1 Introduction

Increasing memory requirements for applications, e.g., machine learning, coupled with the slowdown of DRAM scaling , makes DRAM one of the costliest components in data centers, constituting as much as 30% of the entire cost . 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 .

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 and are successful at fetching spatially related pages. One of the recent state of art prefetcher, Leap , 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 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 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.

2.2 Machine Learning Prefetching

Past success of machine learning in cache eviction and cache prefetching 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 , 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.

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

   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.

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

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

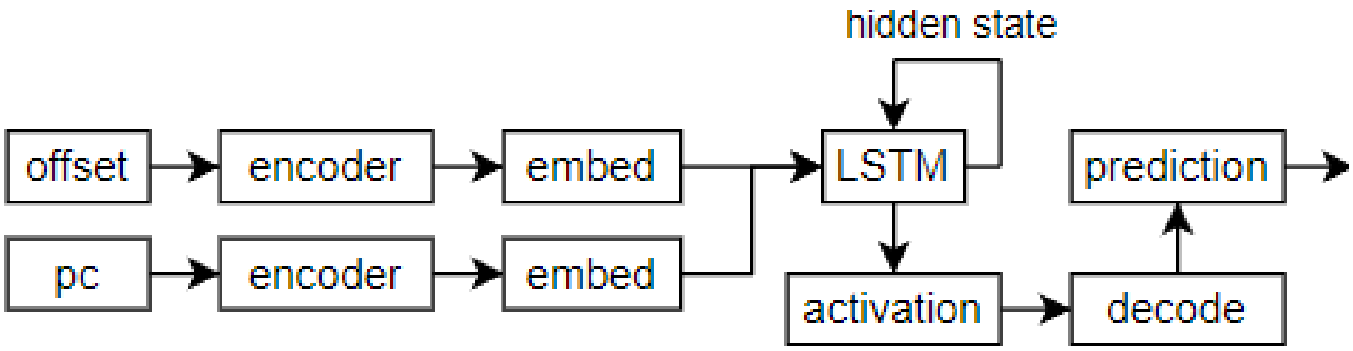

Figure 1. LSTM based prefetcher architecture.

LSTM has shown promising results in cache prefetching 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 TO BE ADDED

2.2.3 Large Language Models based prefetching. TO BE ADDED

3 Problem Statement

WHAT MOTIVATES OUR WORK?

Possible accessible virtual memory pages for a program is enonourmous. It is normal for program to access several 10th of GBs of memory. Then that would be easy to get a billion 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 Information to be used for prefetching

WHAT INFORMATION CAN BE USED FOR PREFETCHING?

3.2 Prefeching as a classification problem

Past work 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.

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 .

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

5 Evaluation

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 = $\frac{\text{Page faults predicted by prefetching}}{\text{Total predictions}}$

We use LEAP as the baseline prefetcher for our evaluations to compare with the proposed ML prefetchers.

5.1 Data Collection and processing

To collecte page faults, we use `fltrace` , 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.

Privous work 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 5.3. Offset = Next Page Address − Last Page address

`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`.

5.2 Workloads

We chose the following workloads for our evaluation:

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

5.3 Analysis

TODO: Wait for all my results to come in.

6 Challenges and Future work

Our focus in CHERI-picking was to demonstrate the feasibility of a generalized kernel pointer prefetcher. However, the prototype is not yet a complete implementation. While CHERI-picking improves coverage for BFS and canneal, it does not reduce the number of major faults for canneal; Canneal is a pointer-dense benchmark and a challenging one for CHERI-picking. Each faulted page contains many pointers; canneal first indexes into a pointer array randomly. If that access causes a page fault, CHERI-picking will prefetch 4 pages that will not be used. In fact, CHERI-picking prefetches 30 million pages, but only about 10% of those are ever accessed. BFS provides a nice contrast here; although it too is pointer dense, since we are traversing the entire data structure, every pointer prefetched is ultimately accessed. This results in 5% fewer major faults compared to the default prefetcher. Optimizing the CHERI-picking policy parameters such as `prefetch_count` is critical to reduce thrashing.

There are several avenues of performance optimization that we intend to investigate to address the overheads of CHERI-picking. First, we need to add better communication between the swap system, the default prefetcher, and CHERI-picking; by re-using swap data structures, we can optimize the CHERI-picking policy and record sufficient metadata to prevent CHERI-picking from running for cases when the default prefetcher is effective. Second, we do not run CHERI-picking on soft-faulting pages, but doing so asynchronously provides for more aggressive prefetching. Third, blindly prefetching a fixed number of pages on every faulted-in page is unlikely to be a good strategy; we need to explore how to better identify the right pointers to prefetch. Fourth, optimizing the I/O requests that CHERI-picking issues is essential; along with examining CHERI-picking performance on other swap backends such as RDMA. The overhead of scanning pages might be a bottleneck for faster swap backends such as CXL-attached memory or RDMA. To maintain performance in these scenarios, investigating techniques such as asynchronous execution of the CHERI-picking algorithm will be critical. Additionally, potential future hardware optimizations such as scanning the tags in hardware while copying a page can help remove the overhead.

7 Conclusion

CHERI-picking represents an initial step towards integrating application-specific prefetchers in the kernel. Our analysis reveals that both applications and benchmarks exhibit pointer-chasing patterns that the default kernel prefetcher fails to predict effectively. Through evaluation, we demonstrate benchmarks where the kernel prefetcher's effectiveness is limited, while CHERI-picking successfully predicts future accesses. CHERI-picking significantly enhances coverage for two standard benchmarks, improving it by up to a factor of three. However, much remains to be done to transform our prototype into a practical reality.

Acknowledgments

The authors would like to thank – Robert Watson for access to Morello hardware; George Neville-Neil, Jessica Clarke, Brooke Davis, John Baldwin, and Mark Johnston for their help with CheriBSD; Reto Achermann, Joel Nider, and the anonymous reviewers of PLOS for their feedback on the draft. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).