ML prefetching: title to be decided

Abstract

ACM Reference Format:

. 2024. ML prefetching: title to be decided. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Increasing memory requirements for applications, e.g., machine learning, coupled with the slowdown of DRAM scaling [12, 13], makes DRAM one of the costliest components in data centers, constituting as much as 30% of the entire cost [20]. Thus prefetching is essential to reduce the number of page faults and improve application performance.

NEED TO ADD MORE CONTEXT HERE to explain why we need prefetching for page faults.

Prefetching is a very rich field with many different approaches. Traditional prefetching algorithms use heuristics to predict the next page to prefetch. These heuristics are based on the recent access patterns of the application and guess the next page to prefetch based on these patterns using a pre-defined algorithm [2, 9].

With the rececnt bloom of machine learning (ML) techniques, researchers have started to apply ML to the problem of prefetching.

WHAT ARE WE ACTUALLY PROPOSING?

In this paper, we exaplore the use of machine learning to predict page accesses and prefetch pages with high accuracy. In particular, compare LSTM, Transformer or Large Lanuage Models for prefetching.

2 Background and Related work

2.1 Heuristic Page prefetching

Page prefetchers are used in most modern operating systems to reduce the latency of page access from swap. Traditional algorithms prefetch based on sequential access to virtual addresses [5, 10] and are successful at fetching spatially related pages. One of the recent state of art prefetcher, Leap [1], improved traditional prefetching using majority trend detection to identify strided patterns; this makes Leap resilient to short-term irregularities in the memory access stream. Leap improved performance for many applications but cannot prefetch irregular accesses. Multiple researchs have tried to improve Leap's performance by addressing its shortcomings [18, 19, 22] in

- prefetching irregular patterns,
- isolating the swap subsystem and memory access histories of threads, and
- coordinating memory accesses from the garbage collector and the application.

All the above resulted in the non-perfect prefetching performance of the prefetcher in our initial experiments.

HOW TO CONTRAST OUR WORK WITH TRADITIONAL PREFETCHING?

Studies [15] has shown the need for more aggressive prefetching to reduce the number of page faults while scrificing some protability and memory bandwidth. Thus, we propose a machine learning based prefetcher that can improve the prefetching accuracy and reduce the number of page faults specifically made for target applications.

2.2 Machine Learning Prefetching

Past success of machine learning in cache eviction [17] and cache prefetching [7] has shown that machine learning can be used to predict page accesses patterns with high accuracy.

NEED MORE PREVIOUS WORKS

2.2.1 LSTM based prefetching.

LSTM, first introduced in 1997 [8], is a type of recurrent neural network that is capable of learning long-term dependencies. An LSTM is composed of a cell c, a hidden state h, an input gate i, an output gate o, and a forget gate f. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. The hidden state h is recurrent and is passed to the next time step. The process of an LSTM to process an input x_t at time step t is as follows:

1. Parrellelly compute the input gate i_t, forget gate f_t, and output gate o_t.

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i)$$
$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f)$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o)$$

   Here, W and b are the weights and biases of the gates, and σ is the sigmoid function.

2. Update the cell state c_t and hidden state h_t.

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$
$$h_t = o_t \odot \tanh(c_t)$$

   Here, $\odot$ is the element-wise multiplication operator.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

LSTM has shown promising results in cache prefetching [7] due to its ability to learn long-term dependencies. However, LSTM has a large number of parameters and is computationally expensive to train and run.

2.2.2 Transformer based prefetching. CHECK OUT THIS PAPER [6] TO BE ADDED

2.2.3 Large Language Models based prefetching. TO BE ADDED

3 Motivation

WHAT MOTIVATES OUR WORK?

Possible accessible virtual memory pages for a program is enonourmous. It is normal for program to access more than 1 GB of memory. Then that would be easy to get millions of possible pages the program will access. It is hard to predict which page the program will access next and to prefetch it before the program access it.

3.1 Scope and Limitations

WHAT IS THE SCOPE OF OUR WORK?

The scope of this work is to design a application-specific machine learning based page prefetcher that can predict the next page the program will access with high accuracy.

WHAT ARE THE LIMITATIONS OF OUR WORK?

3.2 Information to be used for prefetching

WHAT INFORMATION CAN BE USED FOR PREFETCHING?

There are lots of information that we can capture while a program is running. While collecting more information can help in predicting the next page the program will access, it can also increase the overhead of the prefetcher.

In this work, we will use the following information for prefetching:

1. The virtual address of the page that caused the page fault.
2. The program counter of the memory access instruction.

These information are easily reachable and meaningful for prefetching.

3.3 Prefeching as a classification problem

Past work [7] suggested that although page addresses are number, regression models are not suitable for prefetching. Instead, prefetching can be treated as a classification problem.

We treat the address space as a large, discrete vocabulary and perform classification.

DO WE TALK ABOUT OFFSET HERE?

At the same time, the number of possible pages that can be accessed is very large. This makes the encoding of the page addresses as one-hot vectors very large. Also fixed encoding of the page addresses can be problematic as address randomization can change the encoding of the page addresses.

Privous work [7] has suggested that offsets of page address accesses works better than absolute addresses, and since we are doing classification, we need to filter out the rare classes and only leave up to 50,000 most occured classes for offsets and pc values.

Definition 3.1. Offset = Next Page Address - Last Page address

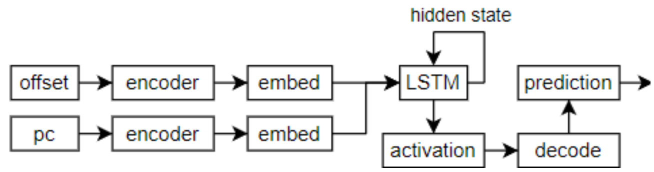

Figure 1. LSTM based prefetcher architecture.

4 Design and Implementation

In this section, we introduce the design and implementation of our LSTM, Transformer, and LLM based prefetchers.

4.1 LSTM based Prefetcher

We implement an LSTM based prefetcher based on past work [7]. It is shown at a high level in Fig. 1.

Each input offset, output offset and pc are encoded as a one hot representation of the total classes. Then the one hot encoding is embeded in a high dimensional space. The embeddings are then concatenated and fed to the LSTM The LSTM outputs the probability of the next page to prefetch. To get multiple pages to prefetch, we can apply softmax on the output and sample from the distribution.

- Embeding dimension: 128
- LSTM hidden dimension: 128
- Number of LSTM layers: 2

4.2 Transformer based Prefetcher

4.3 Large Language Model based Prefetcher

5 Evaluation

Correct prefetching choices can significantly reduce the number of page faults and improve the performance of applications. However, wrong prefetching choices can lead to unnecessary page faults due to memory pollusion and degrade performance. In this section, we evaluate the performance of the proposed ML prefetchers on a set of workloads and compare them with the state-of-the-art prefetcher, LEAP [1].

5.1 Metrics

We use memory access traces of benchmark workloads to evaluate the performance of the proposed ML prefetchers. The metrics we will focus on is:

Definition 5.1. Coverage = $\frac{\text{Page faults predicted by Prefetching}}{\text{Total Page faults}}$

Definition 5.2. Accuracy = $\frac{\text{Page faults predicted by prefetching}}{\text{Total predictions}}$

Coverage measures the percentage of page faults that were satisfied by the prefetcher. A higher coverage indicates that the prefetcher is able to predict more page faults. Accuracy measures the percentage of correct predictions made by the prefetcher. A higher accuracy indicates that the prefetcher is making correct predictions.

We want both coverage and accuracy to be high. However, there is a trade-off between the two. A prefetcher can achieve high coverage by prefetching aggressively, but this may lead to a decrease in accuracy. On the other hand, a prefetcher can achieve high accuracy by prefetching conservatively, but this may lead to a decrease in coverage.

We focus on the coverage metric as it is more important for prefetchers in our experiment since none of our prefetchers are aggressive. **TODO: HOW TO WORD THIS BETTER?**

5.2 Data Collection and processing

To collecte page faults, we use `fltrace` [21], which will interpose on all **heap allocations**. Thus, we can collect page faults for all heap accesses. Stack accesses are excluded from the analysis since stack page faults are rare and are not the focus of this work.

`fltrace` will need local RAM size to be set to run. It simulates the physical memory the program can access. Since each workload has a different memory requirement, we set the local RAM size to be 25% and 50% of the maximum memory requirement of the workload. This can be found by running `/usr/bin/time -v <workload>` and looking at the `Maximum resident set size`.

5.3 Workloads

We chose the following workloads for our evaluation:

- SPEC2017 [4]: mcf omnetpp and lbm with the default input files offered by the benchmark suite.
- GAP [3]: the default `bfs, pr, bc` workloads on the `twitter` dataset [11].
- WiredTiger [14]: the default `ycsb-a` and `ycsb-c` workloads with `icount=30000000` in the benchmark configuration file.
- DiLos-redis [16, 22]: the `redis-benchmark` workload with `lrange` on a list of queries made by DiLOS [22].

5.4 Analysis

TODO: Wait for all my results to come in.

References

[1] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 58, 15 pages.

[2] Mohammad Bakhshalipour, Mehran Shakerinava, Fatemeh Golshan, Ali Ansari, Pejman Lotfi-Karman, and Hamid Sarbazi-Azad. 2020. A Survey on Recent Hardware Data Prefetching Approaches with An Emphasis on Servers. arXiv:2009.00715 [cs.AR] https://arxiv.org/abs/2009.00715

[3] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[4] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[5] CheriBSD. 2023. *CheriBSD prefetcher.* https://github.com/CTSRD-CHERI/cheribsd/blob/565ae56372dec95ac74e3cc3f5130ada41a80b05/sys/vm/vm_fault.c#L862

[6] Q. Duong, A. Jain, and C. Lin. 2024. A New Formulation of Neural Data Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 1173–1187. https://doi.org/10.1109/ISCA59077.2024.00088

[7] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. arXiv:1803.02329 [cs.LG] https://arxiv.org/abs/1803.02329

[8] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[9] Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. *SIGARCH Comput. Archit. News* 25, 2 (May 1997), 252–263. https://doi.org/10.1145/384286.264207

[10] Linux kernel. 2017. *Linux Kernel VMA readahead prefetcher.* https://lwn.net/Articles/716296/

[11] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media? *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[12] Seok-Hee Lee. 2016. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*. 1.1.1–1.1.8. https://doi.org/10.1109/IEDM.2016.7838026

[13] Chris A. Mack. 2011. Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing* 24, 2 (2011), 202–207. https://doi.org/10.1109/TSM.2010.2096437

[14] MongoDB. 2024. *WiredTiger Workloads.* https://wiki.ubc.ca/Running_WiredTiger_workloads

[15] Athanasios E. Papathanasiou and Michael L. Scott. 2005. Aggressive prefetching: an idea whose time has come. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10* (Santa Fe, NM) *(HOTOS'05)*. USENIX Association, USA, 6.

[16] Redis. 2023. *Redis | Real-time Data Platform.* https://redis.com/

[17] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 529–544. https://www.usenix.org/conference/nsdi20/presentation/song

[18] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 35–53. https://www.usenix.org/conference/osdi22/presentation/wang

[19] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2022. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. https://doi.org/10.48550/ARXIV.2203.09615

[20] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[21] Anil Yelam, Stewart Grant, Enze Liu, Radhika Niranjan Mysore, Marcos K. Aguilera, Amy Ousterhout, and Alex C. Snoeren. 2023. Limited Access: The Truth Behind Far Memory. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless* (Koblenz, Germany) *(WORDS '23)*. Association for Computing Machinery, New York, NY, USA, 37–43. https://doi.org/10.1145/3605181.3626288

[22] Wonsup Yoon, Jinyoung Oh, Jisu Ok, Sue Moon, and Youngjin Kwon. 2021. DiLOS: Adding Performance to Paging-Based Memory Disaggregation. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 70–78. https://doi.org/10.1145/3476886.3477507