Lab 4: Hardware Data Prefetcher Design and Analysis

Instructor: Prof. Onur Mutlu, Dr. Mohammad Sadrosadati

TAs: Dr. Konstantina Koliogeorgi, Dr. Zhiheng Yue, Rahul Bera, Konstantinos Kanellopoulos,
Nika Mansouri Ghiasi, Ataberk Olgun, Nisa Bostanci, Rakesh Nadig,
Ismail Emir Yuksel, Haocong Luo, Mayank Kabra, Andreas Kosmas Kakolyris,
Harshita Gupta, Konstantinos Marios Sgouras, Harsh Songara, Susana Rebolledo Ruiz

Corresponding TAs: Rahul Bera, Konstantinos Kanellopoulos

Assigned: Thursday, Nov 20, 2025
Due: **Wednesday, Dec 3, 2025** (Midnight)

1. Introduction

In this lab you will design and evaluate multiple hardware data prefetching techniques on top of a processor microarchitectural simulator, ChampSim. The ChampSim simulator is purely written in C++ and models a modern high-performance processor core with decent details: an aggressive out-of-order pipeline, with three levels of cache hierarchy and a DRAM-based main memory. The simulator takes an *instruction trace*, essentially a file containing instructions from a real-world program, and simulates the behaviour of the processor by executing every instruction. The goal of such a simulation methodology is to faithfully evaluate new ideas and their impact on a program's execution (e.g., how many cycles does the processor take, how many DRAM accesses does the processor make, etc.), without fabricating the design changes in the processor.

You will focus only one aspect of the processor's design in this lab: hardware data prefetcher. A hardware data prefetcher predicts addresses of future memory requests and fetches their data to the on-chip caches before the program executing in a processor demands them [1, 2, 3]. Our goal in this lab is to design multiple data prefetching mechanisms and understand (1) their impact on processor's performance, and (2) the effects of the overhead incurred by them.

To simplify the prefetcher implementation process, ChampSim provides a set of interface functions (see more here) that a prefetcher uses to interact with the rest of the processor simulator. Your task would be to (1) define each interface function as suggested by each task, (2) evaluate the resultant data prefetcher, and (3) prepare a comprehensive report summarizing all your observations and experiences.

> The ChampSim simulator you will use in this lab is also getting used in the **4th Data Prefetching Championship (DPC4)** that will be held in conjunction with one of the flagship conference in computer architecture, **HPCA**, on Feb 2026 in Sydney, Australia. The deadline for submitting to this championship is on **December 14, 23:59 AoE**. If you are interested, you can indeed submit the prefetcher you would design in task 4 to this championship and get a chance to present your work infront of experts in this field.

2. Lab Resources

2.1. How To Get Started

Note that, all the following instructions are tested in an Ubuntu-based machine. They should also work as-is in any Linux machine. If you are using Windows or MacOS, either try to work on a Linux machine, or figure out the equivalent commands in your OS. If you need help, please email us.

First, clone the ChampSim repository.

```
$ git clone https://github.com/CMU-SAFARI/ChampSim.git
```

To setup the build system, do

```
$ git submodule update --init
$ vcpkg/bootstrap-vcpkg.sh
$ vcpkg/vcpkg install
```

Now to build the simulator, you first need to specify a processor configuration and then build it:

```
$ ./config.sh dpc4/1C.fullBW.nopref.json
$ make -j
```

Here, `dpc4/1C.fullBW.nopref.json` specifies the configuration of our baseline processor that has only one core with three levels of cache hierarchy and without any data prefetcher. Please go through the JSON file and get a basic understanding of how the processor parameters are defined, especially the prefetchers (i.e., the prefetcher attribute defined inside L1D, L2, and LLC objects, all of which are currently set to "no", signifying no prefetcher). If the compilation finishes successfully, you should see an executable named `1C.fullBW.nopref` inside the bin directory.

You need program instruction traces to run ChampSim. Execute the following commands inside the ChampSim directory (Note: You would need Python3 and $\sim$14 GB of free space to download the traces).

Tips: Please be careful while copying the Google Drive links from the handout; the underscore character may not get copied correctly.

```
$ pip install gdown
$ mkdir traces
$ cd traces
$ gdown --folder https://drive.google.com/drive/u/1/folders/1pj9Tq-lz3TEbztHUdzUoT33wzk1Yqb_r
$ gdown --folder https://drive.google.com/drive/u/1/folders/1ynHx_dVL4s_ipBvJ8bgE_9Fs45Qj3AuQ
```

Note: Google sometimes limits the download bandwidth from Google Drive. If you face any error in downloading traces, please download the traces from this cloud storage bucket: `https://console.cloud.google.com/storage/browser/dpc4-public-traces`. Please only download the `Graph/GAP` and `gtraces_v2/charlie` directories.

The previous command should download two directories named `GAP` and `charlie`, containing 9 and 5 instructions traces, respectively. Traces inside GAP directory are captured from graph analytics benchmarks, whereas traces inside charlie are captured from real-world workloads running on Google's datacenters.

Now to simulate the processor using a given instruction trace, execute the following commands from ChampSim directory:

```
$ ./bin/1C.fullBW.nopref --warmup-instructions=10000000 --simulation-instructions=50000000
  <TRACE-PATH>
```

Unless otherwise specified, **you will always warmup the simulator for 10 million instructions and then simulate for the next 50 million instructions.**

Tips: Each simulation might take 10 minutes to an hour to get completed, depending on the instruction trace, the simulated processor configuration, and the capability of the machine you are running your simulations on. The simulator would print a progress log in every one million instructions. You can adjust the

logging frequency by passing a custom value to the simulator executable using the heartbeat knob (e.g., `--heartbeat=100000`). You may also want to use the ETH `Euler` cluster [4] for launching your runs, in case your personal computer is not running fast enough.

2.2. Important Simulation Metrics

ChampSim reports an array of metrics at the end of simulation. Some metrics to look for while evaluating prefetchers are: cumulative IPC, cpu0_L2C LOAD ACCESSES and MISSES, cpu0_L2C PREFETCH REQUESTED, ISSUED, USEFUL, and USELESS. For the course of this lab, we will mainly focus on cumulative IPC, which represents the performance of the processor: with a better prefetcher, you should observe a higher cumulative IPC.

2.3. How to Implement a Prefetcher?

Implementing a prefetcher needs two broad steps: (1) defining all prefetcher interface functions, and (2) configuring the processor to use the newly-implemented prefetcher. The following sections describe these two steps in detail.

2.3.1. Defining Prefetcher Interface Functions

To implement a new prefetcher named `my_prefetcher`, follow the following steps:

1. Create a new directory called `my_prefetcher` inside `prefetcher`.
2. Create two files called `my_prefetcher.h` and `my_prefetcher.cc` inside the `my_prefetcher` directory.
3. Define an empty prefetcher class named `my_prefetcher` inside `my_prefetcher.h` using the skeleton defined in `no/no.h`. Notice that the skeleton declares six interface functions, out of which only two are uncommented.
4. For a given task in this lab, define each interface function appropriately in `my_prefetcher.cc`.

Each interface function does the following task. You can get a detailed documentation about these functions here.

- **`prefetcher_initialize()`**: Gets called when the prefetcher gets initialized at the beginning of the simulation. Use this to initialize any dynamic structures you may define inside the prefetcher class.
- **`prefetcher_cache_operate()`**: This function is called when a cache lookup happens. This function provides a multitude of information related to the memory request that is currently looking up the cache, e.g., (1) the address of the memory request, (2) the address (also called instruction pointer or IP) of the load instruction that generated the request, (3) whether or not this memory request hits the cache, and many more. These information would come in handy implementing the pattern learning logic employed by a prefetcher.
- **`prefetcher_cache_fill()`**: This function is called when a cache fill happens. You can use this function to implement functionalities (if there is any) when a prefetched cacheline gets filled in the cache.
- **`prefetcher_cycle_operate()`**: This function is called at every clock cycle.
- **`prefetcher_final_stats()`**: This function is called at the end of the simulation. You can use this function to print any prefetcher statistics you may collect.

☞**Note:** Even though there are six prefetcher interface functions shown in `no/no.h`, you will never need to define the function `prefetcher_branch_operate()` for the scope of this lab. You can also keep the definition of some interface function blank, if you wish to do so. Irrespective of how you define the interface functions, *please do not change their prototype*. Doing so will raise compilation errors, unless the rest of the simulator is also changed.

✒**Tips:** To help you get started, we have provided implementations of many prefetchers. Please take a

look at two simple prefetcher implementations: (1) the next-line prefetcher and (2) the table-based IP-stride prefetcher. This should give you a comprehensive understanding of how to implement the prefetcher interface functions and how to inject a generated prefetch request to the cache hierarchy (note the `prefetch_line()` function).

2.3.2. Configuring the Processor to Use the New Prefetcher

Once you have finished implementing `my_prefetcher`, now you need to attach this prefetcher to the processor, more specifically to a cache level of the processor. To do this, follow these steps:

1. Create a new processor configuration file by copying `dpc4/1C.fullBW.nopref.json`.
2. Change the prefetcher attribute of the **L2C** object from "no" to "my_prefetcher".
3. Change the "executable_name" attribute value from "1C.fullBw.nopref" to the name of your choice (let's say "1C.fullBW.mypref").
4. Now rebuild the simulator using the commands mentioned before (i.e., execute `config.sh`, followed by `make`). This will create a new simulator executable inside the bin directory. This processor is now configured with the new prefetcher you just defined.

☞**Note:** You can technically attach your prefetcher to *any* cache level (e.g., L1D, L2C, or LLC) by simply changing the "prefetcher" attribute of the appropriate cache level as shown in point 2 above. For the scope of this lab, **you will attach the prefetchers you would design only to L2C**.

3. Tasks

This section defines the tasks of this lab. Please go through them carefully. Each task has a *minimum deliverable*, which you **must** include in your report to get points for that task. You are also encouraged to include any other observation that you find interesting in your report.

3.1. Warm Up: Running the No-prefetching Baseline

Your first task is to run the baseline processor configuration (i.e., `1C.fullBW.nopref.json`) with all 14 traces and report their cumulative IPC metrics.

Minimum deliverables. One bar plot, where the x-axis plots the workloads and the y-axis plots the IPC values. Mark the IPC value on top of each bar. Mention the workload names that has the highest and lowest IPC values.

3.2. Task 1/4: Implementing GHB-based Stride Prefetcher

Your second task is to implement a GHB-based stride prefetcher. Please go through the GHB prefetcher paper [5]. It comprises of two major structures: (1) an Index Table (IT) and (2) a Global History Buffer (GHB). You will be implementing the PC/CS scheme mentioned in Section 3.2 of the paper.

Prefetch 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 distance* and n is the *prefetch degree*. For your design, please statically set l and n to 4 and 6, respectively. Please also size the IT and GHB each with 256 entries.

Minimum deliverables. One bar plot, where the x-axis plots the workloads and y-axis plots the *speedup* in each workload. The speedup of a workload is defined as the ratio of IPC with the stride prefetcher and IPC of the baseline. A speedup value higher than 1 signifies that the prefetcher improved the processor performance, whereas a speedup value less than 1 signifies that the prefetcher degraded the performance. Add a separate

bar called "GEOMEAN" that plots the *geometric mean* of the speedup values of all 14 workloads. Mention the values of each bar on top of the bar. Also mention the workload names that has the highest and lowest speedup.

3.3. Task 2/4: Prefetching with Limited Main Memory Bandwidth

Your second task is to evaluate the GHB prefetcher you implemented in Task 1 in a processor with limited main memory bandwidth. To do this, you need to run two experiments for all 14 workloads.

First experiment: run the baseline processor with limited main memory bandwidth. To do this, reconfigure the simulator with `dpc4/1C.limitBW.nopref.json` configuration file. This configuration is identical to the `1C.fullBW.nopref.json` in every aspect, except that the previous configuration has $\frac{1}{6}\times$ main memory bandwidth than the later. Compile the simulator, run all traces, and note their IPC values.

Second experiment: run the GHB prefetcher you implemented in Task 1 with the limited main memory bandwidth. To do this, copy `dpc4/1C.limitBW.nopref.json` configuration and replace the "prefetcher" attribute value of L2C with the name of your GHB prefetcher. Reconfigure and recompile the simulator with the new configuration file, run all traces, and compute the IPC speedup in each trace. Speedup in this case would be the ratio of the IPC with the GHB prefetcher in limited bandwidth to the IPC of the baseline in limited bandwidth (i.e., the IPC you got in the previous experiment).

Minimum deliverables. One bar plot, where the x-axis plots the workloads and y-axis plots the *speedup* in each workload. Add a separate bar called "GEOMEAN" that plots the *geometric mean* of the speedup values of all 14 workloads. Mention the values of each bar on top of the bar. Also mention the following: (1) how does the geomean IPC improvement in this task compare to the same in task 1? (2) which workloads achieve the highest and lowest speedup in this task? (3) which workloads have the highest and lowest **change** in the speedup between task 1 and task 2? (4) is there any workload whose speedup got flipped in task 2 (e.g., it was higher than 1 in task 1 but became lower than 1 in task 2, or vice-versa)?

3.4. Task 3/4: System-Aware Prefetcher Design

In task 2, you would likely observe that the speedup of GHB prefetcher reduces with limited main memory bandwidth configuration. This is because the memory requests generated by the prefetcher (especially if the prefetch requests are inaccurate, meaning, these requests are not later demanded by the processor) consumes the scarce main memory bandwidth, which otherwise could have been used by memory requests demanded by the processor.

In this task, your goal is to extend the GHB prefetcher design with system awareness. The key idea is to enable the GHB prefetcher you implemented in task 1 to dynamically adjust the prefetch degree based on the prefetcher's accuracy (i.e., the fraction of the issued prefetch request that are demanded by the processor) and main memory bandwidth usage. To do this, you need to access three system-level information: (1) prefetch requests issued by the prefetcher, (2) prefetch requests that are later demanded by the processor, and (3) the processor's memory bandwidth usage. The first two information can be accessed by reading two statistics counters, `pf_issued` and `pf_useful`, maintained by the cache.[1] The bandwidth usage information can be accessed by calling the function `get_dram_bw()`, which returns the main memory bandwidth usage as a fraction of the peak main memory bandwidth.[2] Based on these information, implement the system awareness in the following way:

1. Keep a measure of the total number of issued prefetches and useful prefetches happened in an epoch of 1000 cycles.
2. At the end of an epoch (i.e., 1000 cycles): (1) compute the prefetcher accuracy as the fraction of useful prefetches to the issued prefetches happened in that epoch, and (2) take the measurement of the main

[1] Tips: you can can access the cache object to which the prefetcher is attached using the `intern_` pointer. Please check the ip_stride prefetcher's implementation.

[2] Please check `inc/dpc_api.h` to know more about this function.

memory bandwidth usage.

3. Adjust the prefetch degree for the next epoch based on the computed accuracy and memory bandwidth usage as per Table 1. **In no case, the prefetch degree should exceed the maximum degree of 6 or go below the minimum degree of 1**.

Table 1. Prefetch degree selection logic based on prefetch accuracy and main memory bandwidth usage (as a fraction of the peak bandwidth).

Memory Bandwidth Usage	Prefetch Accuracy	Prefetch Degree
$\geq 75\%$	$\geq 90\%$	No change
	$\geq 50\%$	Decrease by 1
	$< 50\%$	Decrease by 2
$25\% \leq BW < 75\%$	$\geq 90\%$	Increase by 1
	$\geq 50\%$	No change
	$< 50\%$	Decrease by 1
$< 25\%$	$\geq 90\%$	Increase by 2
	$\geq 50\%$	Increase by 1
	$< 50\%$	No change

Minimum deliverables. Compare your prefetcher's performance against the processor with both full bandwidth (i.e., the baseline experiment you ran in warmup task) and limited bandwidth (i.e., the performance you got in the first experiment of Task 2). Prepare two bar plots, one for full bandwidth, and another for limited bandwidth. The x-axis of each bar plot would be the workload names, y-axis would be the speedup in that workload in that bandwidth configuration. For example, the speedup of workload A in full bandwidth would be the IPC of A with your prefetcher in full bandwidth divided by the IPC of A without any prefetcher in full bandwidth. Similarly, the speedup of workload A in limited bandwidth would be the IPC of A with your prefetcher in limited bandwidth divided by the IPC of A without any prefetcher in limited bandwidth. Please also add a separate GEOMEAN bar in each bar plot. Does the system-awareness help improving prefetcher performance?

Food for thought: All thresholds we used in Table 1 are largely ad-hoc and fixed at design time. We can tune them by running many experiments and selecting the values that provides the highest performance on a set of workloads. Yet once they are tuned, we cannot change them. This rigidity significantly limits the prefetcher design; the same thresholds may not work for a wide range of workload or system configuration. Wouldn't it be better if prefetcher can tune these thresholds dynamically at runtime - by observing how a change in the threshold changes the performance? You can exploit meta-heuristic search methods (e.g., stochastic hill climbing [6], simulated annealing [7], random walks [8]) or even reinforcement learning [9] to achieve it. You may chose to design such prefetcher for the next task!

3.5. Task 4/4: Design Your Own Prefetcher

Your fourth and last task of this lab is to design a data prefetcher by your own. For this task, you can take inspiration from prior data prefetching mechanisms proposed in the literature [1, 10, 11, 12, 13, 14, 15, 16, 17, 3, 18]. You do not need to constrain your prefetcher design with any hardware overhead or complexity.

Minimum deliverables. (1) a brief description of your prefetching algorithm, and (2) extended two bar plots from previous task. Extend the first bar plot by showing the speedup of provided by your prefetcher in each workload and on average (i.e., the geometric mean) in over the baseline processor with full main memory bandwidth. Extend the second bar plot by showing the speedup fo your prefetcher over the baseline

processor with limited main memory bandwidth.

3.6. Bonus (+20%) Task: Comparison Against a State-Of-The-Art Prefetcher

The goal of the bonus track is to quantitatively compare all the prefetchers you designed till now against a state-of-the-art prefetcher, Pythia [18]. Pythia is a hardware data prefetcher that exploits reinforcement learning to adaptively generate prefetch requests. To ease the task, we have made Pythia's source code already available in the repository. For this task, you will get an additional 20% of the total points of this lab.

Minimum deliverables. Two bar plots: each of which will be extended from the previous tasks. The first bar plot will show the speedup of provided by Pythia in each workload and on average (i.e., the geometric mean) in over the baseline processor with full main memory bandwidth. Please add Pythia's bar in the bar plot you got in Task 3 for full-bandwidth processor. The second bar plot will show the speedup in the same way, but over the baseline processor with limited main memory bandwidth. In this case also, please add Pythia's bar in the bar plot you got in task 3 for limited-bandwidth processor.

4. Submission and Evaluation

4.1. Submission

Use the corresponding assignment in Moodle (`https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=1306626`) to submit your work. Please include at least the following items in your submission: (1) all your prefetcher codes, properly organized inside the `prefetcher/` directory, and (2) the report PDF. ***Please note that you may not get any point for the lab without the report***.

Please prepare your submission in the following way:

- Execute `make clean` in the ChampSim home directory, to remove of any compiler-generated files from the submission.
- Put your prefetcher code inside the `prefetcher/` directory.
- Put all your config files also inside the `prefetcher/` directory.
- Put the report in the ChampSim home directory and rename it as `report.pdf`.
- **Please make sure that you do not have any downloaded traces inside the ChampSim directory**. This will unnecessarily inflate the submission size.
- Prepare a single tarball of your ChampSim repository excluding the traces directory and rename the tarball to `lab4_YourSurname_YourName.tar.gz`

4.2. Evaluation

Your submission will be evaluated based on the quality of the report (any additional observations/insights beside the minimum deliverables are very encouraged), quality of the prefetcher code (both logical quality and readability), prefetcher you may design for task 4, and the optional bonus task. Special points will be awarded based on the novelty of the task 4 prefetcher.

5. 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=1306628`).**
- 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 any prefetcher you would design is to test it using a workload with regular pattern in it memory addresses. To do this, you can download a sample instruction trace from the following

link: `https://dpc3.compas.cs.stonybrook.edu/champsim-traces/speccpu/462.libquantum-714B.champsimtrace.xz`. This link would download a trace named `462.libquantum-714B.champsimtrace.xz`, which you can simulate through ChampSim in the same way as mentioned in Section 2.1. This trace is extracted from a C-based library called `libquantum` [19] which is used to simulate quantum computers. 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] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *MICRO*, 1992.

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

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

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

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

[6] Ari Juels and Martin Wattenberg. Stochastic Hillclimbing as a Baseline Method for Evaluating Genetic Algorithms. In *NIPS*, 1995.

[7] Peter J. M. van Laarhoven and Emile H. L. Aarts. *Simulated Annealing*. 1987.

[8] Samuel Karlin and James McGregor. Random Walks. *Illinois Journal of Mathematics*, 1959.

[9] Richard S Sutton, Andrew G Barto, et al. *Reinforcement Learning: An Introduction*. MIT press Cambridge, 1998.

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

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

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

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

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

[15] Pierre Michaud. Best-offset hardware prefetching. In *HPCA*, 2016.

[16] 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`.

[17] 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.

[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] libquantum - the C library for quantum computing and quantum simulation. http://www.libquantum.de.