Online prefetching: a report after experiments

IDEAS NCBR

August 11, 2023

This report is an extended version of the previous report on online prefetching. We added `NEXT` strategy as prefetching and experiments section.

1 The online prefetching problem

We study the following *online prefetching* problem.

- The input is a sequence of page requests from the set of n possible pages.
- At time t, if the requested page is not inside the cache (of size k), then a page fault occurs and a cost of 1 is incurred.
- The goal is to minimize the total number of the faults.
- At time t, when the t-th request has already arrived but the $(t+1)$-th request hasn't arrived, the online algorithm can do p $(1 \leq p \leq k)$ swaps for free - here a swap means to evict one page from the cache and place another page inside the cache.
- Note that when $p = k$, then this problem becomes the pure prefetching problem [3].

1.1 Some intuition

What is a "good" prefetching algorithm for this prefetching problem under the $p = 1$ case?

- After dealing with each request at time t and before the next request arrives, a "good" prefetching algorithm behaves well in two aspects.
- On one hand, to check whether there exists a page j to arrive quickly after time t.
- On the other hand, if this page j exists and not inside the cache, then a page should be evicted from the cache to make room for this page j - the question is, is it really worthy to evict any page from the cache to make room for this page j, and if it is worthy, which page inside the cache should be evicted?

1.2 Remarks on differences between online prefetching and online paging problems

- A paging algorithm is not allowed to evict a page from the cache (and bring another page inside the cache) for free.
- Given the same sequence of requests, in the offline setting, even the cost produced by a paging algorithm is non-zero, the cost produced by an offline prefetching algorithm is always 0 - just before each request arrive at any time t, do the swap to place the page to be requested at time $t+1$ inside the cache.

1.3 A framework for theoretical evaluation

Since the offline optimal solution is directly 0, when evaluating the performance of an online prefetching algorithm `ALG`, it doesn't make any sense to compare the expected cost of `ALG` with the expected offline optimal cost 0 (even taking expectation, the offline optimal cost is still 0). As an alternative, the expected cost of `ALG` is compared with the expected cost of the optimal online algorithm $\texttt{OPT}_{\texttt{online}}$ (this $\texttt{OPT}_{\texttt{online}}$ knows all the distributional information used by the adversary to generate the sequence, but does not know the exact sequence of requests, and hence is weaker compared with the offline optimal solution `OPT`).

To be more precise, to evaluate the performance of a prefetching algorithm `ALG`, following some important previous work on stochastic paging [2, 4], we use:

$$\frac{\mathbb{E}[\texttt{ALG}]}{\mathbb{E}[\texttt{OPT}_{\texttt{online}}]}.$$

Here, the expectation is taken over all the sequences of length n with n as large as possible.

1.4 A framework for experimental evaluation

The goal of the experimental evaluation is to analyze different combinations of eviction and prefetching strategies, on both syntetical data drawn from different distributions (e.g., Markov chains) and on real-world data (e.g., the traces obtained from Huawei).

As a first step, we focus on the quality of the strategies, i.e., exclusively the cost measured in cache misses. We neglect the performance in terms of time/space resources used, as long as we are able to obtain the results in a reasonable amount of time.

Modeling the future. Many of the strategies that we test assume some prior estimations about the probability of certain events in the future. As a result, one can test some eviction/prefetching strategies combinations also with such different *probabilistic models*. We consider this subproblem the main point where using machine learning techniques may be successful. Of course, the probabilistic models themselves might require tuning some parameters, or applying data transformations (e.g., hashing) for specific types of data.

2 Experiments

The following reflects the current state of our implementation for experiments.

2.1 Eviction strategies

We implement and test the following strategies for evicting pages. We denote by parameter Φ the probabilistic model used.

- `LRU`: evict the least-recently used page.
- `MQ`: the multi-queue replacement queue from [6].
- `MET`(Φ): evict the page the has currently the *maximum expected time* till its next request (according to the model Φ).
- `LP`(Φ): evict the page that is *least probable* to appear in the next request, according to Φ.
- `DOM`(Φ): use the *dominating distribution* algorithm of [4]. See Section 4 for more details on this algorithm.

2.2 Prefetching strategies

The LP(Φ), MET(Φ) eviction strategies have their natural "symmetric" variants for prefetching. That is, prefetching with LP(Φ) amounts to prefetching the page that is *most likely* to appear next, whereas MET(Φ) prefetches the page that has the *minimum* expected next request time.

It is a bit more tricky to obtain a symmetric variant of the DOM(Φ) strategy. We nevertheless propose one way to do it in Section 4.1.

Each of the above prefetching strategies has also a parameter $p \geq 0$ limiting the maximum number of allowed pages to prefetch. Moreover, each of them may suggest prefetching less than p pages, e.g., if the strategy finds the current content of the cache reasonable.

However, in practice prefetching the next page in the address space seems to be the most efficient method. We denote that strategy simply NEXT.

We also tried to mix prefetching strategies, by averaging page probabilities output by two models.

2.3 Probabilistic models

The performance of LP(Φ), MET(Φ) and DOM(Φ) strategies depends heavily on the quality of the probabilistic model Φ that we use. Moreover, each of them requires a different type of estimations from Φ. For example, DOM(Φ) requires estimating "pairwise" probabilities that certain pages appear before the other. As a result, we might not be able to use every strategy combined with every model.

We test the following models.

- PROP: at each step, the pages are distributed proportionally to the number of their occurrences in the past.
- MARKOV(t): a generalization of PROP that also takes into account the context of size t (i.e., the last t requests). That is, if $t = 2$ and the last two requests were (a, b) then a page that was most frequently observed following the pages (a, b) is the most likely to appear next.
- LZ: the probabilistic model built upon a trie maintained by the Lempel-Ziv compression algorithm. As proved in [5], prefetching using LP(LZ) is optimal in the case of pure prefetching ($p = k$) and Markov models.
- FFM: the machine learning technique from [1]. See Section 5 for details.

The probabilistic models may also be combined with various data transformation techniques, like hashing, or encoding the differences between pages requested.

2.4 Data used

The experiments are done using the traces provided by Huawei.

3 Current theoretical results on prefetching

Unfortunately, we only found the theoretical results for either distributional paging problem [4], or pure prefetching [3, 5], but not for prefetching in our setting (with $p < k$). We review these results below.

Lund et al. [4] proved that if the probabilistic model Φ provides accurate predictions (as required by the respective eviction stategies), then:

- for $\texttt{DOM}(\Phi)$, $\frac{\mathbb{E}[\texttt{DOM}(\Phi)]}{\mathbb{E}[\texttt{OPT}_{\texttt{online}}]} = O(1)$, that is, DOM is almost optimal for caching against a distribution.
- for LRU, $\frac{\mathbb{E}[\texttt{LRU}]}{\mathbb{E}[\texttt{OPT}_{\texttt{online}}]} = \Omega(k)$;
- for MET, $\frac{\mathbb{E}[\texttt{MET}(\Phi)]}{\mathbb{E}[\texttt{OPT}_{\texttt{online}}]}$ is unbounded;

For pure prefetching (i.e., $p = k$), Vitter and Krishnan [5] proved that if one uses LZ probabilistic model, then $\texttt{LP}(\texttt{LZ})$ has $O(1)$ competitive ratio for Markov sources.

We have tried to extend the analysis of [4] and bound the competitiveness of using the $\texttt{DOM}(\Phi)$ strategy for both evictions and prefetching for $p = 1$. Unfortunately, our attempts have failed so far.

4 The DOM algorithm for the distributional paging problem

We consider the DOM algorithm for the distributional paging problem introduced in [4]. Upon a page fault at time t, the DOM algorithm proceeds as follows.

- Assume we have (estimated) probabilities $p(a, b)$ for any two different pages $a, b \in \mathcal{C}$ inside the cache – here $p(a, b)$ denotes the probability that page a arrives earlier than page b after time t. These (accurate) pairwise probability estimates are expected to be provided by the probabilistic model, which we denote by Φ.
- Given these k^2 probabilities as input, a so-called *dominating distribution* $\pi : \mathcal{C} \to (0, 1)$ is determined (by solving a linear program, with the coefficients being these $p(a, b)$). Lund et al. [4] proved that if a page j is sampled randomly according to the distribution π then, with probability at least $1/2$, this page j will be requested later than any other $k-1$ pages inside the cache.
- The sampled page j is evicted.

Lund et al. [4] prove that such an online paging algorithm is proved to achieve a performance ratio no more than 4 compared with the optimal online algorithm.

4.1 The possible DOM algorithm for prefetching

A symmetric version of $\texttt{DOM}(\Phi)$ could also be proposed for online prefetching in the case $p = 1$. By building a dominating distribution on the set of *all possible pages*, we could sample a page $j \notin \mathcal{C}$ such that with probability at least $1/2$, j will be requested earlier than any other page, and prefetch that page j.

We have not succeeded in proving that the symmetric DOM algorithm has good competitiveness ratio compared to the online OPT. We conjecture so based on the good properties of DOM for paging.

However, applying the symmetric DOM strategy for prefetching is even more challenging than for paging only. This is because we expect to have the cache size k much smaller than n, and making the decision at each step requires solving a linear program in n variables. As a result, in experiments we need to make further simplifications, for example limit our attention (when calculating the dominating distribution) to a small number of chosen pages from outside the cache, e.g., the top-scoring pages according to symmetric $\texttt{LP}(\Phi)$ or $\texttt{MET}(\Phi)$.

5 Field-aware Factorization Machine (FFM)

The FFM probabilistic model is based on the article [1]. It is used to model probability $p(a, b)$ that page a arrives earlier than page b assuming that a recent history is $s_{t-(h-1)}, \ldots, s_t$. To encode the input into FFM model, i.e. the history $s_{t-(h-1)}, \ldots, s_t$ and pages a and b we introduce $m = h + 2$ fields which we can think of as another dimension along the pages space. Fields 0 and 1 are fields of pages a and b respectively — the pages from the probability query $p(a, b)$. Field $i + 2$ is a field of page s_{t-i} in the history, for $i = 0, \ldots, h - 1$. The input is then encoded as a set of pairs (field, page number): $(0, a), (1, b), (2, s_{t-(h-1)}), \ldots, (h + 1, s_t)$.

FFM model formula for probability $p(a, b)$ is a logistic function:

$$p(a, b) = \frac{1}{1 + \exp(-\phi_{\mathrm{FFM}}(\boldsymbol{w}, \boldsymbol{x}))},$$

where

- $\boldsymbol{x}$ is a feature vector describing the state $(a, b, s_{t-(h-1)}, \ldots, s_t)$,
- $\boldsymbol{w}$ is a model we are trying to learn online,
- $\phi_{\mathrm{FFM}}(\boldsymbol{w}, \boldsymbol{x})$ is a function combining the feature vector with the model [1, Equation (4)].

The feature vector $\boldsymbol{x}$ is created by mapping each page into a number from the set $\mathbb{Z}_n = \{0, \ldots, n - 1\}$. This is done by hashing using modular arithmetic. Thus $\boldsymbol{x} \in \mathbb{Z}_n^m$. This hashing technique is common to reduce the space of the model. Model $\boldsymbol{w}$ is a tensor from $\mathbb{R}^{n \times m \times k}$, where k is a parameter regulating the size of the model. Then[1]

$$\phi_{\mathrm{FFM}}(\boldsymbol{w}, \boldsymbol{x}) = \sum_{0 \le j_1 < j_2 < m} \boldsymbol{w}_{x_{j_1}, j_2} \cdot \boldsymbol{w}_{x_{j_2}, j_1}.$$

We learn the model online by following the method described in [1, Section 3.1] with the following difference: the data point is sampled from the observed history. If the observed history is $s_0, \ldots, s_{T-1}$, we sample the moment $t \in [h, T - 2]$. Currently, the distribution used to sample moment t is such that after finishing entire run of an algorithm, the expected number of the total samples of each moment is equal. However, it is uncertain what distribution should be used. Having a moment t we can detect, for each page q of the cache, what is the next request time:

$$r(q) = \min\{t' \in [t, T) \,:\, s_{t'} = q\}.$$

We assume $r(q) = +\infty$ if the set on the right hand side is empty. Then, for each two distinct pages a and b from cache, we prepare a single sample $(y, (a, b, s_{t-(h-1)}, \ldots, s_t))$ where

$$y = \begin{cases} 1 & \text{if } r(a) < r(b) \\ 0 & \text{if } r(a) > r(b) \\ 1/2 & \text{if } r(a) = r(b) \end{cases}$$

The last case can happen if both $r(a)$ and $r(b)$ are infinities. For such prepared sample one single step of gradient descent is performed.

[1] If we represent $\boldsymbol{x}$ as a sequence of size m of indicating vectors of size n, i.e. $\boldsymbol{x} \in \{0, 1\}^{m \times n}$, then the formula would be exactly the same as Equation (4) in [1].

6 Experimental results

As data we used traces from Huawei.

6.1 Data preparation

We divided the address space into pages of size $S = 16$KB. If there is a request of some continues space on a volume, we check what pages are covering it. We treat such request as sequence of subsequent page requests.

Technically, assume the original request wants to read bytes from the interval $[Sa+r, Sb+q]$, for some a, b, r, q such that $Sa + r \leq Sb + q$ and $0 \leq r, q < S$. Then, we translate it into a sequence of requests of page numbers $a, a + 1, \ldots, b$.

We consider only operations with code `READ`. We take into account `volume_id` to distinguish requests from different volumes.

In order to create larger test cases we collected recursively operations from some given directory, merged them and sorted according to timestamp field.

6.2 Some results

We used cache size of 8, 16 or 64 pages. For prefetching we used $p = 1$, which allows only to prefetch a single page before the next page request, except when we mixed two models then we set $p = 2$. We tested all algorithms for all shared traces, but only some combinations are worth noting.

For caching we finally focused only on `LRU` and `DOM`. Two underlying probability models were leading: `FFM` and `MARKOV`, however `FFM` is much faster and sometimes much better, thus we fixed to it. We used history size of 14 page requests for `FFM`.

For prefetching `MARKOV` seems to be better than `FFM` so here we just included `MARKOV` model to predict the most probable page to occur next and prefetch it. However, the simplest strategy `NEXT` which prefetches the next page in an address space is much more efficient than any other method.

All methods are implemented in `Python`. This is a proof-of-concept implementation to see how the presented methods evaluate in practice.

Table 1 presents the results for cache size of 8 pages. We measured the total number of misses, so lower values are better. The selected traces are good representatives of all traces. Each other trace is similar to some trace from the table. `OPT` is an optimal offline algorithm only for caching.

We see that for prefetching `NEXT` strategy is the most efficient. Only for trace `3` prefetching by mixing models `MARKOV(3)` and `NEXT` we were able to reduce the number of misses. Thus mixing models does not seem to be a very profitable approach since `NEXT` is efficient enough. It is interesting to see `DOM(FFM)` can be competitive with `LRU` and `MQ`, especially, for very large test `11`. `MQ` caching strategy behaves similarly to `LRU`, possibly because of small cache, so we repeated experiment with cache sizes of 16 and 64.

Table 2 presents results for cache size of 16 pages. It is worth to note that `MQ` works very well for test `2`, possible because this large test has very low number unique pages comparing to the size of the test.

For cache size of 64 pages our proof-of-concept implementation of `DOM(FFM)` started to be slow. We had to lower the number of learning samples per step from 4 to 2, reduce history size from 14 to 6. As Table 3 shows it affected the quality of this algorithm. This time the combination of `LRU` and `NEXT` gave the best results.

Algorithm for caching	Algorithm for prefetching	Subdirectory 1	2	3	11	VDI_virus_scan
length		55709	120326	531873	15165	54750
unique pages		25262	2229	8779	2822	1436
OPT		50730	44895	179610	10214	22812
LRU		54971	72450	462974	15078	30468
MQ		54709	80082	458465	13871	35666
DOM(FFM)		54830	68472	274187	13470	29929
LRU	NEXT	32803	7202	52058	2726	14997
MQ	NEXT	32724	9166	48612	2705	25591
LRU	MARKOV(3)	44977	10532	59315	3773	17310
DOM(FFM)	NEXT	32926	7979	37684	2640	16197
DOM(FFM)	MARKOV(3)	44998	11366	44348	3713	17950
DOM(FFM)	MARKOV(3) + NEXT	34371	8387	34662	2655	16491

Table 1: Results of experiments for cache size of 8 pages. Length is the total number of request in the test, and unique pages says how many different request were seen.

Algorithm for caching	Algorithm for prefetching	Subdirectory 1	2	3	11	VDI_virus_scan
length		55709	120326	531873	15165	54750
unique pages		25262	2229	8779	2822	1436
OPT		48263	21739	66032	5484	16884
LRU		54939	48776	150748	14292	27895
MQ		54178	30133	172059	10814	26214
DOM(FFM)		53910	42044	127255	8984	25145
LRU	NEXT	32544	4143	21868	2683	13465
MQ	NEXT	32416	2209	24703	2646	13563
DOM(FFM)	NEXT	32721	4501	18228	2384	14822
DOM(FFM)	MARKOV(3) + NEXT	34144	4674	18089	2418	14822

Table 2: Results of experiments for cache size of 16 pages.

6.3 Conclusions

Prefetching the next page in the address space is the best choice. We see that simple caching strategy LRU can be replaced by the algorithm based on dominating distribution, which has much better theoretical properties than LRU. However, it depends on probabilistic model describing the nature of the data. In practice we used Field-aware Factorization Machine which is fast enough to be used in production environment which was proved by our proof-of-concept implementation. We suspect there is a room for improvement of using FFM by tuning parameters or by designing better approach for online learning.

References

[1] Y. Juan, Y. Zhuang, W.-S. Chin, and C.-J. Lin. Field-aware factorization machines for ctr prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, page 43–50, 2016.

Algorithm for caching	Algorithm for prefetching	Subdirectory 1	2	3	11	VDI_virus_scan
length		55709	120326	531873	15165	54750
unique pages		25262	2229	8779	2822	1436
OPT		43547	4134	26088	3190	8330
LRU		50211	5645	37183	4458	14772
MQ		50179	5646	47831	4458	14672
DOM(FFM)		51003	7715	44683	4861	16194
LRU	NEXT	31399	996	6643	2062	8575
MQ	NEXT	31383	996	8312	2062	8494
DOM(FFM)	NEXT	31940	1240	8254	2126	11043
DOM(FFM)	MARKOV(3) + NEXT	33356	1384	8758	2144	10808

Table 3: Results of experiments for cache size of 64 pages.

[2] A. R. Karlin, S. J. Phillips, and P. Raghavan. Markov paging. *SIAM J. Comput.*, 30(3):906–922, 2000.

[3] P. Krishnan and J. S. Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27(6):1617–1636, 1998.

[4] C. Lund, S. J. Phillips, and N. Reingold. Paging against a distribution and IP networking. *J. Comput. Syst. Sci.*, 58(1):222–232, 1999.

[5] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. *J. ACM*, 43(5):771–793, 1996.

[6] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue replacement algorithm for second level buffer caches. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.