses. However, our comparators, and our online algorithms, can further reduce the miss rate by actually increasing the number of fetches– by cleverly refetching objects before they are requested. Depending on the system, there may be a tradeoff between increasing the number of fetches (I/O's) and reducing the end-user latency of requests (miss rate).

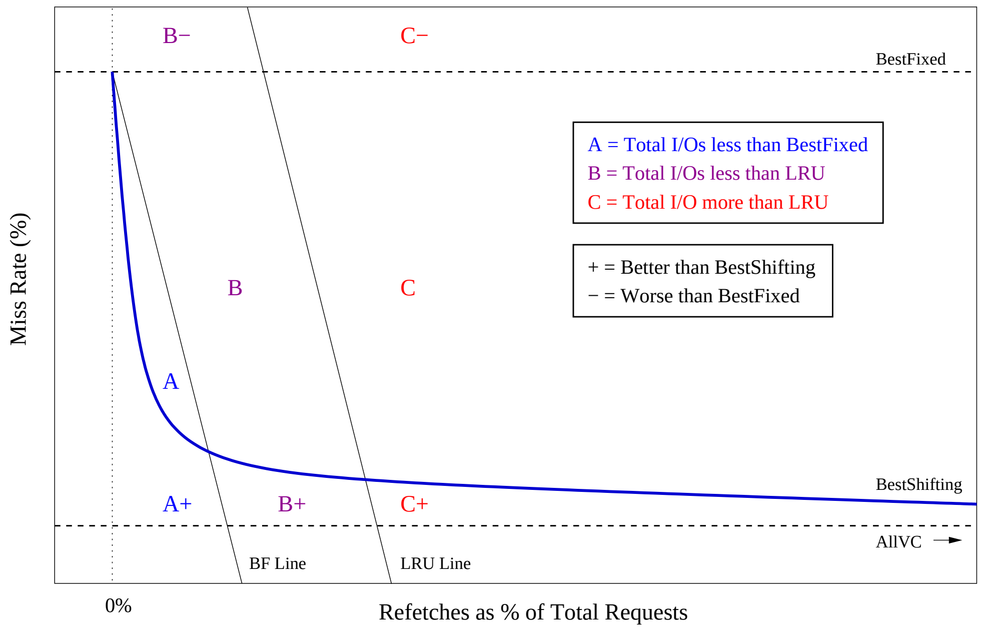
B−
C−
BestFixed
A = Total I/Os less than BestFixed
B = Total I/Os less than LRU
C = Total I/O more than LRU
+ = Better than BestShifting
− = Worse than BestFixed
Miss Rate (%)
B
C
A
BestShifting
A+
B+
C+
AllVC
BF Line
LRU Line
0%
Refetches as % of Total Requests

Figure 3.4: Eight regions of savings (or success) with respect to BestFixed, BestShifting, and LRU. The regions are horizontally partitioned by the BestFixed miss rate, and the AllVC miss rate. BestRefetching$(R)$, BF and LRU Lines partition the space vertically. BF and LRU Lines are constructed by drawing lines connecting the BestFixed and LRU miss rates, $\ell$ and $b$ respectively, with their counterpart on each axis (BF Line=$\{(0,b),(b,0)\}$ and LRU Line=$\{(0,\ell),(\ell,0)\}$).

Figure 3.4 shows eight regions in a plot of miss rate versus refetches where success (or failure) can be interpreted eight different ways. The regions are divided by BestRefetching, BestFixed, and two lines which represent an equal tradeoff between misses and refetches of BestFixed and LRU. I/O's result from both misses and refetches. Policies whose miss rate and refetches place it below and to the left of the *LRU Line*, in the figure, cause less total I/O’s (misses + refetches) than LRU. Likewise, policies mapped below and to the left of *BF Line* cause less total I/O’s than BestFixed.

Compared to LRU, regions A* and B* in the figure correspond to a decrease in *both* the number of refetched objects, and the number of misses. C* regions correspond to fewer misses than LRU, but ultimately result in an increase in total number of I/O’s due to refetching. Compared to BestFixed, only A* regions correspond to both a decrease in miss rate, and a decrease in the total number of fetches.

Regions labeled “+” correspond to miss rates better than BestShifting, whereas regions labeled “-” correspond to miss rates worse than BestFixed. If the amount of I/O’s is a non-issue, then to be the C+ region is our goal. If, like in most cases, we prefer to minimize both the number of total fetches and the number of misses, then our goal is to be in one of {A,A+,B,B+}.[1]

[1] B regions can disappear if BestFixed=LRU, but this has never happened with our data. With the set of 12 policies we used, LRU is never the best policy.

## 3.3 Virtual Caches and Master Policies

We seek to develop an on-line master policy that determines how a set of baseline policies should govern the real cache at any given time. First, the policies need to be evaluated. Our key idea is to use "virtual caches". A *virtual cache* simulates the operation of a single baseline policy. Each virtual cache records a few bytes of metadata about each object in its cache: ID, size, and calculated priority. Object data is kept only in the real cache, making the cost of maintaining the virtual caches negligible[2].

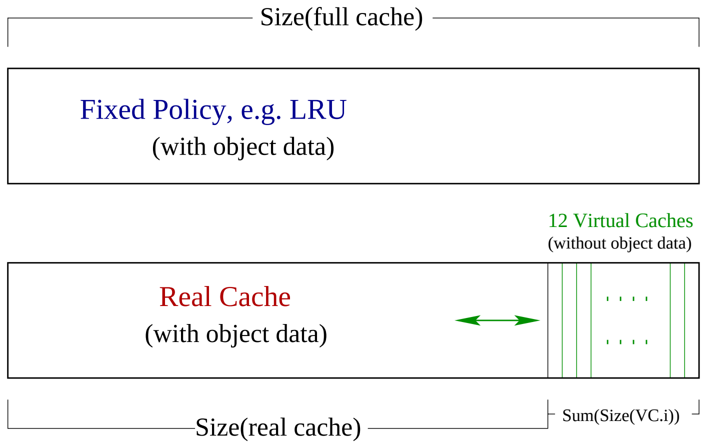

Figure 3.5: Virtual caches consume real cache memory.

To be fair we also require that virtual caches reside in the same cache memory that would otherwise have been used to cache "real" object data. Figure 3.5 shows the virtual caches nestled inside of the real cache space. Virtual caches operate with fixed (virtual) cache space equal to the size of the full cache. Therefore, the actual space used to cache real objects is a function of the size of the 12 virtual caches, and changes with time:

$$\text{Size(real cache)} = \text{Size(full cache)} - \sum_{i=1}^{12} \text{Size}(\text{VC}_i).$$

[2]As an additional optimization, we record the id and size of each object only once, regardless of the number of virtual caches it appears in.

Dictionary data structures can be used to manage object headers cached in any of the $N$ virtual caches (and the real cache). Virtual caches themselves only contain pointers to dictionary entries. This makes their space usage (in the total cache) even more compact, and lookups of object headers in all caches more efficient. In our experiments the total size of all twelve virtual caches consumed less than 1% of the full cache memory.

Virtual caches are proving grounds for a pool of candidate policies. Our task is to produce a *master policy* which observes the success of each policy on its virtual cache, and bases its own replacement policy on those it judges to be currently "best". This leaves the question of how to make such a judgment. One possibility is the Rolling Window Algorithm discussed in the following subsection. A cleaner, more efficient approach uses the Expert Framework from on-line learning. This will be the focus of much of the rest of this paper.

### 3.3.1 One possible approach: Rolling Window Algorithm

Via virtual caches, the master policy can observe the miss rates of each policy on the actual request stream in order to determine their performance on the current workload. A simple heuristic which can be used to determine how to manage the real cache at any given time, is to continuously monitor the number of misses incurred by each baseline policy in a past window of $W$ (say 300) requests (depicted in Figure 1.1(a)). The master policy can give control over the real cache to the policies with the least misses in this window. We call this the *rolling window* algorithm.

This approach has two disadvantages. Firstly, $W$ is difficult to tune. In our experiments (not shown) the optimal window sizes range is $W \in [100, 8000]$, depending on the workload. Secondly, $O(NW)$ memory (in addition to that required by virtual caches) is needed for $N$ policies each with a window size of $W$. Using windows, the master policy's bookkeeping would encroach even further into the real cache space. Both of these problems are mitigated by using the Expert Framework for on-line learning in place of a rolling window. This is the topic of the next section.

# 4. Using the Expert Framework

Choosing caching strategies is a useful application of the *Expert Framework* for online classification. While not a classification problem, policies or their virtual caches can be thought of as as experts, misses as some unit of loss, and predictions as object replacements. Miss rates on past windows of a fixed size $W$ are similar to the idea of maintaining a distribution over the experts, but we have already argued that this can be expensive ($O(NW)$ for $N$ experts). Using updates developed for the Expert Framework [16, 6] gives a similar effect by maintaining only a single weight $w_n$, for each expert $n \in \{1, \ldots, N\}$. The weight $w_n$ is an estimate of the performance of policy $i$ relative to the other policies. We commonly refer to the distribution of weights as a vector $\mathbf{w} = \langle w_1, \ldots, w_N \rangle$.

## 4.1 Updating the Expert Weights

After each request the master policy updates $\mathbf{w}$ in two ways, thereby updating its belief in each policy. Updating the weight vector $(w_1, \ldots, w_N)$ after each trial commences in two stages. First, the weights of all policies that missed the new request are *multiplied* by a factor $\beta \in (0, 1)$ and then renormalized. This is called the *loss update* (see Section 2.1). Since the weights are renormalized, they remain unchanged if all policies miss the new request. As noticed by Herbster and Warmuth [16], multiplicative updates drive the weights of poor experts to zero so quickly that it becomes difficult for them to *recover* if their experts subsequently start doing well. This is sometimes referred to as the “curse of the multiplicative update”. Therefore, a second *share update* prevents the weights of experts that did well in the past from becoming too small, allowing them to recover quickly, as shown in Figure 4.1.

There are a number of share updates [16, 6] with various recovery properties. Some of these are outlined in Section 2.1. We chose the FIXED SHARE TO UNIFORM PAST

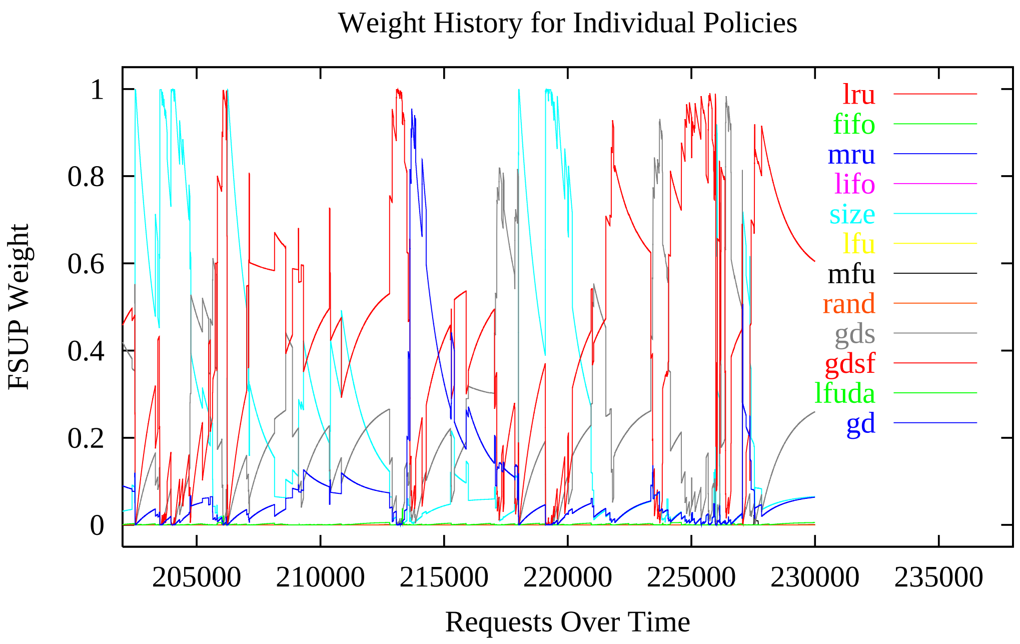

Figure 4.1: Weights of baseline policies over time under loss and share updates. (same snapshot of 30,000 requests from UMo, see Section 6.1).

(FSUP) update because of its simplicity and efficiency (more on this later), and because of its ability to fixate on a small sub-pool of the full pool of policies, encoding our prior belief that only a small subset of the policies will be useful. Note that the loss bounds proven in the Expert Framework for the combined loss and share update do not apply in this context. That is, we do not know how to relate the loss of the master policy to the weighted loss of its expert policies. Nevertheless, our experimental results suggest that we are indeed exploiting the recovery properties of the combined update that are discussed extensively by Bousquet and Warmuth [6].

Formally, for each trial $t$, the loss update is

$$w^m_{t,n} = \frac{w_{t,i}\beta^{\,\text{miss}_{t,n}}}{Z_{t+1}}, \qquad Z_{t+1} = \sum_{n=1}^{N} w_{t,n}\beta^{\,\text{miss}_{t,n}}, \qquad \text{for } n = 1, \ldots, N \tag{4.1}$$

where $\beta$ is a parameter in $(0, 1)$ and $\text{miss}_{t,i}$ is 1 if the $t$th object is missed by policy $i$

and 0 otherwise. The initial distribution is uniform, i.e. $w_{1,i} = 1/N$. The FSUP update mixes the current weight vector with the past average weight vector $\mathbf{r}_t = \sum_{q=1}^{t} \mathbf{w}_q^m / t$. This is easy to compute on-line without using more than a constant amount of memory per expert:

$$\mathbf{w}_{t+1} = (1-\alpha)\mathbf{w}_t^m + \alpha \mathbf{r}_{t-1} \qquad \mathbf{r}_t = \frac{(t-1)\mathbf{r}_{t-1} + \mathbf{w}_t^m}{t} \tag{4.2}$$

where $\alpha$ is a parameter in $(0,1)$. A small $\beta$ parameter causes high weight to decay quickly if its corresponding policy starts incurring more misses than other policies with high weights. The higher $\alpha$ is the more quickly previously good policies will recover. In our experiments we used $\beta = 1/e \approx 0.37$ and $\alpha = 5/1000 = 0.005$. Setting $\alpha = 0$ gives the STATIC EXPERT algorithm (mentioned in Section 2.1). In our experiments, using this algorithm resulted in all of the weight going to the BestFixed policy. Setting $\beta, \alpha$ will be discussed in more detail in Section 5.4.1.

### 4.1.1 Expert Framework v.s. Rolling Window

There are many reasons why we prefer using the Expert Framework to the rolling window algorithm for measuring the success of policies. As already mentioned, the $O(N)$ extra information required by the Expert Framework is much more reasonable than the $O(NW)$ required by the rolling window. Figure 4.1 illustrates a number of other advantages gained by using the Expert Framework instead of a rolling window. Compared to a rolling window, tracking weights over time gives a much clearer description of the policies competition. This is illustrated pictorially by comparing Figure 1.1 (a) and (c) with Figure 4.1.

The rolling window algorithm tracks an absolute performance metric in a fixed window. In contrast, the weights of the Expert Framework measure the success of policies relative to one another by taking recent requests into account and, to a smaller degree, all past requests. In both Figures 4.1 and 1.1 (c) it is apparent which policy is favored and when. Note that they agree. However, Figure 4.1 gives a visually clearer portrayal of the relative success of each of the policies over time, whereas Figure 1.1(a) is a complete mess in this respect.

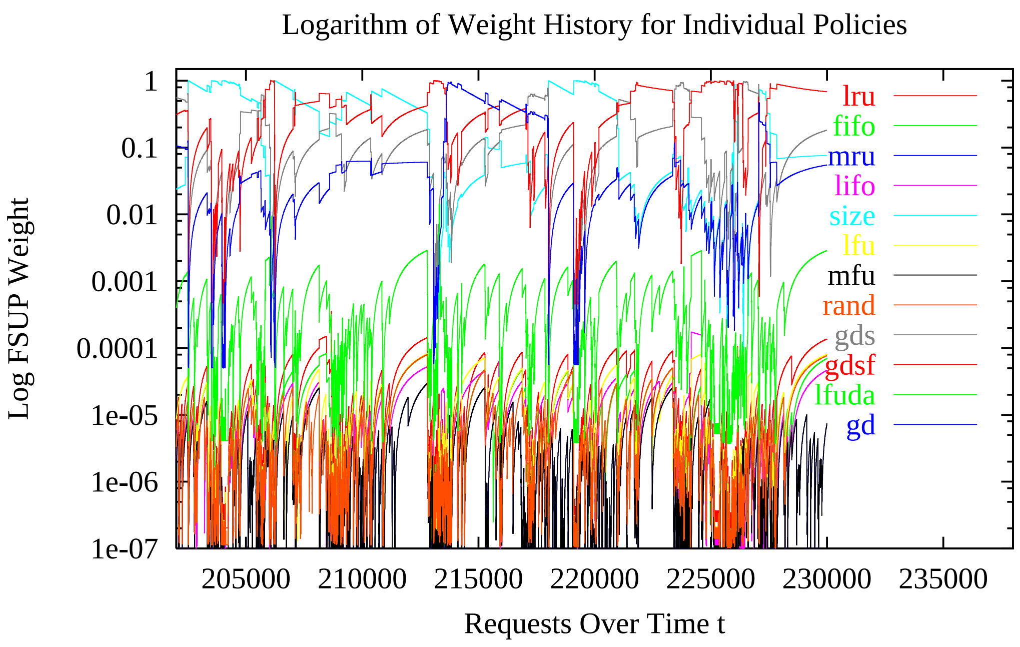

Figure 4.2: Logarithm of weights of baseline policies over time under loss and share updates. (same snapshot of 30,000 requests from UMo, see Section 6.1).

Figure 4.2 shows how the master policy uses FSUP to manipulate weights, favoring policies which have performed well in the past, by plotting the logarithm of the expert weights over time. Notice that the four best policies (SIZE, GDS, GDSF, GD*) maintain a log-weight several orders of magnitude higher than the other policies, even when they are doing poorly. Also, the figure brings out contenders like LFUDA, a policy designed to minimize *byte miss rate*[1], and to a lesser extent LRU, by showing that both maintain a noticeably higher dormant weight than the other policies.

[1] *byte miss rate* is an alternative measure of loss whereby policies are punished proportional to the size of the object missed.

We chose not to implement the Fixed Share to Decaying Past (FSDP) update because it requires extra memory on the order of the entire history (for $T$ requests) of weights for each of the $N$ policies ($O(NT)$). Bousquet and Warmuth [6] show that an approximation to FSDP can be achieved with only $O(N \log(T))$ more memory. Nevertheless, as $T$ gets large, the extra memory required would even further constrain the amount of space allotted for caching real objects, negating any initial benefit gained. In a real application $T \to \infty$. Any approach which requires memory or time resources that depends on $T$ is essentially useless.

# 5. Master Policy Managing the Real Cache

Expert framework weights represent the master policy's belief in each expert policy at any given time. Analogous with the rolling window, it makes sense to give control over the real cache to the policy with the highest weight. However, the success of a policy is essentially a function of both the policy *and* the (virtual) cache state that results from implementing that policy. If the master policy's belief in its expert policies changes over time, the objects in the real cache will likely differ from the objects cached virtually by the better rated expert policies. This suggests that a technique which switches both policy and cache contents will likely give the best results. With this in mind, we proceed with a high-level overview of our master policy, and the following subsections cover the details.

Rather than constructing a master policy by adapting and combining the best of the baseline policies' replacement criteria, we instead choose to develop a master policy which combines the best virtual caches. This approach is motivated by the observed success of BestShifting$(K)$ and BestRefetching$(R)$ and from our discussion in Section 3.1.1; namely the insight that a policy's current success is more a reflection of its past decisions than its present or future ones. Accordingly, we view the experts as rankings over the objects its policy chose to retain. The basic protocol of our master policy goes as follows:

1. A request is processed on each virtual cache and then the loss and share updates of the Expert Framework are applied.
2. Next, the request is processed on the real cache, based on the rankings of objects cached virtually by policies with high weight.
3. If desired, before processing the next request, objects can be refetched into the real cache, based on the rankings constructed in Step 2.

We will often need to reference all of the objects in all caches, real or virtual. So, from now on, we will refer to these objects as the master policy's *scope*. That is, the *scope* of the master policy contains those objects that are currently in the real cache, and all $N$ virtual caches.

In our previous work [12] we were concerned with finding an on-line shifting policy where the real cache was only governed by the single best policy at any given time. This approach is discussed in Appendix A. Notice that with our current setup, there need not be a single governing policy.

Viewing the experts' predictions as rankings makes them easy to combine. How rankings can be computed, combined, used to facilitate replacements, and suggest *refetches*, will be the topic of the following subsections. *Refetches* are like prefetches [23, 8, 13] except that the objects fetched have necessarily resided in the real cache at some time prior. More precisely, we define a *refetch* as a fetch of a previously seen object that was kept in the virtual cache(s) of higher weight expert policies, but was discarded from the real cache.

Based on these rankings, three *rollover* strategies will be discussed: demand, continuous, and background. *Rollover* is the process of bringing preferred objects into the real cache through a series of replacements and refetches. Demand rollover represents a baseline for feasible master policies which consist only of replacements, prescribing no additional refetches other than those required anyway on a per-request basis. Continuous rollover refetches aggressively and thus serves as a representative of what can be achieved by combining rankings on-line under the most liberal cost model. Background rollover is a cost-aware alternative to continuous rollover which combines replacements and a small amount of refetching as time and system resources permit. It turns out that a small number of refetches is just what's needed to reap the full benefit of combining policies.

## 5.1 Ranking

Each expert keeps a priority for the individual objects in its virtual cache. These priorities specify the order in which objects are discarded, inducing a ranking on its virtually cached objects. Lower ranked objects are discarded first, higher ranked objects last. The master policy can use these rankings together with the experts' weights to make a ranking of its own. This master ranking specifies which objects in the real cache should be discarded first, and which objects should be refetched, if possible. Figure 5.1 gives a rough sketch of how the expert's rankings are combined; the details follow.

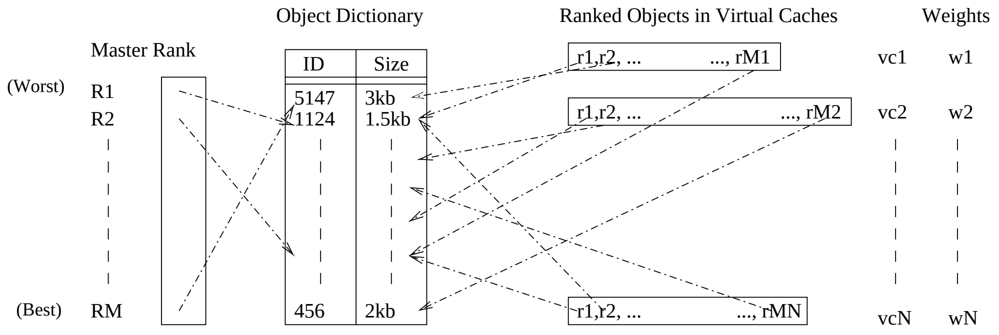

Figure 5.1: Weighted experts' ranks are combined to make a master ranking of all objects. Here, the Object Dictionary is a mechanism which allows quick access to the scope of all objects the master policy knows about. Object ranks flow from the weighted virtual caches pointing from the right, forming a master rank of all objects pointing from the left.

Let RealCache denote the set of objects in the real cache, and $\mathrm{VC}_i$ be the set of objects in the $i$th virtual cache. The scope is contained in the set $O$. That is, $O$ is the collection of objects in the union of the real cache, and all $N$ virtual caches:

$$O = \mathrm{RealCache} \cup \bigcup_{n=1}^{N} \mathrm{VC}_n \tag{5.1}$$

Let $r_{n,o}$ be the position of object $o \in \mathrm{VC}_n$ relative to the lowest priority object when the set is sorted by priority into non-increasing order. An example of a list of a virtual cache's objects which are ranked by priority is given below.

| object | $o_5$ | $o_1$ | $o_{11}$ | $o_{12}$ | $o_3$ | $o_6$ | $o_2$ | $o_{15}$ | $o_9$ |
|---|---|---|---|---|---|---|---|---|---|
| priority | 4 | 12.5 | 8 | 1 | 3 | 16 | 3.14 | 5 | 12 |
| rank | 4 | 8 | 6 | 1 | 2 | 9 | 3 | 5 | 7 |

Object $o_{12}$ would be the first object to be discarded from the above virtual cache; $o_6$ is currently the last to be discarded. In the terminology of the Expert Framework this ranking is analogous to the experts' *prediction*. Each expert's weight and ranking can be combined to induce a master ranking over all objects in the scope of the master policy. This weighted combination of rankings is analogous to the inner product's role in Expert Framework for classification. Master priorities $P_o$, for each $o \in O$, are constructed by summing the weighted ranks of $o$ in each virtual cache:

$$P_o = \begin{cases} \sum\limits_{n:o\in \text{VC}_n} w_n r_{n,o} & \text{if } \exists n : o \in \text{VC}_n \\ 0 & \text{if } \forall n : o \notin \text{VC}_n \end{cases} \tag{5.2}$$

Notice that the contribution of each of the virtual caches is *not* normalized by a count of the number of objects they contain. Virtual caches with more (smaller sized) objects have a higher influence on the master rank, regardless of their weight.

Finally, objects are sorted into non-increasing order with respect to $P$. The master rank $R_o$ gives the position of $o$ from the lowest priority object in its sorted list. Based on this raking we define the on-line combined ideal cache. The *ideal cache* contains as many of the highest ranked objects, according to $R$, that fit in the real cache space (size(RealCache)).

$$\text{IdealCache} = \langle o_{R_1}, \ldots, o_{R_m} \rangle : \left\{ \begin{array}{c} \sum_{i=1}^{m} \text{size}(o_{R_i}) \leq \text{size(RealCache)} \\ \text{and} \\ \left( \begin{array}{c} \sum_{i=1}^{m+1} \text{size}(o_{R_i}) > \text{size(RealCache)} \\ \text{or} \\ \sum_{i=1}^{m} \text{size}(o_{R_i}) = \text{size}(O) \end{array} \right) \end{array} \right. \tag{5.3}$$

Operations on the real cache should proceed so as to make its contents match the ideal cache as much as possible, and at the lowest cost. At a high level this is accomplished by the following rules of thumb:

- Objects in the real cache ranked lowest by $R$ will be discarded first in order to make room for a new request.
- If resources permit, objects $o \in$ IdealCache – RealCache, which are ranked highest by $R$, are the next ones to be refetched.

```
foreach request $t$ do

        process the request on the virtual caches.
        update the weights $\mathbf{w}$.

        set $P_o = 0$, for all $o$ in scope $O$.
        foreach virtual cache VC$_n$ do
                Let $m = |\mathrm{VC}_n|$ .
                $r_n = \langle o_1, \ldots, o_m \rangle \leftarrow$ pointers to objects $o \in$ VC$_n$
                      sorted by priority.
                for $i = 1, \ldots, m$ do
                    $P_{o_i} = P_{o_i} + w_n * r_{n,o_i}$.
                end for
        end foreach
        $R = \langle o_{R_1}, \ldots, o_{R_m} \rangle \leftarrow$ pointers to objects $o \in O$ sorted by $P$.

        Let IdealCache = $\emptyset$.
        foreach $o_{R_i} : i = 1, \ldots, |O|$ do
                if (size(IdealCache) + size($o_{R_i}$) $>$ size(RealCache))
                  break foreach.
                end if
                Let $o_{R_i} \in$ IdealCache.
        end foreach

        process the request on the real cache.

end foreach.
```

Figure 5.2: Pseudocode for the master policy which uses virtual cache ranks to construct its own master ranking.

Figure 5.2 depicts the ranking operation described above in pseudocode. This illustration assumes that a dictionary manages the objects under the scope of the master policy. Each virtual cache maintains a list of pointers to objects in the dictionary, together with priorities. New requests are processed on each virtual cache, then the weights of the experts are updated. Before the request is processed in the real cache, the experts rankings are computed and combined. First, the master priorities ($P_o$) are initialized to zero. Next, the contents of each virtual cache are sorted by priority, and the resulting weighted ranking is added into the master priority for each object. Finally, the master ranking, $R$, is constructed by sorting all objects with respect to $P$.

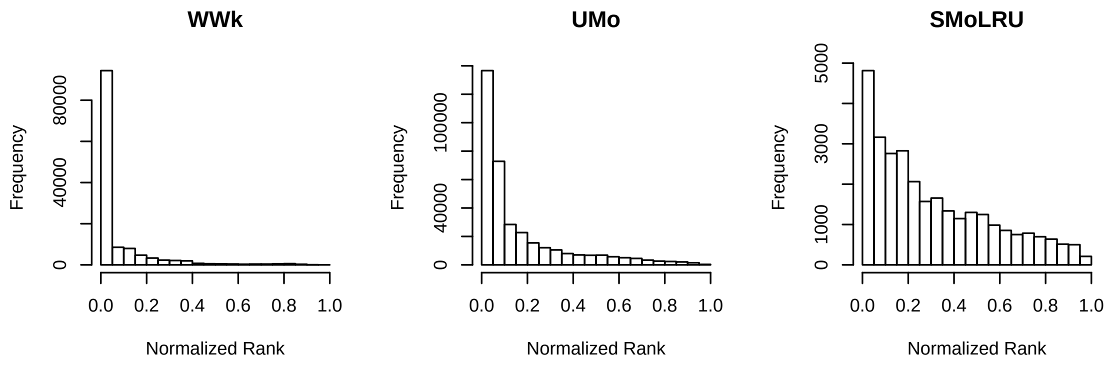

Figure 5.3: Histograms of the position of hits in the ranked ideal cache for three representative datasets (see Section 6.1). The ranks in the real cache (x-axis) were normalized by the total number of objects in the real cache, because the number can change over time. Frequency (y-axis) refers to the number of times an object in a bucket was hit.

To evaluate just how ideal the ideal cache is, we plotted histograms tallying how often hits were incurred at various positions in the master rank. Histograms for our three representative datasets are shown in Figure 5.3. Hits to objects in the ideal cache seem to decay with the object's rank. We take this as an indication that our combined ranking is sensible, but that there is also room for improvement (see the histogram for SMoLRU). Finding the best master ranking, and developing some theory for combining arbitrary rankings, is part of our ongoing research.

## 5.2 Demand v.s. Continuous Rollover

Each rollover strategy follows the same basic protocol: When space is needed to cache a new request, the master policy discards objects in the real cache with low master rank. This operation purges the real cache of objects which are either not present or have low priority in the virtual caches of currently favored policies. Discarding low ranked objects causes the content of the real cache to *rollover* to the content of the virtual caches which are governed by high weight policies. We call this strategy *demand rollover* because high ranked objects not in the real cache are only fetched on demand (i.e. when requested). In Section 6 we will show that a master policy based on demand rollover typically does about as well as than BestFixed. Figure 5.4 shows the demand rollover operation in pseudocode, picking up where Figure 5.2 left off.

Objects in the ideal cache are *never* discarded in order to make room for a new request. If enough room cannot be made by discarding objects from RealCache $-$ IdealCache, then the newly requested object is NOT fetched into the cache. In the terminology of Blum, et. al., [4] such an object is *rented*, rather than cached. The only time an object can be discarded from the ideal cache is if the memory allocated to virtually cached objects begins to encroach on the real cache space, and there are no objects in RealCache $-$ IdealCache which can be discarded to make room.[1]

```
foreach request $t$ do

        ⋮
        IdealCache = ... (as in Figure 5.2 and Eq. (5.3)).

        while ( free(RealCache) $<$ size($t$) ) do
              discard $o \in$ RealCache $-$ IdealCache with the lowest rank in $R$.
        end while

        if ( free(RealCache) $<$ size($t$) ) do
          fetch $t$ into RealCache

end foreach.
```

Figure 5.4: Processing requests on the real cache using demand rollover.

While we were initially happy to have developed a policy which works almost as well as BestFixed, we were ultimately not satisfied and wanted to do as well as

[1] This is not shown in Figure 5.4.

BestRefetching. We noticed that under demand rollover the content of the real cache *lagged behind* the better policies' virtual caches. We conjectured that "quicker" rollover strategies would improve overall performance. Our search for a better master policy began by considering an extreme and unrealistic rollover strategy that assures no lag time: After each request, refetch *all* of the highest ranked objects that were not retained in the real cache, discarding low ranked objects as necessary. In other words, take the real cache to be the ideal cache. As expected, the miss rate of this policy was greatly improved over demand rollover. We call this strategy *continuous rollover* because the real cache is continuously overhauled, request after request. The continuously rolling real cache is shown in Figure 5.5, picking up where Figure 5.2 left off.

```
foreach request $t$ do
        ⋮
        IdealCache = ... (as in Figure 5.2 and Eq. (5.3)).
        RealCache = IdealCache.
end foreach.
```

Figure 5.5: Processing requests on the real cache using continuous rollover.

Figure 5.6 shows how the master policy based on continuous rollover tracks the best policy over time, essentially hugging the bottom of the policy's window miss rates as plotted in Figure 1.1(a). Like BestRefetching, continuous rollover assumes a cost model where an arbitrary amount of refetching can essentially be done for free. Experimentally, a master policy employing continuous rollover always performs as well as the best on-line combination of policies, assuring no lag time when the mixture changes. Unfortunately, such a master policy is largely unrealistic. Miss rates obtained using continuous rollover are still interesting because they represent a lower bound on the number of misses suffered by our particular flavor of on-line policy. As will be shown in Section 6, continuous rollover outperforms BestFixed, and does nearly as well as BestShifting$(K)$ or BestShifting$(R)$, for large $K$ or $R$. Our next task is to present a compromise between demand and continuous rollover, which will lead to a more realistic rollover strategy with low lag time.

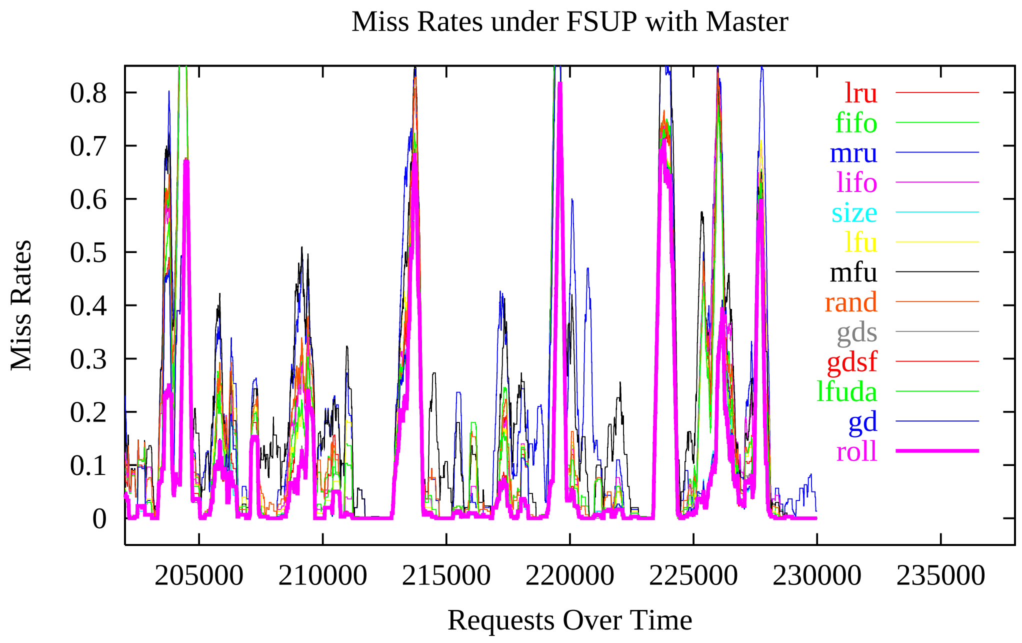

Figure 5.6: Continuous rollover based master policy (bold pink) tracks the best expert (on the 30,000 request snapshot of UMo, see Section 6.1).

## 5.3 Background Rollover

Continuous rollover can cause a large number of refetches even when weights of experts remain relatively constant over time. If all refetches are counted as misses, then the miss rate of such a master policy is comparable to that of BestFixed. We suspect that same holds for BestShifting and BestRefetching, as more shifts or refetches are allowed. However, from a user perspective, refetching is advantageous because of the latency advantage gained by having required objects in memory before they are needed. And, from a system perspective, refetches can be "free" if they are done when the system would otherwise be idle.

To take advantage of possible "free" refetches, we introduce the concept of *background rollover*. The exact criteria for when to refetch an object will depend heavily on the system, workload, and the expected cost and benefit of each object. To characterize the performance of background rollover without addressing these architectural details we allow the number of refetches per trial to be a Poisson (Pois$(\lambda)$) random variable, following the conventional assumption that the elapsed time between events (requests, in this case) are exponentially distributed. We experiment with rate parameters to the Poisson draws from $\lambda = 0.025$ (seldom refetching at all) to $\lambda = 10000$ (essentially continuous rollover). Only objects which reside in the ideal cache, but are currently not in the real cache, are refetched. Refetches proceed in order of rank $R$: To make room for refetched objects, objects not in the ideal cache are discarded, lowest rank first.

Background refetching techniques with $\lambda \geq 0.1$ typically gave fewer misses than BestFixed, approaching and nearly matching the performance obtained by the master policy using continuous rollover with $\lambda \geq 2$. Of course, techniques which slow down the redistribution of weights among experts (by tuning $\beta$ and $\alpha$, see Section 5.4.1) also tend reduce the number of refetches. In Section 6 we show that expensive continuous rollover is not required; only a small amount of refetching in the background, as time permits, is all that is needed to reap the full benefits of combining policies.

Figure 5.7 shows background rollover pseudocode which can be inserted after the ranking code in Figure 5.2. After each request is processed on the real cache, a draw $d$ is taken from Pois$(\lambda)$. The draw $d$ is the maximum number of objects which can be refetched during this trial. These objects should make up a subset of those which could be found in a continuously rolling real cache. $O_R$ contains the highest ranked objects which will be refetched ($O_R \subseteq$ IdealCache $-$ RealCache). Notice that $|O_R| \leq d$, and so the number of objects refetched each trial is $|O_R|$, and not $d$. Notice also that $\lambda = 0$ reduces this refetching strategy to demand rollover, and that $\lim_{\lambda \to \infty}$ is equivalent to continuous rollover. Only objects from RealCache $-$ IdealCache should be discarded to make room for a refetched object. Things have been set up such that size(RealCache $-$ IdealCache) $\geq$ size($O_R$), so making enough room for refetched objects should never

```
foreach request $t$ do

        ⋮
        IdealCache = ... (as in Figure 5.2 and Eq. (5.3)).

        while ( free(RealCache) $<$ size($t$) ) do
            discard $o \in$RealCache $-$ IdealCache with the lowest rank in $R$.
        end while

        if ( free(RealCache) $<$ size($t$) ) do
          fetch $t$ into RealCache

        Let $d = \text{Pois}(\lambda)$.
        Let $O_R =$ the $d$ highest ranked objects in IdealCache $-$ RealCache.
            ($|O_R| \leq d$)
        while ( free(RealCache) $<$ size($O_R$) ) do
            discard $o \in$ RealCache $-$ IdealCache with the lowest rank in $R$.
        end while

        fetch objects from $O_r$ into RealCache.
end foreach.
```

Figure 5.7: Processing requests on the real cache using background rollover.

be a problem.

Our experiments suggest that a minimal amount of refetching is essential for our master policy to approach BestRefetching$(R)$ in miss rate, for large $R$. In some cases, the leverage of our adaptive caching framework is partly contingent on being able to do a small amount of "free" refetching. Often, a small amount of refetching is quite feasible because many systems which cache data are not continuously inundated with requests. On the other hand, if the role of cache replacements on a system is to keep down the number of requests to objects on slower media, then refetching actually increases the number of these requests, which is undesirable. In a real system refetches should only be done when it can be determined that they have low cost. A more in-depth study of when refetching is feasible will be part of our future work.

## 5.4 Cost-Aware Refetching from the Ideal Cache

A key observation that we made while examining the ideal cache, and the objects refetched by more aggressive background rollover policies ($\lambda > 0.1$), was that only a small portion of objects refetched actually resulted in hits before eventually being discarded. Many lower priority objects made homes in the ideal cache with relatively short tenure. Often, objects with low priority in the real cache would toggle back and forth between ideal and discardable states. More aggressive refetching strategies would alternately refetch and discard many of these objects, every few requests, if the weights of the expert policies were changing rapidly. We soon noticed that most of the hits resulting from refetched objects came from those which were ranked in the top half of the ideal cache (see Figure 5.3). Refetched objects from this group proved to be worth the effort. Later, we will show that refetching from just the top 40-60% of the ideal cache left the miss rate essentially unchanged, and resulted in one-tenth of the number of refetches used by attempting to rollover to the entire ideal cache.

### 5.4.1 Tuning $\beta$ and $\alpha$

In the case where refetches are very costly we may wish to take further measures to keep their frequency down. This can be done by making the master policy less responsive to changes in the workload's characteristics. Setting $\beta \gg 1/e$ and $0 \ll \alpha < 1$ will cause the past average of weights to dominate the current distribution, and thus make major changes in the contents of the ideal cache less likely. As a result, policies which dominate for longer intervals of requests will gather a larger share of the weight. This will cause the distribution of weights to explore the simplex cautiously. Cautious exploration will yield an ideal cache with more inertia, which will require less refetching to keep up with. Unfortunately, curbing exploration will also yield miss rates with a less impressive improvement over BestFixed. Alternatively, setting, $\beta \ll 1/e$ and $0 < \alpha \ll 1$ will make the master policy sensitive to rapid changes in the characteristics of the request stream.

Thus, reducing the number of policy switches translates into increasing $\beta$. That is, with larger $\beta$ (lower learning rates) a baseline policy has to be doing really well on its virtual cache in order to be used by the master policy.

While $\beta$ controls the responsiveness of the master algorithm to changes in the workload, $\alpha$ controls the entropy of the distribution $\mathbf{w}$. Large $\alpha$ causes $\mathbf{w}$ to be more uniform, and thus have higher entropy. Small $\alpha$ causes $\mathbf{w}$ to be more skewed, resulting in a smaller working set of expert rankings. This leads to lower entropy distributions, and a more dynamic ideal cache. As a general rule, we have found that our master policies are far more sensitive to changes in $\beta$ than to changes in $\alpha$ in terms of the number of refetches, and the number of misses. However, tweaking $\alpha$, by starting with small $\alpha < 1/1000$ and then increasing it once a good $\beta$ has been set, can help reduce the miss rate of the master policy without significantly increasing the number of refetches.

That being said, we fix $\beta = 1/e$ and $\alpha = 5/1000$ for all experiments in this paper, as good results can still be can obtained without paying much attention tuning parameters.

# 6. Experimental Results and Discussion

Results presented here have an underlying theme: Miss rates of master policies, as described in the previous section, are measured against typically chosen policies like LRU, and against off-line comparators like those introduced in Section 3. The main suite of datasets we used in our experiments are described Section 6.1, and a table of statistics describing them is given for quick reference. As our master policies have several flavors, depending on the amount of refetching allowed, we illustrate our success graphically in Section 6.2.

## 6.1 Filesystem Data

| **Dataset:** | Work-Week (WWk) | User-Month (UMo) | Server-Month-LRU (SMoLRU) |
|---|---|---|---|
| #Requests | 138k | 382k | 48k |
| Cache size | 900KB | 2MB | 4MB |
| %Skipped | 6.5% | 12.8% | 15.7% |
| # Compuls | 0.020 | 0.015 | 0.152 |
| **LRU** Miss Rate | 0.166 | 0.076 | 0.870 |
| **BestFixed** Pol / MR | SIZE 0.055 | GDS 0.075 | GDSF 0.399 |
| %<LRU | 36.8% | 54.7% | 54.2% |

Table 6.1: Highlights of the CMU DFSTrace filesystem workloads used in our experiments.

Table 6.1 gives the highlights of the the three large datasets used in our experiments. These were gathered using Carnegie Mellon University's DFSTrace system [22] and had durations ranging from a single day to over a year. The traces we used represent a variety of workloads including a personal workstation (Work-Week, or WWk), a single user (User-Month, or UMo), and a remote storage system with a large number of clients, filtered by LRU on the clients' local caches (Server-Month-LRU, or SMoLRU). Small cache sizes were chosen to maximize the disparity between baseline policies and to exaggerate the potential benefit of combining policies. For each data set, the table shows the number of requests in the workload, the cache size used, the % of requests skipped because they were un-cacheable (larger than the real cache size, or the request was to meta-data), and the number compulsory miss rate. The table also shows which of our twelve baseline policies performed best (BestFixed), and the percent improvement of BestFixed over LRU (labeled '$\% <$LRU'). For each dataset, all 12 virtual caches consumed, on average, less than 2% of the total cache space.

The original traces provide information at the system-call level, and represent the original stream of access events, not filtered through any intervening caches. For these CMU traces we extracted file accesses based on file open requests. This assumes a data-object granularity for the analysis. We focus on patterns of file requests and are not concerned with intra-file access patterns.

It is common for systems to work with file accesses that have been filtered through prior caching stages. Under these conditions the workload would effectively be the misses of such cache stages. As mentioned in the Section 2.1.3, one of the many difficult systems problems is that of choosing an appropriate caching policy for a cache arbitrarily deep in an $N$-level or distributed cache. Server-Month-LRU (SMoLRU) is included as a representative dataset for such a situation. The requests in this workload have been pre-filtered by a 100KB LRU cache.[1]

## 6.2 Results, Graphically

Figure 6.1 depicts the degree of "adaptivity" of our ranking master policies on the WWk dataset by showing their miss rates up against the comparators outlined in Section 3. The basic background rollover strategy is labeled "Rank Ideal". Our most impressive results were obtained by restricting the background rollover policy to refetch only from the top 40% of ranked objects in the ideal cache (see Section 5.4). This curve is labeled "Rank 40% Ideal".

[1] This was done by our Storage Systems Group, here at UCSC.

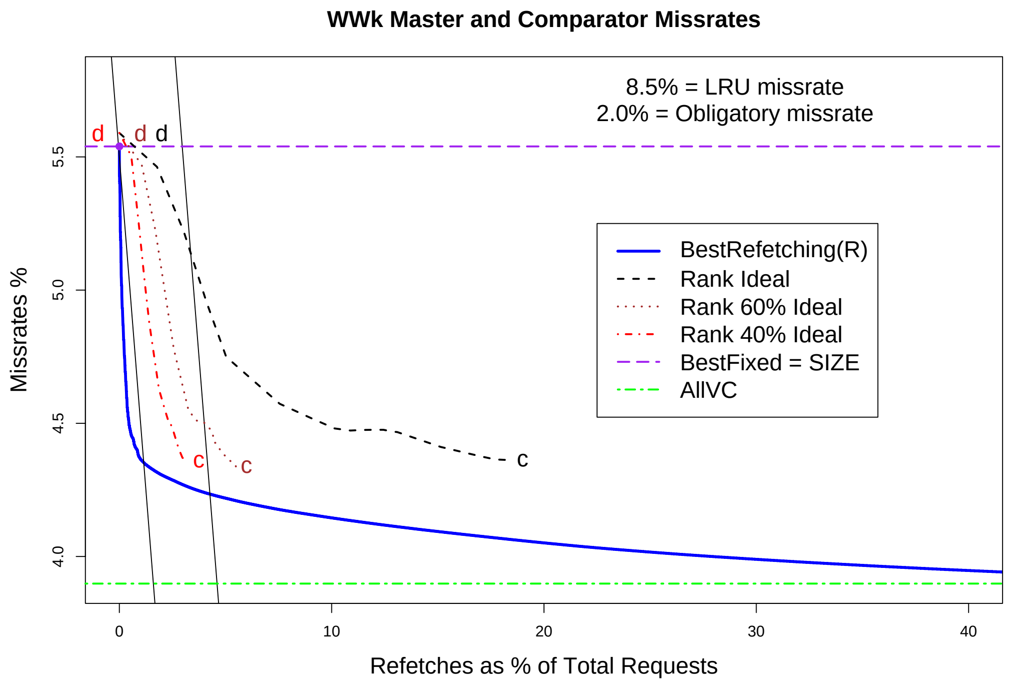

Figure 6.1: Online ranking master policies compared for WWk against off-line comparators, including LRU and AllVC. “d” marks the performance of demand rollover; “c” is for continuous rollover. Each master policy, labeled “Rank * Ideal”, is plotted for $\lambda = 0, \ldots, 10000$.

All curves in Figures 6.1-6.3 are plotted for varying $\lambda$– the parameter to the Poisson distribution which governs the number of refetches attempted in each trial. The x-axis represents refetches as a percent of the total number of requests. Notice that each master policy (“Rank * Ideal”) exhibits the same overall performance trends in terms of miss rate, but require different amounts of refetching.

The curve labeled “Rank 40% Ideal” in Figure 6.1 (for WWk) depicts a demand rollover-based master policy ($\lambda = 0$) which performs slightly worse than BestFixed, and background rollover policy ($\lambda \geq 0.025$) which beats it! Using $\lambda = 1$, corresponding to an attempt at, on average, one free re-fetch per request, “Rank 40% Ideal” gives miss-rate almost 21% better than BestFixed, and ends up requiring only one refetch after, on average, more than fifty requests. As $\lambda$ becomes large, its miss rate approaches

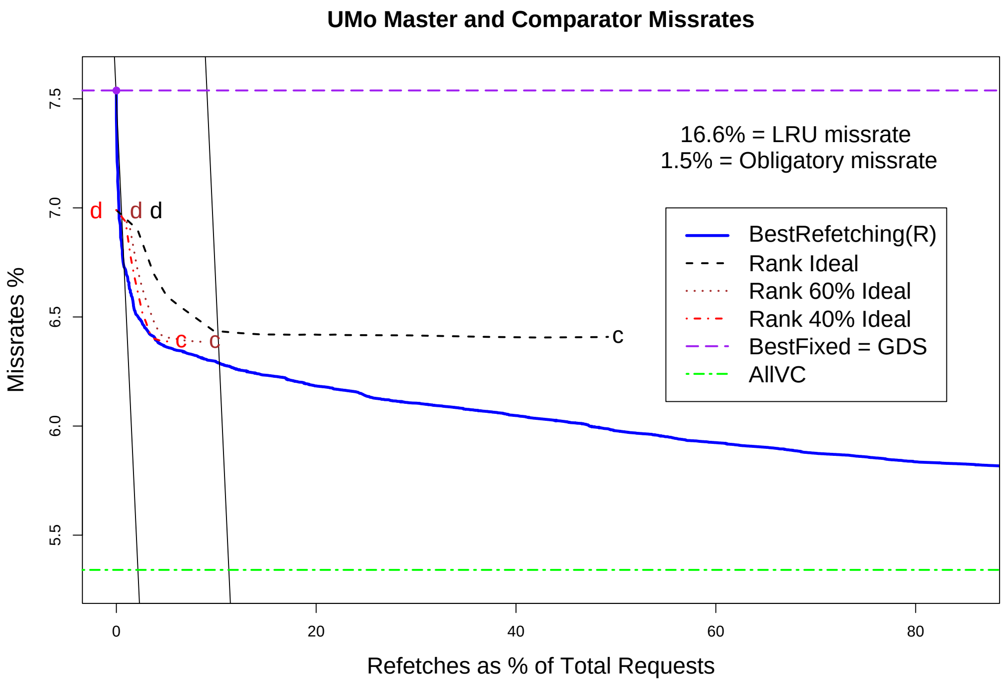

Figure 6.2: Online ranking master policies for UMo compared against offline comparators, including LRU and AllVC. "d" marks the performance of demand rollover; "c" is for continuous rollover. Each master policy, labeled "Rank * Ideal", is plotted for $\lambda = 0, \ldots, 10000$.

that of continuous rollover, which takes a free refetch in one out of every twenty requests. Regardless of the amount of refetching, continuous rollover on WWk has a miss rate $\sim 22\%$ better than BestFixed. All master policies do significantly better than LRU.

Using the regions outlined in Figure 3.4, our master policies on WWk fall into the lower portion of the B region, providing a significant savings in both fetches and misses over LRU. Also, a significant improvement over BestFixed was gained in terms of misses, with a slight increase in the total number of fetches. Discounting compulsory misses (accounting for over 2% of requests), our best policies have about one quarter fewer "real" misses than BestFixed and about half as many "real" misses as LRU.

Figures 6.2 and 6.3 show similar, and in some ways better, results for the UMo and SMoLRU datasets. These datasets are interesting for the following reason. In both cases $\lim_{R\to\infty}$ BestRefetching$(R)$ causes an order of magnitude more refetches than the number of requests in the workload. (The same holds for $\lim_{K\to\infty}$ BestShifting$(K)$.) In fact, to actually compute either of these comparators using dynamic programming proved to be too computationally intensive. Instead, the comparators were computed for refetch counts close to the total number of requests, and the curves were interpolated by including the AllVC comparator at the limit. Our on-line master policies, on the other hand, were able to take advantage of the major policy switches in the request stream, and match BestRefetching by using only a modest number of refetches.

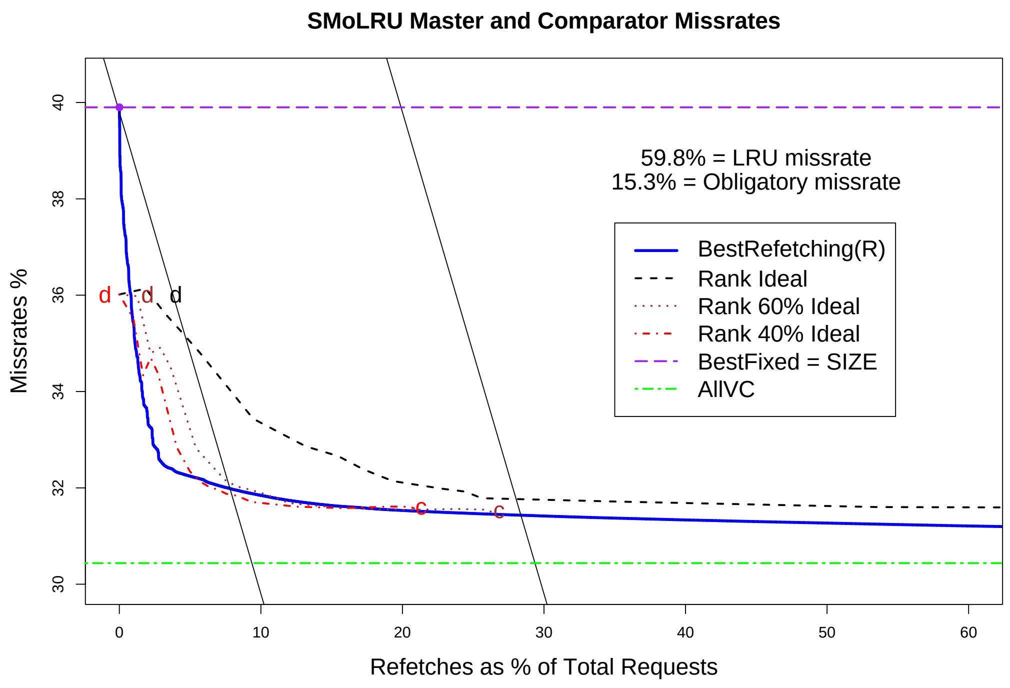

Figure 6.3: Online ranking master policies for SMoLRU compared against off-line comparators, including LRU and AllVC. “d” marks the performance of demand rollover; “c” is for continuous rollover. Each master policy, labeled “Rank * Ideal”, is plotted for $\lambda = 0, \ldots, 10000$.

UMo is our longest request stream. Demand rollover gives a master policy whose miss rate is better than BestFixed, placing it in the A region of Figure 3.4. A small amount of refetching allows “Rank 40% Ideal” to track BestRefetching (16% better than BestFixed), but refetches slightly more often than BestFixed, which places it in the B+ region. Compared to LRU, “Rank 40% Ideal” provides a significant reduction in user latency by achieving more than 61% fewer misses without increasing the total number of fetches.

Our results on the SMoLRU dataset are our most impressive. Demand rollover strategies, without doing any refetching, performed almost 10% better than BestFixed, putting it in the A region of Figure 3.4. With a small amount of refetching, “Rank 40% Ideal” was able to reach the A+ region, which corresponds to missing less requests than BestRefetching$(R)$, for modest $R$, while fetching less often than BestFixed. Such dramatic success on a workload whose requests have been pre-filtered through an LRU cache is very intriguing. This means that the previously difficult problem of choosing good policies for arbitrary nodes in a distributed cache can be completely automated. Experimenting with our master policies at *all* levels of a distributed cache will be part of our future work.

As promised, we have demonstrated that, with a small amount of (clever) refetching, beating BestFixed possible, both in terms of misses (which reduce end-user latency), *and* in terms of total I/O’s. It is even possible to achieve fewer misses than BestRefetching! We have developed a way of combing caching policies that, depending on the amount of refetching affordable, approaches in miss rate the best possible partition of policies into segments. Our master policies accomplish this using less actual cache space than that simulated by its virtual caches. Moreover, the framework is very flexible. Any number of caching policies, of any flavor, can be included as baseline policies. It is even conceivable that master policies themselves be included as a baseline policies in a meta-master policy.

# 7. Conclusion

Operating systems have many hidden parameter tweaking problems which are ideal applications for on-line Machine Learning algorithms. These parameters are often set to values which provide good average case performance on a test workload. For example, we have identified candidate parameters in device management, file systems, and network protocols. Previously, on-line algorithms for predicting as well as the best shifting expert were used to tune a time-out for spinning down the disk of a PC. Similar, yet less mature, on-line algorithms have recently been developed to determine the order in which pages should be discarded from a cache [4], and also to help choose the best policy for nodes in a distributed cache [1].

In this paper we attempt to determine, dynamically, the best caching policy over time, trading off past and current performance, measurable by many metrics. Virtual caches are an ideal proving ground for policies, and the weights of Expert Framework from on-line learning provide a good mechanism for judging the relative performance of policies over time. Our algorithms are simple, scalable, and fare well when measured against powerful off-line comparators. Viewing expert policies as rankings of the objects they cache allows a master policy to combine the policies' suggestions about which objects to retain, which to discard, and which to refetch when appropriate. In future work we plan to do a more thorough study of the feasibility of refetching in a variety of scenarios by building actual systems which use our master policy.

# Appendix A. Shifting Master Policy Managing the Real Cache

The discussion in this section reviews our initial work [12] in developing adaptive caching strategies with the help of the Expert Framework, and refetching. This approach involved exploring how the real cache could be managed by making use of the advice of the best shifting expert. This older approach is much more complicated, and gives less impressive results and has other obvious drawbacks. It is included here for completeness and comparison. The basic protocol of the master policy goes as follows:

1. Each request is processed on each virtual cache, and then loss and share updates of the expert framework are applied.
2. Next the request is processed on the real cache, based on the virtual cache contents of the single policy with the highest weight.

Just as before, we discuss three "rollover" strategies– demand, instantaneous, and background. The policy with the highest weight is referred to as the *governing policy*.

## A.1 Demand v.s. Instantaneous Rollover

At their core, each rollover strategy follows the same protocol: When space is needed to cache a new request, the master policy discards objects not present in the governing policy's virtual cache[1]. This causes the content of the real cache to "roll over" to the content of the current governing policy's virtual cache. We call this baseline strategy *demand rollover* because objects in the governing virtual cache are refetched into the real cache on demand. In Section A.3 we will show that a master policy based on demand rollover typically does as well, or slightly worse than BestFixed.

[1] We update the virtual caches before the real cache, so there are always objects in the real cache that are not in the governing virtual cache when the master policy goes to find space for a new request.

```
Given: $N$ baseline policies $p_1, \ldots, p_N$, and RealCache.size $= C$.
Init: $\mathbf{w}_0 = \mathbf{1}/N$, and Virtual Caches VC$_1, \ldots,$ VC$_N$ each of (virtual) size $C$.
      Let GoverningPolicy$(0) = p_1$.

Begin:
foreach request Object$_t$, $t = 1, \ldots, T$ do

        // Process Virtual Caches and update weights
        foreach virtual cache VC$_n = 1, \ldots, N$ do
                Process Object$_t$ on VC$_n$ using policy $p_n$,
                  obtain miss$_{t,n} \in \{0, 1\}$.
        end foreach.
        $\mathbf{w_t}$ = ExpertFrameworkUpdates($\mathbf{w}_{t-1}$, $\mathbf{miss}_t$).

        // Update Governing Policy
        Let $m = \max_i\{w_{i,t}\}$.
        Let GoverningPolicy$(t) = p_m$.

        /* // Instantaneous Rollover, optional
         * if (GoverningPolicy$(t) \neq$ GoverningPolicy$(t-1)$) then
         *    Discard all objects in RealCache not in VC$_m$.
         *    refetch objects in VC$_m$ not in RealCache, to capacity
         *            highest priority first.
         * end if.
         */

        // Process the request on the real cache
        if (Object$_t \notin$ RealCache) then
           discard from the RealCache an object not in VC$_m$
                  until enough space is made for Object$_t$.
           fetch Object$_t$ into RealCache.
        end if.

        // Resize the real cache
        RealCache.size $= C -$ size$(VC_1, \ldots, VC_N)$.
        while (size of objects in RealCache $>$ RealCache.size) do
              if (RealCache $\subseteq$ VC$_m$) then
                 discard the object from RealCache
                   with the lowest priority under $p_m$.
              else
                 remove from the RealCache an object not in VC$_m$
              end if.
        end while.

end foreach.
```

Figure A.1: Master policies employing demand or instantaneous rollover (shown in C-style comment).

As before, we conjectured that “quicker” rollover strategies would improve overall performance; and as before, our search for a better master policy began by considering an extreme and unrealistic rollover strategy that assures no lag time: After each switch of governing policy, *instantaneously* refetch all the objects in the new governing virtual cache that were not retained in the real cache. As expected, the miss rate was greatly improved over demand rollover. This strategy was dubbed *instantaneous rollover*.

Master policies employing both strategies, demand and instantaneous rollover, are outlined explicitly via pseudocode in Figure A.1. Demand rollover is shown as a baseline, and instantaneous rollover as commented out routine. Implementationally, there are four stages to be completed for each request: process the request on each virtual cache and update weights; if desired perform instantaneous rollover when the governing policy changes; process the request on the real cache; and then resize the real cache if necessary. In the code, $m$ indexes the current governing policy, or the governing policy’s virtual cache. Notice that the way we’ve set it up there is always a (non-empty) disjunction (the set RealCache – VC$_m$) between the real cache and the governing policy’s virtual cache containing objects large enough such that when discarded from the real cache enough room can be made for the new object. Finally, whenever discarding or refetching objects, we prefer to proceed by priority rather than arbitrarily.

A plot strikingly similar to Figure 5.6 in Section 5.2 shows how the master policy based on instantaneous rollover tracks the best policy. Instantaneous rollover makes all the same cost assumptions as continuous rollover from Section 5.2, and has all of the same desirable properties– unlimited free refetching guarantees that the master will always perform as well as its current governing policy. As before, instantaneous rollover outperforms BestFixed, and does almost as well as BestShifting$(K)$, but is functionally just another comparator representing what can be accomplished by switching policies.

```
Given: $N$ baseline policies $p_1, \ldots, p_N$, and RealCache.size $= C$.
        $r_m =$ the number of refetches to be done for each miss.
        $r_m =$ the number of refetches to be done for each hit.
Init: $\mathbf{w}_0 = \mathbf{1}/N$, and Virtual Caches VC$_1, \ldots,$ VC$_N$ each of (virtual) size $C$.
    Let GoverningPolicy$(0) = p_1$.

Begin:
foreach request Object$_t$, $t = 1, \ldots, T$ do

        // Process Virtual Caches and update weights
        // Same as baseline in Figure A.1

        // Update Governing Policy
        // Same as baseline in Figure A.1

        // Process the request on the real cache
        // And, determine how many objects to refetch: $r$
        if (Object$_t \notin$ RealCache) then
            discard from the RealCache an object not in VC$_m$
                    until enough space is made for Object$_t$.
            fetch Object$_t$ into RealCache.
            Let $r = \max\{r_m, |\text{RealCache} - \text{VC}_m|\}$.
        else
            Let $r = \max\{r_h, |\text{RealCache} - \text{VC}_m|\}$.
        end if.

        // Refetch, if appropriate
        for ($i = 1;\ i < r;\ i = i + 1$) do
            Let $O_i$ be the priority object in VC$_m$ − RealCache.
            discard from the RealCache objects not in VC$_m$
                    until enough space is made for $O_i$.
            refetch $O_t$ into RealCache.
        end for.

        // Resize the real cache
        // Same as baseline in Figure A.1

end foreach.
```

Figure A.2: Master policy employing background rollover.

## A.2 Background Rollover

Our approach to *background rollover* for shifting policies was much more kludgey in this work. The following (model) refetching strategies were examined: 1 refetch for every cache miss; 1 for every hit; 1 for every request; 2 for every request; 1 for every hit and 5 for every miss, etc. Essentially the same conclusions were reached here as compared to our more graceful approach using draws from a Poisson in Section 5.3:

a small amount of refetching in the background, as time permits, is all that's needed to reap the benefits of switching policies. Moreover, we concluded that refetching is essential for the master policy to outperform BestFixed. Figure A.2 shows background rollover pseudocode embedded in the baseline master policy of Figure A.1. $r_m, r_h \geq 0$ specify the maximum number of refetches that should be attempted following each miss, hit respectively. Using known priorities where applicable is desired here as well.

## A.3 Shifting Results, Graphically

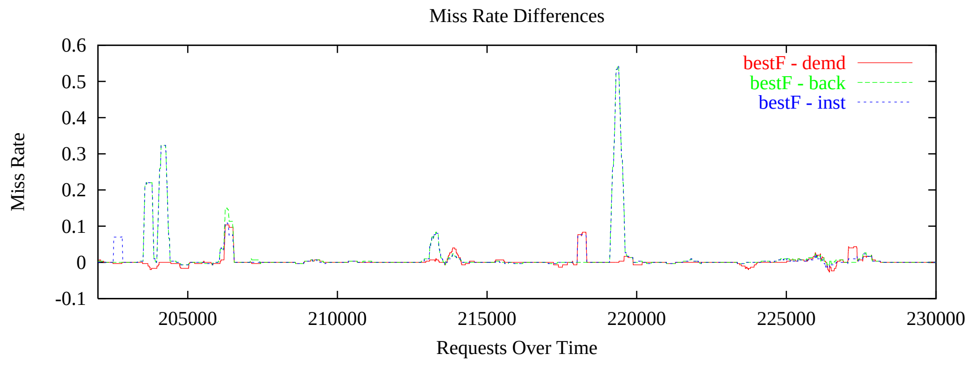

Figure A.3: BestFixed$-P$, where $P \in \{$Instantaneous, Demand, and Background Rollover$\}$. The baseline ($y = 0$) is BestFixed. Deviations from the baseline $y = 0$ show how the performance of our online shifting policies differ in miss rate. Above (Below) $y = 0$ corresponds to fewer (more) misses than BestFixed. (Snapshot of 30,000 requests from UMo, see Section 6.1).

Figure A.3 compares the performance of master policies based on demand, instantaneous, and background rollover with that of BestFixed by treating BestFixed as a baseline and plotting differences in miss rates in a rolling window of 300 requests. For any master policy $P$, BestFixed$-P > 0$ when $P$ is currently missing less objects than BestFixed in the window ending at time $x$. Notice in the figure that the demand rollover-based master policy is the only one of the three that dips (a little bit) under the $y = 0$ line, and that instantaneous and background rollover-based policies are always at or above $y = 0$, often incurring many fewer misses than BestFixed.

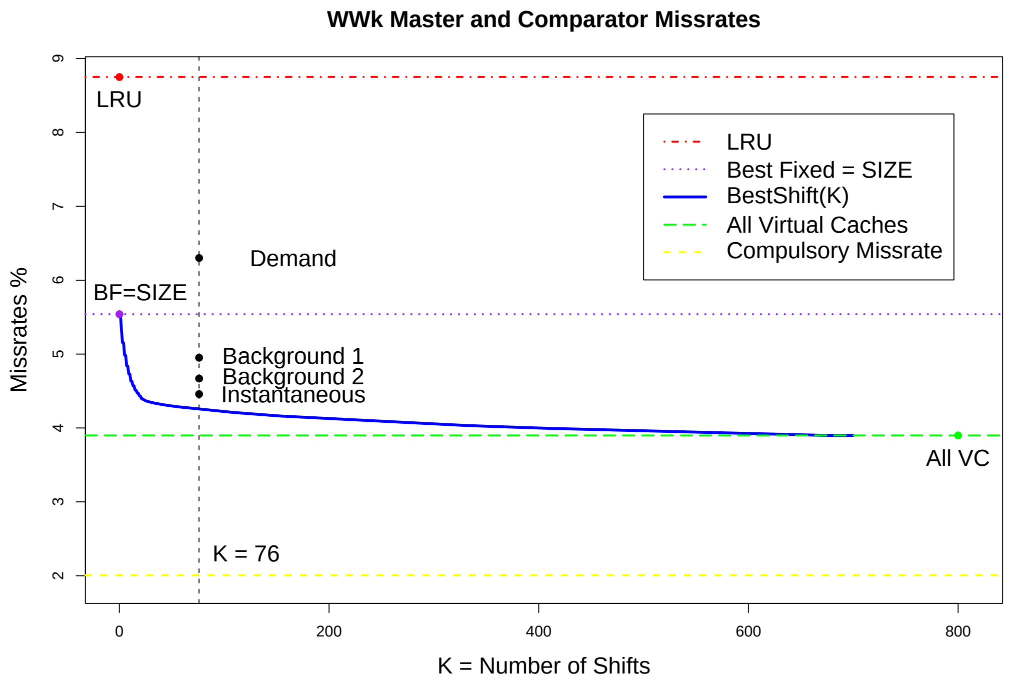

Figure A.4: Online shifting master policies employing Demand, Instantaneous and Background rollover compared against off-line optimal comparators, including LRU and AllVC.

Figure A.4 depicts the degree of "adaptivity" of each flavor of master policy by showing their miss rates up against the comparators outlined in Section 3. It shows that demand rollover does slightly worse than BestFixed, while background 1 (1 refetch every request) and background 2 (1 refetch every hit and 5 every miss) do better than BestFixed and almost as well as instantaneous rollover, which itself does almost as well as BestShifting. All of the policies do significantly better than LRU. Discounting compulsory misses, our best policies have about one third fewer "real" misses than BestFixed and about half as many "real" misses as LRU.

Setting $\beta = 1/e$ and $\alpha = 5/1000$ resulted in $K = 76$ switches of governing policy on the WWk request stream of 138,000 requests. This translates a switch of governing policy every 1,800 requests (on average). BestShifting$(K)$ on WWk (see Figures 3.1 & A.4) shows that $K \approx 30$ shifts chosen off-line suffices to get about a 20% decrease in misses over BestFixed, and as $K \to \infty$ there is only marginal further improvement totaling 29% fewer misses. Online refetching strategies like background 1 & 2 and instantaneous rollover give miss rates $\sim$20% better than BestFixed with slightly more shifts than necessary off-line.

Unfortunately, shifting policies online can be a double-edged sword. Like many online algorithms, its easy to imagine an adversary who can thwart a switching policy by altering the request stream's characteristics as soon as a new governing policy is realized. Master policy miss rates on adversarial data can not only be worse than BestFixed, but we believe it is possible to produce request streams which cause the master policy to perform worse than any of its baseline policies. This is possible because the general theory– particularly, the loss bounds– for the updates we borrowed from Expert Framework does not apply to caching.

## A.4 Full Tabular Results on Filesystem Data

Table A.1 summarizes the performance of our algorithms over three large datasets (see Section 6.1). We fixed $\beta = 1/e$, $\alpha = 5/1000$ for all experiments.

### A.4.1 Tuning $\beta, \alpha$, & Adding Hysteresis

The same comments in Section A.4.1 about tuning $\beta$ and $\alpha$ for slowing down the weights' exploration of the simplex in order to reduce the amount of refetching apply here as well. However, a more intuitive and straightforward way to control the number of rollovers, in this situation, is to add hysteresis. We put two kinds of thresholds on the criteria used to decide when to switch to a new governing policy. Like reducing the refetchable portion of the ideal cache, the general idea here is to keep the master policy from changing governing policies too quickly. Minimum and maximum thresholds are put on the weight vector $\mathbf{w}$ which must be met before a new policy can govern the real cache. Max-thresholds assert that a policy has to be *really good* before it can govern

|  | Dataset |  |  |
|---|---|---|---|
|  | Works Week | User Month | Server LRU |
| # Shifts | 88 | 485 | 93 |
| **Demand** Miss Rate | 0.063 | 0.076 | 0.450 |
| %<LRU | 28.1% | 54.4% | 48.5% |
| **%<BestF** | **-15.2%** | **-4.3%** | **-19.6%** |
| **BackRoll 1** Miss Rate | 0.050 | 0.068 | 0.401 |
| %<LRU | 43.5% | 59.4% | 55.5% |
| **%<BestF** | **12.0%** | **5.0%** | **12.1%** |
| **BackRoll 2** Miss Rate | 0.047 | 0.067 | 0.349 |
| %<LRU | 46.8% | 60.1% | 60.3% |
| **%<BestF** | **17.0%** | **8.7%** | **16.6%** |
| **Instant** Miss Rate | 0.046 | 0.065 | 0.322 |
| %<LRU | 49.1% | 60.8% | 63% |
| **%<BestF** | **18.8%** | **13.6%** | **18.9%** |
| **BestShifting** Miss Rate | 0.042 | 0.039 | 0.312 |
| %<LRU | 52.2% | 48.0% | 30.1% |
| **%<BestF** | **23.6%** | **48.7%** | **21.8%** |

Table A.1: Performance summary on CMU DFSTrace filesystem workloads.

the real cache. Min-thresholds prevent the situation where none of the policies have extremely high fitness but somehow the policy governing the real cache starts doing very poorly. When the min-threshold is met, control of the real cache transfers to the current best policy. Only when one of these two thresholds are met can a switch in governing policy occur. We have found that a good rule of thumb is to set the maximum thresholds somewhere between 90% and 95%, and take $1-$max-thresh for the minimum threshold. Figure A.5 shows code implementing this heuristic, replacing the part of the code used to update the governing policy given in Figures A.1 and A.2.

Hysteresis thresholds like these are heuristics which help prevent oscillations between states in an attempt to find modes of operation with inertia. Threshold driven hysteresis is rampant in control theory, implemented in thermostats, and used to help model a wealth of naturally occurring phenomena. Of course, such heuristics can still be thwarted by adversarial conditions. Still, we find that adding min/max thresholds into the master policy helps it to select larger segments, resulting in less refetching.

The impact of thresholds on the on the miss rate is low, but not negligible. Thresholds give master policies with less aggressive refetching strategies more time to resolve

```
⋮
Given: $\ldots$, thresholds $0 <$maxthresh $>$ minthresh $< 1$
⋮
foreach request Object$_t$, $t = 1, \ldots, T$ do
        ⋮
        // Update Governing Policy
        Let $m' = \max_i\{w_{i,t}\}$.
        if ($w_{m'}$ >maxthresh OR $w_m$ <minthresh) then
            Let $m = m'$.
            Let GoverningPolicy$(t) = p_m$.
        ⋮
        end if.
end foreach.
```

Figure A.5: Master policy with hysteresis thresholds, augmenting Figures A.1 & A.2.

virtual/real cache lag, making the miss rates of such policies less likely to fall below BestFixed. Loose thresholds encourage policy switches, and may require aggressive refetching for the best results. When refetching is costly, strict thresholds make the master policy more conservative, switching its policy only when a potential candidate has proven itself, or when the current governing policy has clearly fallen out of favor.

### A.4.2 Comparing Ranking with Shifting

Everything else being equal, Figure A.6 shows why were prefer ranking to shifting. The points labeled “Shift Dmd”, “B1”, “B2”, “Inst”, represent demand rollover, background rollover strategy 1, and 2, and instantaneous rollover respectively. Shifting gives master policies which fall into the B- and C regions labeled in Figure 3.4. We prefer ranking because it refetches less often, gives lower miss rates, and seems more natural (which is why it is easier to explain). Its as simple as that!

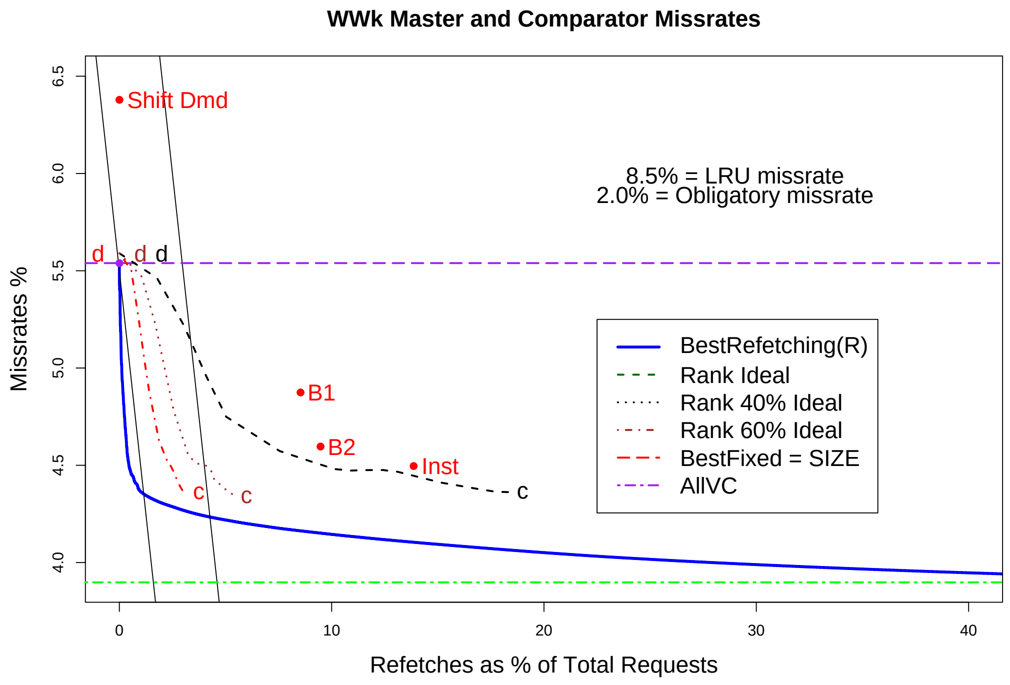

Figure A.6: Online shifting master policies employing Demand, Instantaneous and Background rollover compared against off-line optimal comparators, including LRU and AllVC, plotted on the refetch scale.

## References

[1] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. ACME: adaptive caching using multiple experts. In *Proceedings of the 2002 Workshop on Distributed Data and Structures (WDAS 2002)*. Carleton Scientific, 2002. Extended version of the WDAS 2002 workshop paper.

[2] Martin Arlitt, Ludmilla Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for Web proxy caches. In *Proceedings of the Workshop on Internet Server Performance (WISP99)*, May 1999.

[3] Avrim Blum. On-line algorithms in machine learning. In *Online Algorithms*, pages 306–325, 1996.

[4] Avrim Blum, Carl Burch, and Adam Kalai. Finely-competitive paging. In *IEEE Symposium on Foundations of Computer Science*, pages 450–458, 1999.

[5] Allan Borodin, Nathan Linial, and Michael E. Saks. An optimal on-line algorithm for metrical task system. *Journal of the ACM (JACM)*, 39(4):745–763, 1992.

[6] Olivier Bousquet and Manfred K. Warmuth. Tracking a small set of experts by mixing past posteriors. In *COLT/EuroCOLT*, pages 31–47, 2001.

[7] Mudashiru Busari and Carey Williamson. Simulation evaluation of a heterogeneous web proxy caching hierarchy. In *Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '01)*, pages 379–388, Cincinnati, OH, August 2001. IEEE.

[8] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *ACM SIGMETRICS*, pages 171–182, June 1995.

[9] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, 1997.

[10] N. Cesa-Bianchi, Y. Freund, D. Haussler, D. P. Helmbold, R. E. Schapire, and M. K. Warmuth. How to use expert advice. *Journal of the ACM*, 44(3):427–485, 1997.

[11] Nicolò Cesa-Bianchi, Yoav Freund, David Haussler, David P. Helmbold, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. *Journal of the ACM*, 44(3):427–485, 1997.

[12] Robert B. Gramacy, Manfred K. Warmuth, Sott A. Brandt, and Ismail Ari. Adaptive caching by refetching. In *Advances in Neural Information Processing Systems (NIPS 2002)*. Morgan Kauffman, 2003. (to appear).

[13] James Griffioen and Randy Appleton. The design, implementation, and evaluation of a predictive caching file system.

[14] D. Haussler, J. Kivinen, and M. K. Warmuth. Sequential prediction of individual sequences under general loss functions. IEEE Transactions on Information Theory, 44(2):1906–1925, September 1998.

[15] David P. Helmbold, Darrell D. E. Long, Tracey L. Sconyers, and Bruce Sherrod. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, 5(4):285–297, 2000.

[16] Mark Herbster and Manfred K. Warmuth. Tracking the best expert. In *International Conference on Machine Learning*, pages 286–294, 1995.

[17] Shudong Jin and Azer Bestavros. Greedydual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. Technical Report 2000-011, 4, 2000.

[18] J. Kivinen and M. K. Warmuth. Additive versus exponentiated gradient updates for linear prediction. *Information and Computation*, 132(1):1–64, January 1997.

[19] J. Kivinen and M. K. Warmuth. Averaging expert predictions. In Paul Fischer and Hans Ulrich Simon, editors, *Computational Learning Theory: 4th European Conference (EuroCOLT '99)*, pages 153–167, Berlin, March 1999. Springer.

[20] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, 1994. An early version appeared in FOCS 89.

[21] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies (FAST 03)*, March 2003.

[22] Lily Mummert and Mahadev Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software - Practice and Experience (SPE)*, 26(6):705–736, June 1996.

[23] A. J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[24] V. G. Vovk. A game of prediction with expert advice. In *Proc. 8th Annu. Conf. on Comput. Learning Theory*, pages 51–60. ACM Press, New York, NY, 1995.

[25] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 161–175, Monterey, CA, June 2002. USENIX.