Lab 5: Prefetcher Design and Analysis (Bonus)

Instructor: Prof. Onur Mutlu

TAs: Juan Gómez Luna, Mohammad Sadrosadati, Mohammed Alser, Ataberk Olgun, Can Firtina, Rahul Bera, João Dinis Ferreira, Geraldo Francisco De Oliveira Junior, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Abdullah Giray Yaglikci, Rakesh Nadig, Haocong Luo, Joel Lindegger, Nisa Bostanci, Banu Cavlak, Aditya Manglik, Yahya Can Tugrul

Assigned: Saturday, December 3, 2022
Due: **Sunday, December 18, 2022** (Midnight)

1. Introduction

In this lab you will implement and evaluate multiple different prefetchers using ChampSim [1], a trace-based microarchitectural simulator written in C++. ChampSim models a modern high-performance out-of-order (OoO) core and enables evaluating ideas on classical problems of microarchitectures like data prefetching, branch prediction and cache replacement policies. Unlike the timing simulator you used in previous labs, ChampSim does not natively run an executable/assembly program but uses an instruction trace, extracted out from an application, to simulate the processor microarchitecture model.

We will provide you the program instruction traces from SPEC CPU 2006 and 2017 [2, 3] as well as ChampSim's source code with a baseline branch predictor and cache replacement policy. We will also provide a high-level prefetcher API. Your job is to use this API to implement different prefetching algorithms and evaluate the impact of your prefetcher implementations on the supplied program traces. An efficient prefetcher will reduce the number of cycles a program trace takes to execute.

2. Lab Resources

2.1. How To Get Started

Clone the ChampSim repository.

```
$ git clone https://gitlab.ethz.ch/rahbera/champsim.git
```

You need to run `build_champsim.sh`, which is a shell script to build the simulator. The script takes a single input parameter that is the prefetching algorithm name. We included three sample L2 prefetchers to help you get started. Compile ChampSim using the following command. This should create the ChampSim executable inside `bin/` directory.

```
$ ./build_champsim.sh next_line
```

You need program instruction traces to run ChampSim. You can download all the traces in your local machine by executing the following command (Note: You need ∼2 GB of free space to download the traces). This script takes `lab_traces.txt` as input to download the traces. Windows users can easily find a work around of the bash script.

```
$ cd scripts/
$ ./download_traces.sh
```

Your prefetchers will be evaluated using all 16 traces from `lab_traces.txt` only. However, there is an additional trace list `lab_traces_extended.txt` with 10 more traces. Feel free to test your prefetchers on these traces too.

To run ChampSim, execute `run_champsim.sh`.

```
$ ./run_champsim.sh <BINARY> <WARMUP_INST> <SIM_INST> <TRACE>
```

This script takes five arguments:

1. `BINARY`: The full path of the executable
2. `WARMUP_INST`: Number of instructions (in millions) to warmup (Default: 1)
3. `SIM_INST`: Number of instructions (in millions) to simulate (Default: 10)
4. `TRACE`: Full path of the trace file to execute.

Unless otherwise specified, you will always use 100 as `WARMUP_INST` (i.e., warming up for 100 M instructions) and 500 as `SIM_INST` (i.e., simulating 500 M instructions) for all simulations.

NOTE: each trace simulation might take around an hour to get completed. Hence, you might use the ETH `Euler` cluster [4] for launching your runs. Euler uses `bsub` to launch and manage jobs. Please go through the Euler website to learn more.

ChampSim metrics. ChampSim reports an array of metrics at the end of simulation. Some metrics to look for while evaluating the L2 prefetcher are: number of cycles, L1D average miss latency, number of L2 loads/hits/misses, number of L2 prefetch requests and useful prefetch requests.

2.2. Source Code

We will briefly walk you through the major directories of ChampSim and explain their functionality. Do **NOT** modify any of these files or directories unless explicitly mentioned.

- `inc`: This directory contains all the header files. These headers will automatically get included during the compilation. *Do not change any parameters in these files.* If you create any header file by yourself while implementing prefetchers, you need to put it here.
- `src`: This directory contains files that define the microarchitecture of an OoO core, uncore, cache and DRAM controller. You might skim through these files to get a better understanding of ChampSim.
- `branch` and `replacement`: These directories contain the source code of different branch predictors and last level cache (LLC, in our case it is L3) management policies. You will use the perceptron branch predictor [5] and SHiP [6] cache management policy for the LLC in this assignment.
- `prefetcher`: This is the main directory you will work with. ChampSim is extensible to implement prefetchers at all three cache levels, but you will only focus on implementing prefetchers at L2. The L2 prefetcher API is defined in `l2c_prefetcher.cc` file. The API consists of five key functions:
    1. `l2c_prefetcher_initialise`: This function is called when the cache gets created and should be used to initialize any internal data structures of the prefetcher.
    2. `l2c_prefetcher_final_stats`: This is the final function that is called at the end of simulation. This can be a good place to print overall statistics of the prefetcher.
    3. `l2c_prefetcher_operate`: This function is called for *each L2 lookup operation.* This means it is called for both L2 load and store accesses that can either hit or miss in the cache. The third and fourth arguments of this function helps in identifying the type of lookup.

In this lab, you will focus on finding patterns only in *L2 load accesses*. The first and second arguments provide the cacheline aligned address and the PC of the memory access, respectively.

4. `l2c_prefetcher_cache_fill`: This function is called for *each L2 fill operation*. The function argument names are self explanatory.
5. `prefetch_line`: You do *NOT* need to implement this function, but to use it when a prefetch request needs to be injected into the memory hierarchy. (Tip: see how how this function is used in `next_line.l2c_perf` or `ip_stride.l2c_perf`). The first two arguments are the PC and the cacheline address of the access that triggers this prefetch request. The third argument is the cacheline address of the prefetch request. By default, a prefetch request generated by the prefetcher at a cache level N first looks up the N_{th}-level cache. On a miss, the prefetch request looks up the next cache levels ($N+1, N+2$ etc) and eventually goes to main memory if it misses in the LLC. Once the memory supplies the corresponding data, the prefetched cachelines gets filled in all cache levels from the LLC to the N_{th} level. However, you can modify the prefetcher to send a hint to the memory subsystem to refrain from filling the cacheline in all cache levels if you think filling a prefetched line in all cache levels might hurt performance. This hint is passed by the fourth argument, `pr_fill_level`. For an L2 prefetcher, this argument can take either of the two values `FILL_L2` or `FILL_LLC`. For Task 1 and 2, you will set this to `FILL_L2`, i.e., the prefetched cacheline will get filled in both in L2 and LLC. You are free to use this parameter as you deem fit in Task 3.

2.3. Implementing a Prefetcher

To implement your own prefetcher, create a new file with `<prefetcher_name>.l2c_pref` naming format and define the four API functions in C++ in your own way. If you do not need to use any specific function, simply define the function with an empty body. *Please do not change the file `l2c_prefetcher.cc` by yourself.* The build script will automatically generate the appropriate `l2c_prefetcher.cc` file using your l2c_pref file and link it with the ChampSim model.

To help you get started, we have provided two simple prefetcher implementations: (1) next-line prefetcher and (2) table-based IP-stride prefetcher. These are taken from the 3rd Data Prefetching Championship [7] and can be found in `next_line.l2c_pref` and `ip_stride.l2c_pref`. These files should give you a brief understanding of how to implement a prefetcher. We have also provided an empty L2 prefetcher in `no.l2c_pref` to simulate a system without any prefetcher.

3. Basics of Prefetching

As we discussed in depth in lectures, prefetching is a speculation technique to predict a future memory request and fetch it into the caches before the processor core actually demands it. This prediction helps to *hide* the long latency of a DRAM-based main memory access and makes the program run faster.

The effectiveness of a prefetcher can be measured using four metrics:

- Performance: The number of cycles saved by the prefetcher (the higher the better)
- Coverage: The fraction of the memory accesses saved by the prefetcher (the higher the better)
- Accuracy: The fraction of prefetched addresses that are actually needed later by the program (the higher the better)
- Timeliness: The fraction of the latency of memory accesses hidden by the prefetcher (the higher the better)

In a processor core with a traditional three level cache hierarchy, a prefetcher can be employed at any level. For example, a prefetcher at the L2 cache level tries to capture the program access pattern by observing the accesses coming out from the L1 data cache and fills the prefetched cachelines up to the L2 cache level. Many works propose prefetching algorithms [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] to accurately predict future program accesses. Commercial processors use multiple prefetchers in their cache hierarchies to improve performance [19]. To learn more in depth about prefetching, please watch Professor Mutlu's lecture video on the topic [20].

4. Task 1/4: Implement GHB-based Stride Prefetcher

Your goal is to *implement* a Global History Buffer (GHB) based stride prefetcher at the L2 cache by using the prefetcher API. The prefetcher learns access patterns on cacheline addresses of loads that miss in the L1 data cache and prefetches predicted cachelines into the L2 cache. We provide the detail design of the prefetcher microarchitecture in the next subsection.

4.1. GHB Prefetcher

The GHB prefetcher [10] was introduced in 2004. It comprises of two major structures:

- Index Table (IT): A table that is indexed by program properties, like PC (program counter), and that stores a pointer to a GHB entry.
- Global History Buffer (GHB): A circular queue that stores time-ordered sequence of the observed cacheline addresses. Each GHB entry also stores a pointer (called `prev_ptr`) that points to the last cacheline address that has the same IT index. By traversing `prev_ptr`, one can get the temporal sequence of cacheline addresses that points to the same IT entry.

Stride Prefetching Algorithm. For each L2 cache access (both hit and miss), the algorithm uses the PC of the access to index into the IT and insert the cacheline address (say A) into the GHB. Using the PC and the link pointers in GHB entries, the algorithm retrieves the sequence of last 3 addresses by this PC that accessed L2 cache. The stride is computed by taking the difference between two consecutive addresses in the sequence. If two strides match (say d), the prefetcher simply issues prefetch requests to cachelines $A + ld, A + (l + 1)d, A + (l + 2)d, ..., A + (l + n)d$, where l is the *prefetch look-ahead* and n is the *prefetch degree*. For your design, please *statically* set both l and n to 4. Please also size the IT and GHB so that they are 256 entries each. For more detail on the GHB based stride prefetcher implementation, please read the original GHB paper [10]. Doing so will be important for you to implement Task 1 correctly.

5. Task 2/4: Implement Feedback Directed Prefetching

Your second task is to incorporate feedback information to evaluate the effectiveness of the prefetcher in the system and accordingly adjust the prefetcher's aggressiveness. Srinath et al. [11] propose a mechanism that incorporates dynamic feedback into the design of the prefetcher to increase the performance improvement provided by prefetching as well as to reduce the negative performance and bandwidth impact of prefetching. In Task 2, you need to extend the prefetcher you designed in Task 1 using a simplified version of Srinath et al.'s feedback directed mechanism [11]. The two dynamic feedback metrics you need to incorporate into the GHB prefetcher are described below:

- **Prefetch Accuracy**: Prefetch accuracy is a measure of how accurately the prefetcher can predict the memory addresses that will be accessed by the program.

$$Prefetch\ Accuracy = \frac{Number\ of\ Useful\ Prefetches}{Number\ of\ Prefetches\ Sent\ to\ Memory}$$

  where Number of Useful Prefetches is the number of prefetched cache blocks that are used by demand requests while they are resident in the L2 cache. Your goal is to implement the mechanism that calculates the prefetch accuracy as described in Section 3.1.1 of [11].

- **Prefetch Lateness**: Prefetch lateness is a measure of how timely the prefetch requests generated by the prefetcher are with respect to the demand accesses that need the prefetched data. A prefetch is defined to be late if the prefetched data has not yet returned from main memory by the time a load or store instruction requests the prefetched data.
  The prefetch lateness is defined as:

$$Prefetch\ Lateness = \frac{Number\ of\ Late\ Prefetches}{Number\ of\ Useful\ Prefetches}$$

  In general, prefetch lateness decreases as the prefetcher becomes more aggressive. Your goal is to implement the mechanism that calculates the prefetch lateness as described in Section 3.1.2 of [11].
- **DO NOT** implement the cache pollution metric proposed in [11]. You can do it, if you wish, as a part of Task 3.

The feedback information should be collected **every 1000 instructions**. Based on the collected feedback you need to update a counter that tunes the aggressiveness of the GHB prefetcher based on the information provided in Tables 1 and 2 below. The *prefetch distance* is defined in [11] in Section 2.1. The threshold values which define the accuracy and the lateness of the prefetcher can be found in Section 4.3 in [11]. **DO NOT** implement the changes in the cache insertion policy proposed in [11]. You can do it, if you wish, as a part of Task 3.

Table 1. Prefetcher aggressiveness configurations

Counter	Aggressiveness	Distance	Degree
1	Very Conservative	4	1
2	Conservative	8	1
3	Middle-of-the-Road	16	2
4	Aggressive	32	4
5	Very Aggressive	48	4

Table 2. Feedback counter updates based on prefetch accuracy and lateness.

Case	Prefetch Accuracy	Prefetch Lateness	Counter Update
1	High	Late	Increment
2	High	Not-Late	No Change
3	Medium	Late	Increase
4	Medium	Not-Late	No Change
5	Low	Late	Decrement
6	Low	Not-Late	No Change

6. Task 3/4: Design Your Own Prefetcher

Your goal is to come up with an L2 prefetcher implementation of your choice that beats the existing prefetchers. You might want to take a look at the prior prefetcher literature to select one for implementation, or bring your new ideas on prefetcher design. (Note: you can take inspiration from algorithms proposed in prior data prefetching championships. But **DO NOT** use the source code provided by them. We will specifically look for any abuse of this constraint and you will get penalized for plagiarism.)

Implement and evaluate your prefetcher in the similar way of Task 1 and 2. Please submit a brief write-up in PDF (no more than 2 A4 pages excluding references), clearly describing the prefetching algorithm including references to the relevant prefetching works.

NOTE: The designers of the *top three* highest-performing prefetching implementations will be rewarded 1.5% of their entire course grade, provided the prefetching technique is of their own. You can be inspired by past works, of course.

7. Task 4/4 (Bonus +25%): Comparison Against a State-Of-The-Art Prefetcher

Your goal is to quantitatively compare against Pythia [18], a state-of-the-art hardware prefetcher that makes use of Reinforcement Learning to perform (i) accurate prefetching decisions and (2) tune the aggressiveness of prefetching based on the system conditions (e.g. high/low bandwidth utilization). The source code is publicly available along with a walkthrough to reproduce the evaluation results (only for the 10 provided workload traces that you used in the previous tasks- do NOT download all the 150 traces) : `https://github.com/CMU-SAFARI/Pythia`. Once you obtain the evaluation results, you need extract insights regarding the performance achieved by your prefetcher compared to Pythia. We highly-recommend reading the paper to understand where the benefits of Pythia are coming from and how it compares against existing state-of-the-art-prefetchers like [15, 16, 17].

8. Submission and Evaluation

8.1. Submission

Use the corresponding assignment in Moodle (`https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=832909`) to submit your work. Please prepare your submission in the following way:

- Rename the prefetcher files from Task 1, 2 and 3 to `ghb.l2c_pref`, `fdp.l2c_pref` and `task3.l2c_pref` respectively. Put them inside `prefetcher` directory.
- Include a brief write-up in PDF format as mentioned in Task 3, describing your prefetching algorithm. Rename it as `report.pdf` and put it in the ChampSim home directory. Optionally, you may also include another short writeup describing any surprising observations that you think we should be aware of while grading your submission.
- Prepare a single tarball of your ChampSim repository excluding the traces directory and rename the tarball to `lab5_`*YourSurname*`_`*YourName*`.tar.gz`

8.2. Evaluation

Your prefetchers will be evaluated based on performance improvement over a baseline system without any prefetchers. We will individually compute the performance improvement for each of the 10 traces. The overall improvement will be calculated as the geometric mean of individual improvements. For the prefetcher implementation from Task 3, we will release a leaderboard, where you can check how your proposal performs as compared to other proposals from your peers.

9. Tips

- **If needed, please ask questions to the TAs via the online Q&A forum in Moodle (`https://moodle-app2.let.ethz.ch/mod/moodleoverflow/view.php?id=832910`).**
- When you encounter a technical problem, please first read the error messages. A search on the web can usually solve many debugging issues, and error messages.
- One easy way to debug your prefetcher is to test it using the `462.libquantum-714B.champsim-trace.xz` trace. `libquantum` a library for the simulation of a quantum computer and is part of SPEC CPU 2006 workload suite. It provides a structure for representing a quantum register and some elementary gates. Programmatically, it iterates over a 32 MB integer array and this

access pattern generates cacheline addresses with a constant stride of +1 over a physical page. You can use this workload to test your prefetcher.

References

[1] ChampSim. `https://github.com/ChampSim/ChampSim`.

[2] SPEC CPU 2006. `https://www.spec.org/cpu2006/`.

[3] SPEC CPU 2017. `https://www.spec.org/cpu2017/`.

[4] ETH Zurich Euler Cluster. `https://scicomp.ethz.ch/wiki/Getting_started_with_clusters`.

[5] D.A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA*, 2001.

[6] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *MICRO*, 2011.

[7] 3rd Data Prefetching Championship. `https://dpc3.compas.cs.stonybrook.edu`.

[8] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. In *IEEE TC*, 1995.

[9] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *MICRO*, 1992.

[10] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, 2004.

[11] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[12] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for data cache prefetch. In *ISC*, 2009.

[13] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *ISCA*, 2006.

[14] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *MICRO*, 2015.

[15] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *MICRO*, 2016.

[16] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo spatial data prefetcher. In *HPCA*, 2019.

[17] Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. DSPatch: Dual Spatial Pattern Prefetcher. In *MICRO*, 2019.

[18] Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO*, 2021.

[19] Disclosure of Hardware Prefetcher Control on Some Intel® Processors. `https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors`.

[20] Onur Mutlu, Computer Architecture Lecture: Prefetching, ETH Zurich, Fall 2020. `https://youtu.be/xZmDyj0g3Pw`.