On combining disjoint caches

Robert B. Gramacy
rbgramacy@cse.ucsc.edu
University of California at Santa Cruz
CS290C, Advanced Topics in Machine Learning
March 28, 2003

Abstract

Following the ideas of Megiddo & Modha [9] for developing ARC (Adaptive Replacement Cache), we study how to best combine disjoint lists of pages in order to construct an adaptive paging algorithm that has a miss rate lower than LRU. This paper contains our preliminary results. The approach we use takes advantage of the Expert Framework from online learning[7, 5, 8, 4, 3]. It is also an interesting application of Path Kernels [11], which are used to keep track exponentially many combinations of the disjoint lists implicitly, by only maintaining polynomially many weights. We introduce the concept of rollover [6] in order to better track a dynamically changing cache. We compare our results to both ARC and LRU (the de facto standard policy which discards the least recently used page).

1 Introduction & Related Work

The problem is caching for fixed block sizes (a.k.a. paging). In paging there is a cache which can hold $k$ pages, and pages are requested from the cache. When a request is made for a page not in the cache, it must be fetched into the cache, before it can be forwarded to the requester. If the cache is currently full, e.g. it already contains $k$ pages, a page from the cache must be chosen to discard, in order to make room for the requested page. This operation is called page replacement.

The choice of which page to discard, in order to make room for a newly requested page, is handled by a page replace policy. Usually, the goal of a page replacement policy is to minimize the number of pages that need to be fetched relative to the total number of requests. When requests are made to pages not in the cache this is called a miss, otherwise we refer to the request as a hit. So, in other words, the goal of a page replacement policy is (usually) to minimize the miss rate, or (alternatively) maximize the hit rate of requests.

A commonly chosen page replacement policy is Least Recently Used (LRU). LRU is a heuristic that discards the least recently requested page from the cache in order to make room for a new request. Of course, there are many other policies out there that take criteria other than recency into account: e.g. frequency, fetch time, fetch cost, etc., or combinations of these. Which policy is best depends on the workload of requests. The best policy can not only vary between workloads, but it can also vary within a single workload. Choosing a page replacement policy can be a daunting task.

Recently, a great deal of systems research has been focused on creating adaptive page replacement policies. Informally, adaptive page replacement policies are policies which adapt their replacement criteria based on the observed sequence of requests. Unfortunately, adaptive is seldom clearly defined, and little theoretical or experimental evidence is ever given that policies are good at adapting.
Even more recently, there has been a push by the on-line learning community to apply the Expert Framework [7, 5, 8, 4, 3] to systems problems. Specifically, there has been a push to use the framework to help develop more “adaptive” page replacement policies, and to better quantify what it means for a policy to be theoretically or experimentally [2, 1, 6] adaptive. Such approaches involve combining the criteria of baseline replacement policies (e.g. LRU, etc.) in order to develop a master policy which achieves fewer misses than any of its baseline policies, or perhaps performs almost as well as the best off-line partition of replacement policies into segments of the request stream. It is in this sense that such a master policy is deemed successfully adaptive. The basic idea is to use the weights of the Expert Framework, and treat the baseline policies as experts, where the weight corresponding to each expert represents the master policy’s belief in the pages cached by the corresponding baseline policy.

A recently developed “adaptive” cache replacement policy for fixed block sizes, called ARC (Adaptive Replacement Cache) [9], has piqued our interest. Typically, LRU is the best policy in this setting. However, frequency based policies (e.g. LFU: Least Frequently Used) occasionally perform well. Essentially, ARC maintains two lists (LRU1, LRU2) where objects are ordered by recency. The total number of objects cached are split between the two lists. Objects are initially placed into LRU1. Only objects residing in LRU1, that have been accessed again recently are moved to LRU2. Thus, LRU2 contains both frequency and recency information. A parameter $p$ allows the total size of each list to be adjusted relative to the size of the full (combined) cache. Actually, ARC maintains information about twice as many objects as can fit in the cache in order to make more informed movements of the objects between lists. A heuristic is provided which updates the parameter $p$ when the pages in the request stream seem to be favoring a frequency (recency respectively) based list (or policy). The authors report that ARC has on average 15-40% lower miss-rates than LRU.

We are intrigued by this approach because we see two different ways of using the Expert Framework from machine learning to help ARC determine the best partition parameter ($p$) over time. Firstly, ARC($p_i$), for fixed parameters $p_1, \ldots, p_N$, can be used as experts. The caching framework of Gra-mcy, et. al. [6] could be used to switch between fixed ARC($p_i$) policies, thus reducing it to an already solved problem. Secondly, we believe that we can use machine learning approaches to find a better online update of the parameter $p$ (using a single ARC policy). This, more straightforward, approach maintains only one ARC cache but treats each possible $p \in \{1, \ldots, \text{cachsize}\}$ as an expert, and uses updates from the Expert Framework to choose the best $p$ online.

In the following section we will quickly review our experimental results obtained using the first approach: fixed ARC($p_i$)’s as experts. The rest of the paper will be dedicated to explaining and experimenting with the second approach. We will give the expert-based update of the $p$ parameter, briefly discuss the concept of rollover for making ARC more responsive to changes in $p$, and then generalize our expert-based ARC to work with 3 (or more) lists. Finally, since this work is very preliminary, we will conclude with a detailed list of future work to be done.

2 Shifting between fixed ARC’s

Without going into the details of ARC, we consider $N$ ARC($p_i$) experts, each with a different partition parameter $\{p_1, \ldots, p_N\}$. The parameter $p_i$ roughly determines the number of objects from the list LRU1 which will be include in the $i$th expert’s cache. Depending on the total number of experts, $N$, we let the $p_i$’s partition the cache space $k$ evenly.

$$p_i = \frac{i \cdot k}{N}$$

The remaining $k - p_i$ cached objects will be taken from LRU2, where $k$ is the total number of pages that can fit into the cache. To make a long story short, and without going into any details, as the number of experts is increased ($N \rightarrow k + 1$) the miss rate of the master policy of the framework approached that of normal ARC (which was allowed to adapt its own $p$ parameter using the default machinery).
2.1 Best Shifting fixed ARC’s

To get an idea of why the framework of Gramacy, et. al., was relatively unsuccessful we can consider the best off-line partition of ARC($p_i$) policies into segments of the workload. Things like this are done all the time in the machine learning literature in order to explore the amount of improvement that can be achieve by an on-line shifting algorithm.

Figure 1 shows how the best off-line partition of expert ARCs into segments causes a decrease in the miss rate as more segments are allowed. Three different cache sizes are shown. The figure illustrates that many shifts in policy are required to get a significant reduction in miss rate. For smaller caches the decrease in miss rate is almost linearly related to an increase in the number of policy shifts. Research by Gramacy, et. al. [6], indicates that a much quicker (off-line) drop in miss rate per shift is required in order for a master policy to be able to exploit such shifts on-line.

3 Expert based ARC

Using $N$ fixed ARC($p_i$) experts has several disadvantages, and is in some sense more clumsy than it needs to be. First of all, implementing $N$ ARC policies, even if its done “virtually”, means a lot of overhead. Secondly, there is a great deal of overlap between experts with similar $p_i$’s. And since all of the experts are essentially the same policy (with a different partition parameter) it should be possible to move the Expert Framework inside of the actual ARC policy, to where the parameter affects its replacement criteria directly.

We should like to be able to implement a single ARC policy, and somehow use the Expert Framework to help choose the $p$ parameter. Before discussing our general idea in more detail, a brief overview of ARC is required. For the details, see Megidlo & Modha [9].

Two disjoint lists of pages are maintained: LRU1 and LRU2. Both lists order pages by recency. Requests to pages which are not in either list are put into LRU1, and fetched into the cache. Pages in LRU1, which are requested again, are moved into LRU2. The pages kept in the cache make up the most recently accessed pages from each list. A partition parameter, $p \in \{0, \ldots, k\}$, determines the number of pages from LRU1 should be kept in the cache. Likewise, $k - p$ determines the number of most recently accessed pages from LRU2 which should be kept in the cache. Whenever there is a request to a page which is in LRU1 or LRU2, but is not in the cache (e.g. the page has been accessed less recently than the $p$ most recently accessed pages in LRU1, and likewise, for $k - p$ in LRU2) a heuristic is provided for updating the $p$ parameter. Essentially, if the requested page was in LRU1, but not in the cache, $p$ is made larger. Otherwise, if the requested page was in LRU2, but not in the cache, $p$ is made smaller. Pages are replaced from the cache in accordance with the partition parameter $p$.

Our idea is to use the Expert Framework from on-line learning in order to replace ARC’s heuristic for updating the $p$ parameter, by something that is more theoretically motivated. We associate each possible $p \in \{0, \ldots, k\}$ with an expert ($k + 1$ experts total). Each expert represents a possible partition of the lists LRU1 and LRU2. The master policy uses weights $\mathbf{w} = \langle w_0, \ldots, w_k \rangle$ to encode its belief in each expert’s (static) choice of partition parameter. Based on these weights, the master maintains its own partition parameter, $p_M$, which is computed as a weighted average of the expert partitions:

$$p_M = \sum_{i=0}^{k} w_i p_i$$

The weights of the Expert Framework lie on the $k+1$ dimensional probability simplex, $\mathcal{P}^{k+1}$, and so $p_M$ is always a valid partition. Whenever the master suffers a miss because the $p$ parameter was either too high or too low, the master updates the weights of the experts, thereby updating its own partition parameter, $p_M$. This is the topic of the next subsection.

3.1 Updates

Whenever the master policy misses a request because its partition parameter $p_M$ is too high or too low, the weights its experts are updated. Figure 2 shows
the two situations when that would happen. Either the requested page is in LRU1, but not among the \(p_M\) most recently requested pages. Or, the requested page is in LRU2, and its not among the \(k - p_M\) most recently requested pages.

The master policy needs to update the weights, thereby updating its belief in the partition parameter of each expert. Experts whose corresponding partition parameter has the requested page fall outside of the cache (resulting in a miss) should have their weight reduced, whereas experts whose partition parameter results in a hit should have their weight increased. The Expert Framework uses multiplicative updates to accomplish this. “Bad” experts have their weight multiplied by a factor \(\beta \in (0,1)\). “Good” experts weights are left unchanged, except implicitly via normalization. This is usually referred to as the loss update. Formally, after request \(t\), we compute the weight \(w_{t,n}^m\) as follows, based on the loss of \(n\)th expert:

\[
\begin{align*}
  w_{t,n}^m &= \frac{w_{t-1,n} \beta \text{loss}_{t,n}}{Z_{t+1}}, \\
  Z_{t+1} &= \sum_{n=1}^{N} w_{t,n} \beta \text{loss}_{t,n},
\end{align*}
\]

\(1\)
for \( n = 0, \ldots, k \), where \( \beta \in (0, 1) \) and \( \text{loss}_{n} = 1 \) if the \( n \)th expert is a “bad” expert, and 0 if it is a “good” expert. A small \( \beta \) parameter causes high weight to decay quickly if its corresponding partition starts incurring more misses than other partitions with high weights. The initial weight distribution is uniform, i.e. \( w_{1,i} = 1/N \).

As noticed by Herbster and Warmuth [7], 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. There are a number of share updates [7, 3] with various recovery properties. We chose the Fixed Share to Uniform Past (FSUP) update because of its simplicity and efficiency, and because of its ability to fixate on a small sub-pool of experts, encoding our prior belief that the workload’s characteristics are exploited by a smaller sub-pool of the expert partitions.

The FSUP update mixes the current weight vector with the past average weight vector \( r_{t} = \sum_{t=1}^{T} w_{t}^{m} / t \). This is easy to compute on-line without using more than a constant amount of memory per expert:

\[
\begin{align*}
    w_{t+1} &= (1 - \alpha)w_{t}^{m} + \alpha r_{t-1} \\
    r_{t} &= \frac{(t - 1)r_{t-1} + w_{t}^{m}}{t}
\end{align*}
\]

where \( \alpha \) is a parameter in \((0, 1)\). The higher \( \alpha \) is the more quickly previously good policies will recover. In our experiments we used \( \beta = 1/c \approx 0.37 \) and \( \alpha = 5/1000 = 0.005 \).

We also experimented with a share update that we call Fixed Share to Neighbor (FSTN). where each expert shares a fraction \( (\alpha) \) with experts whose partition parameter is one larger and one smaller than itself. FSTN gave almost identical results compared to FSUP, and so the results of these experts are omitted.

### 3.2 Results & Rollover

The updates for the Expert Framework were not designed with caching or paging in mind. Changing the master partition parameter \( p_{M} \) to be the same as the best experts does not guarantee that the miss rate of the master policy will be the same as the best experts. Why is this the case? After an update of the weights, \( p_{M} \) can be quite different than it was before. However, a change in \( p_{M} \) might not immediately change the performance of the master algorithm, because changing \( p_{M} \) does not change which objects are cached. \( p_{M} \) simply represents a target size for the number of cached objects from LRU1. In order to make the master policy more responsive to changes in its partition parameter, \( p_{M} \), we introduce the operation of rollover.

Figure 3 shows ARC (standard-ARC) compared with expert-ARC, plotted for varying cachesize. While both flavors of ARC are better than LRU, expert-ARC lags behind standard-ARC in terms in reduction of miss rate per cache size. This is because
Comparing (Expert) ARC and LRU

Figure 3: Initial results comparing the Expert Framework based ARC (expert-ARC) with its standard formulation (standard-ARC), and LRU.

$p_M$ changes too aggressively. Expert-ARC assumes that changes in $p_M$ immediately result in more (or less) objects in LRU1. However, it may take several requests before the number of cached pages at the head of LRU1 matches its goal number of $p_M$.

After the weights of the experts are updated, corresponding to a change in the master partition parameter, $p_M$, we suggest immediately altering the number of pages from LRU1 and LRU2 which are cached, in order to comply with the partition. That is, if $p_M$ increases, pages from LRU2 are discarded in order to make room for pages from LRU1 which need to be refetched. Likewise, if $p_M$ decreases, pages from LRU1 are discarded to make room for refetched LRU2 pages. We use the term refetch to distinguish pages which have been discarded previously (and reside in the un-cached portions of LRU1 or LRU2), which need to be fetched, in contrast to other pages, not in LRU1 or LRU2, which are simply fetched on demand.

Rollover is an operation that can apply to both expert-ARC and standard-ARC. As Figure 4 shows, expert-ARC is best when used with rollover. Rollover models the situation where pages can be refetched for free before they are requested. While this may result in an increase in I/O’s (input/output, fetches), expert-ARC with rollover has a lower miss rate than ARC with (or without) rollover. All together, refetches made up 5-15% of the total number of requests (counted as fetches) for the expert-ARC with rollover. Standard-ARC with rollover caused about half as many refetches as expert-ARC.

Figure 4: Initial results comparing the Expert Framework based ARC (expert-ARC) with its standard formulation (standard-ARC), where both policies are allowed to use rollover.
3.3 Three Lists

ARC is designed to trade off recency and frequency by moving pages between two lists, and its "adaptability" is provided by a heuristic which alters the partition parameter \( p \). It is easy to imagine adding another list (LRU3) to an ARC-like framework. Doing such might result in a more finely-granulated notion of frequency, when trading off recency. Objects in LRU2 which are hit could be moved into LRU3.

The protocol provided by ARC for moving objects between three lists is easily generalized. But now, two partition parameters need to be chosen: \( p_1, p_2 \geq 0 \), where \( 0 \leq p_1 + p_2 \leq k \). The \( p_1 \) most recently used objects from LRU1, \( p_2 \) from LRU2, and \( k - p_1 - p_2 \) from LRU3, would ideally determine the cached pages from each list. Unfortunately, the heuristic provided by ARC for modifying the partition parameter, \( p \), does not so easily generalize.

Luckily, using the Expert Framework for choosing a master partition parameter \( p_M \) does generalize when there are three lists. \((k+1)/2\) experts can be created, for all possible pairs \( p_1, p_2 \). And then the same weight updates (for \( k(k+1)/2 \) weights) can be computed, as given in Section 3.1. It helps to have a convenient way of mapping experts to partitions \( p_1, p_2 \). Figure 5 shows the way we map the experts to partitions.

Just as before, the master partition \( p_M = \langle p_{M_1}, p_{M_2} \rangle \) is constructed by taking a weighted average of the expert partitions

\[
p_M = \langle p_{M_1}, p_{M_2} \rangle = \sum_{i=1}^{(k+1)/2} \text{expert}(i).
\]

Figure 6 shows how using a three-list expert-ARC page replacement policy measures up against the results we’ve displayed (for two lists) thus far. It is interesting to notice that for some cache sizes three lists is best, but for most cache sizes, two lists are better than three. There are even places where three lists is worse than LRU. This could be an indication that there is a bug in our generalization of ARC to three lists, or the rollover code for three lists. We doubt that there is a bug in the expert-weight code.

\[
i = 0.
\]

for \( p_1 = 0, \ldots , k \)
\[
\text{for } p_2 = 0, \ldots , k
\]
\[
\text{if } p_1 + p_2 > k
\]
\[
\text{continue.}
\]
\[
\text{else}
\]
\[
i = i + 1.
\]
\[
\text{end if}
\]
\[
\text{expert}(i) = \langle p_1, p_2 \rangle.
\]
\[
\text{end for}
\]
\[
\text{end for}
\]

Figure 5: Mapping of experts to partitions \( \langle p_1, p_2 \rangle \).

That code is very straightforward. When three-list expert-ARC beat two-list expert-ARC it refetched about twice as often, and vice-versa.

3.4 Many Lists

The approach in the previous section for using three lists requires experts (weights) on the order of \( k^3 \). Generalizing ARC to work with an arbitrary number of \( \ell \) lists is straightforward, but would require the creation of \( O(k^{\ell-1}) \) experts. The time and space needed to implement such experts could be prohibitive even for modest \( k \), and relatively small \( \ell \).

Luckily, path kernels [11] come to the rescue. With path kernels it is possible to maintain \( O(\ell k) \) weights for an implicit \( O(k^{\ell-1}) \) experts. This is because the loss of each expert depends on which of the \( \ell \) lists, and in what position \( i = 1, \ldots , k \) the requested object is on. Weight updates could be made on the list-position pairs \( (l, \ell) \), rather explicitly updating \( O(k^{\ell-1}) \) expert weights. These pairs can be visualized as vertices in a graph, where edges exist to create all valid paths. Each path through vertices (list-position pairs) represents a possible partition. After the vertex-weights are updated a WEIGHT PUSHING algorithm [10] needs be used to help normalize the weights.

7


4 Conclusion

We have discussed our approach to adaptive paging that combines disjoint lists of pages, trading off recency and frequency. Briefly, we covered ARC (Adaptive Replacement Cache), which is a heuristic for accomplishing this. ARC($p_i$), with fixed partition parameters $p_i$, can be used as experts in the caching framework of Gramacy, et. al. [6]. But, better yet, the partition parameter of ARC can be adjusted by expert weights internally, alleviating the need to simulate many (almost identical) copies of the same ARC algorithm. When used with rollover, the Expert Framework approach to ARC outperforms traditional ARC (which has also been adapted to use rollover). We conclude that expert-ARC is best tuned to the situation where refetching can essentially be done for free. Finally, we discussed some preliminary work in generalized ARC to work with three or more disjoint lists. Three lists is straightforward, but an arbitrary number of lists $\ell$ can be quite expensive. To solve this problem we alluded to a possible approach using some recent developments in using multiplicative updates in Path Kernels.

This research is still very preliminary. In the following two subsections we note some of the roadblocks we encountered thus far, and make some suggestions for future research.

4.1 Roadblocks

- Generalizing ARC (and rollover) to three lists took much more time than we had anticipated, and required an overhaul in the ARC code.
- Even after overhauling ARC to work with three lists, still more overhauling would be needed in order to generalize ARC to work with $\ell$ lists.
- The need for implementing a Weight Pushing algorithm is the main reason we did not have time to implement the Path Kernel approach.
- Data, Data, Data. We have never been able to duplicated Meggido & Modlin’s [9] results using ARC. They report 15-40% decreases in miss rate. We never came close to that on our datasets.

4.2 Future Work

- Full Path Kernel expert-based implementation of ARC for an arbitrary number $\ell$ lists.
- Theory bounding the misses of the master policy based on the Expert Framework in terms of the best expert ($p$ parameter) chosen off-line, and the best off-line partition of experts into segments.
References


