Practical Assignment 2: Understanding Data Cache Prefetching

Computer Architecture

Due: Wednesday, November 8, 2023 at 5:00 PM

This practical assignment contributes 20% of the overall score for this course. It consists of a programming exercise culminating in a brief written report. Assessment of this assignment will be based on the correctness of the code and the clarity of the report, as explained below. The practical is to be solved individually. Please bear in mind the Academic Misconduct Policy guidelines. You must submit your solutions before the due date shown above. Follow the instructions provided in Section 6 for submission details.

In this assignment, you are required to explore data cache prefetching techniques using the Intel Pin simulation tool. You are strongly advised to commence working on the simulator as soon as possible.

1 Overview

In this assignment you will evaluate the hit ratio of a data cache when different hardware prefetch mechanisms are used. Towards this end, you will compare the following data prefetch mechanisms:

1. *Next-N-Line Prefetcher* – provided to you
2. *Stride Prefetcher* – for you to implement
3. *Distance Prefetcher* – for you to implement

2 Prefetching

Cache prefetching is a technique that reduces cache miss ratio by fetching data from memory to a cache, ideally before the data has been demanded from the processor.

2.1 Sequential Prefetcher

The simplest hardware prefetcher is a Next-N-Line Prefetcher, which brings one or several cache blocks adjacent to the one that was not found in a cache. If the next block(s) are already in a cache, they are not prefetched. The number of next blocks to prefetch (N) is called *aggressiveness*. The aggressiveness of the Next-N-Line Prefetcher can vary

across different implementations. The simplest Next-N-Line Prefetcher requests just the next block (N=1). However, a more aggressive prefetcher can prefetch more next blocks.

2.2 Stride Prefetcher

Stride prefetching is a more advanced hardware prefetching technique, which is particularly beneficial for array traversals. A *strided access* with stride M means that every Mth cache block is touched. Once a strided access pattern is detected, upon a load miss, the prefetcher fetches one or more blocks according to the pattern. Similarly to the Next-N-Line prefetcher, the number of blocks prefetched is called *aggressiveness*. If the block(s) are already in the cache, they are not prefetched.

The difference between addresses referenced by the same load instruction when it misses in the cache is called a *stride*. If a stride is the same for several consecutive misses of a load instruction, then the load access pattern is detected, and prefetching is enabled for this load instruction.

The information, required for detecting a strided access pattern is accumulated in a *Reference Prediction Table* (RPT). RPT entries are indexed by the address (i.e. Program Counter – PC) of a load instruction. An RPT entry contains:

- The PC of the load instruction, which acts as a tag.
- The previous address accessed by this load instruction on a cache miss.
- The stride for those entries which have established a pattern.
- A two-bit state.

The RPT structure is shown in Figure 1.

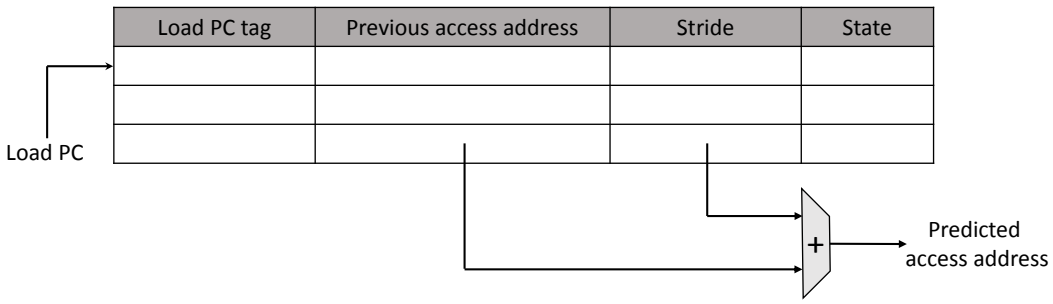

Figure 1: RPT structure.

The state diagram for an RPT entry is given in Figure 2. The four states, captured by the 2-bit state field, are:

- *Initial*: set on allocation of an entry in the RPT or after the entry experienced an incorrect prediction from steady state.
- *Transient*: corresponds to the case when the system is not sure whether it should prefetch or not.

- *Steady*: indicates that a pattern has been detected.
- *No prediction*: no pattern was detected for this entry.

An RPT entry is accessed when a load instruction experiences a cache miss. If an entry corresponding to a load instruction address is not found in the RPT, a new entry is allocated (after replacing an entry according to RPT replacement policy). The stride field of a new entry is initialized to zero. The previous address of a new entry is initialized to the memory access address. If there is a hit to RPT, the load address accessed by the instruction is predicted to be

$$access\_addr = prev\_access\_addr + stride\_value$$

The prediction is correct if the predicted address matches the access address. Note that the prediction is made only if an entry is found in the RPT. On a hit to the RPT, the state of an entry is updated depending on the prediction's correctness, according to the state transition diagram shown in Figure 2. In addition, the previous address field is updated with the memory access address. In all states except steady, in case of an incorrect prediction, a stride field is updated with the last stride. If the prediction is incorrect and the state is steady, the stride value remains unchanged. A prefetch occurs only if the prediction is correct and the corresponding RPT entry is in the state steady. In case of a prefetch, the previous address field is updated with the address of the last prefetched block.

For more information on the Stride Prefetcher, see the original Chen and Baer's paper[1].

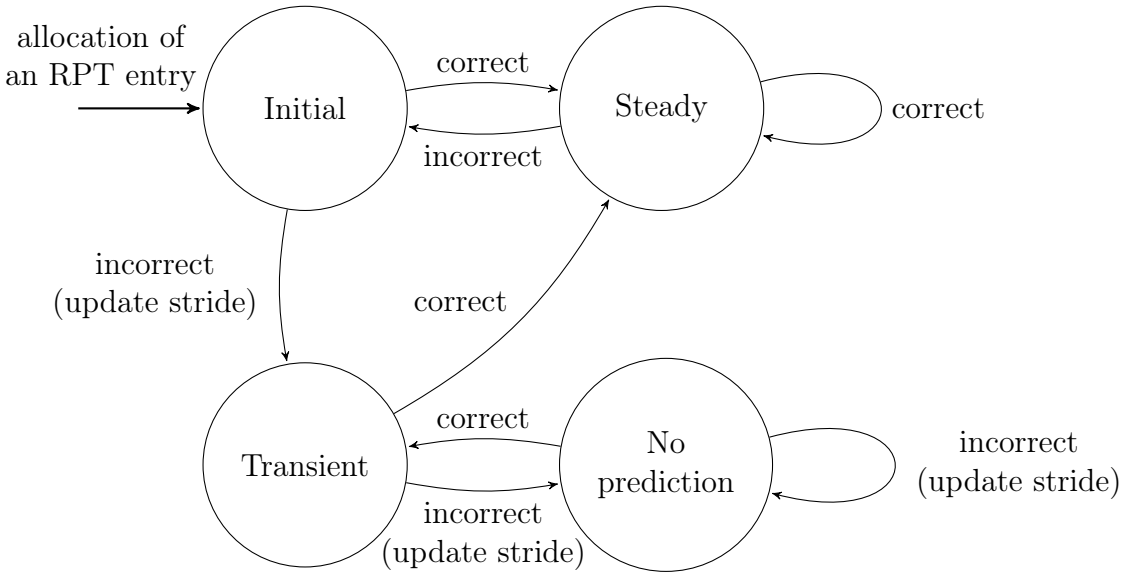

Figure 2: State transition diagram.

[1]Jean-Loup Baer and Tien-Fu Chen, "An effective on-chip preloading scheme to reduce data access penalty"

2.3 Distance Prefetcher

The *distance* between two memory accesses is defined as the numerical difference between two addresses. The Distance Prefetcher (DP) observes distances between pairs of consecutive misses, stores sequences of the observed distances, and makes predictions based on these sequences. For instance, after seeing a stream of global misses to addresses 10, 11, 13, 14, it observes that a distance of 1 (11 - 10) is followed by a distance of 2 (13 - 11), and that a distance of 2 is followed by a distance of 1 (14 - 13). Based on the first observation, the last miss (i.e. on address 14) will trigger a prefetch for address 16. Note that distances can take both positive and negative values.

As shown in Figure 3, the DP stores the following state:

- The previous miss address
- The previous distance
- An RPT with entries that consist of:
  - A tag of the entry, indicating the distance to which the entry refers to
  - A number of predicted distances that have been observed to follow the distance that the entry refers to. The number of the stored predicted distances determines the aggressiveness of the DP.

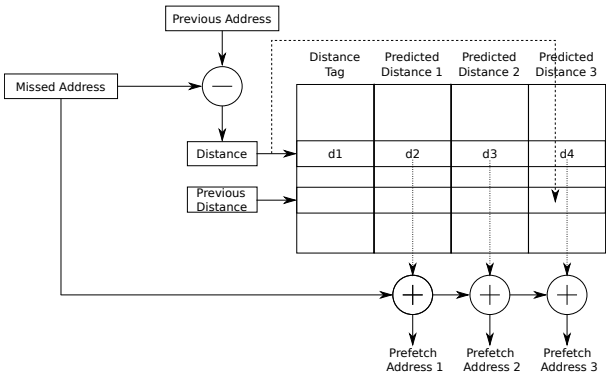

Figure 3: The Distance Prefetcher with aggressiveness equal to three

When the DP receives a new miss address, it compares it to the previous miss address to calculate the distance. Then, it searches the RPT for an entry that corresponds to that distance. If such an entry exists, then DP calculates the prefetch addresses by adding the predicted distances to the miss address and issues prefetches for all calculated addresses. Alternatively, if no entry corresponds to the distance, then an entry is allocated for the distance; if no unused entry exists, then an entry is evicted according to the *RPT replacement policy*.

The DP must be trained with the newly observed distance. As discussed, each entry has a number of slots that contain the predicted distances. In order to train the DP, the newly observed distance must be added as a predicted distance to the RPT entry that refers to the previous distance; if that entry has all of its predicted distances filled, then one of the predicted distances must be evicted to make way for the new distance.

The policy that determines which slot of the entry should be evicted is called the *entry replacement policy*. Finally, the previous distance and previous address state must be updated with the newly observed distance and address, respectively.

Let's examine the DP's functionality through an example. Assume an RPT with aggressiveness equal to 3, where there is an RPT entry with a distance tag equal to 2 and predicted distances $1, 3$ and 6. The previous address field stores the address 23 and the previous distance field stores distance 5. Then assume that the DP receives a miss to address 25. Firstly, it will calculate a distance of 2 and will search the RPT for an entry with tag equal to 2. When the entry is found, DP will compute the prefetch addresses by adding the predicted distances to the miss address, and thus, it will issue prefetches for addresses $26, 28$ and 31. Then, DP will perform the training: it will locate the entry with tag equal to 5, that refers to the previous distance and it will add the distance 2 to one of its predicted distances. If all the predicted address slots are in use, then one of them will be evicted, subject to the entry replacement policy. Finally, the previous address will be replaced with the address 25, and the previous distance will be replaced with 2.

3 Simulator infrastructure

Pin is a dynamic binary instrumentation engine that enables the creation of dynamic analysis tools (e.g. architectural simulators). Pin intercepts program execution and provides an API that allows you to execute high-level C++ code in response to runtime events like loads, stores, and branches. A Pin tool is a program which uses the API and specifies what has to happen in response. In this assignment, you are given an example Pin tool that intercepts data memory requests (loads and stores) and simulates a data cache with a Next-N-Line prefetcher. You will extend this tool to simulate a Stride Prefetcher and a Distance Prefetcher.

The directions below apply to CADE machines.

Download the zip file which contains *Intel pin*. The benchmarks can be found in a directory called "car_asn2_files/benchmarks". Inside the directory "car_asn2_files", you will find a README file that contains the steps needed to run the simulator with the benchmarks.
To access the guidelines within the directory, type the following commands in the terminal:

The steps can also be found in this Gist. [2]

1. Unzip the downloaded zip file using `unzip car2.zip` in your home directory.
2. `cd $HOME/car2/pin-3.28-98749-g6643ecee5-gcc-linux/source/tools/car_asn2_files`
3. `cat README`

The directory that contains the skeleton source code for a data cache (`dcache_for_prefetcher.hpp`) and Next-N-Line prefetcher (`prefetcher_example.cpp`) is further down that path, at:
`$HOME/car2/pin-3.28-98749-g6643ecee5-gcc-linux/source/tools/PrefetchExample/`

[2] https://gist.github.com/sharadbhat/20d2d13a11e704531b19a3308fa78091

How to compile the example code and use Pin?

1. Set all the required environment variables.
   `source $HOME/car2/pin-3.28-98749-g6643ecee5-gcc-linux/source\`

   `/tools/car_asn2_files/shrc-set_env_vars-for-students`

   Note 1: You may need to edit the file (shrc-set_env_vars-for-students) depending your directory structure, if you haven't unzipped the assignment in your home directory.

   Note 2: Copying the above command from this document may add a space between `.../source` and `/tools/....` Remove the backslash and combine the 2 lines. You should see a message being displayed when env vars are successfully set - "Environment has been set for using Pin."

2. Compile the example branch predictor source code, which will produce the branch predictor Pin tool. Note that a Pin tool is a dynamic shared library (file with .so filename extension).
   To generate it, run the following command within the following directory:
   `cd $HOME/car2/pin-3.28-98749-g6643ecee5-gcc-linux/source/tools/PrefetchExample`

   `make obj-intel64/prefetcher_example.so TARGET=intel64`

   The above command will create the pin tool `prefetcher_example.so` in the directory `$PF_EXAMPLE/obj-intel64.`

When the Pin tool runs `prefetcher_example.cpp`, it requires a benchmark program (along with its arguments) to be given as the last command line argument. The Pin tool runs the benchmark and while running it, passes all memory instructions to the simulated data cache. A Next N-Line Prefetcher is implemented inside `prefetcher_example.cpp` in a class called `NextNLinePrefetcher`.

Once Pin exits, it will generate a set of statistics for the simulated data cache and prefetcher in a file. The file's name will be given as an argument. The default name is `data`. The cache simulator, in the file `prefetcher_example.cpp`, allows for 3 different command line arguments:

1. *-pref_type*: Denotes which Prefetcher should be used; the choices are: none, next_n_lines, distance and stride. For example, in order to enable the Distance Prefetcher, use this command line argument `-pref_type distance`

2. *-aggr*: Sets aggressiveness. For example, when using the Next N-Line Prefetcher, in order to prefetch next line only, use this command line argument `-aggr 1`

3. *-o*: Sets the name of the output file to be generated. To generate an output file called MyOutput.out, give this command line argument `-o MyOutput.out`

Note that in the given example implementation, trying to use the Distance or Stride Prefetcher will have no impact on the simulation, as these Prefetchers are not implemented. The provided code contains comments on how to use the arguments that are necessary for your implementation (i.e. `-pref_type`, `-aggr`). These arguments will be used when testing the submitted code, so they must be used as described. Although it is not required for the assignment, there are also arguments available to change the dimensions of the data cache (`-b` for block size, `-a` for associativity and `-sets` for the number of sets).

Inside the car_asn2_files directory[3] there is a folder called `benchmark` that contains 3 micro-benchmark programs, `microBench1.exe, microBench2.exe, microBench3.exe`.

The guidelines inside the README text file contain more information about how exactly to run the prefetcher simulator and how to use the command line arguments.

IMPORTANT: The skeleton code `prefetcher_example.cpp` includes counters that you MUST increment in your code. The counters are: `prefetches, accesses` and `hits`. The comments inside the skeleton code describe the counters and the source code demonstrates when to increment them.

4 Prefetcher Parameters

Your task is to implement a Stride Prefetcher and a Distance Prefetcher. The skeleton code, found in `prefetcher_example.cpp`, includes comments and guidelines on how you can use the existing classes and their member functions. As an example, a Next-N-Line Prefetcher is already implemented. The parameters of the system are as follows:

- Stride Prefetcher:
  - RPT size: 64 entries
  - associativity: fully associative
  - RPT replacement policy: random
  - Aggression values: 1, 3 and 10
- Distance Prefetcher:
  - RPT size: 64 entries
  - entry size: equal to the aggressiveness (i.e. each RPT entry has as many predicted distances as specified by the command line argument "-aggr")
  - associativity: fully associative
  - RPT replacement policy: random
  - entry replacement policy: random
  - Aggression values: 1, 3 and 10

5 Marking Scheme

Correctness (80 marks). Includes code functionality and quality for the Stride and Distance Prefetchers.

Report (20 marks). You should write a report (max 1 page). In the report you should answer the following questions:

- What are the advantages and disadvantages of the Next-N-Line prefetcher?
- What are the advantages and disadvantages of the Stride prefetcher?
- What are the advantages and disadvantages of the Distance prefetcher?

[3] $PF_EXAMPLE/tools/pin-3.5-97503-gac534ca30-gcc-linux/source/tools/car_asn2_files/

- Why and how, varying aggressiveness impacts (or not) the hit ratio?
  (You may want to try different aggression values between 1-10 to notice a trend)
- Which prefetcher would you choose and why?

Note: No graphs are required for the report.

6 Submission

Submit on Gradescope (link available from Canvas). You will submit **one** zip file that consists of 3 files:

- Your source code at `$PF_EXAMPLE/prefetcher_example.cpp`
- The results you obtained at `$PF_EXAMPLE/results`
  (Containing the hit ratio for each variant mentioned in the results file)
- Your report in PDF format

Note: Do not upload any additional files. Make sure your submission has only the 3 files listed above. (No .rar or .7z files)

Your submission should clearly indicate (in both the report and the code) which prefetching techniques you have simulated completely and, in case you did not finish, which you have only partially completed. Make sure all code written by you is well-commented to make it easy for the markers to understand. Remember that your simulator will be compiled/executed on a CADE machine, so you must ensure that it works on CADE.

7 Similarity Checking and Academic Misconduct

We have zero tolerance for cheating. If your score in the assignment is significantly higher than the scores in the exam, only your score in the exam will count towards your grade. You must submit your own work. Detailed guidelines on what constitutes plagiarism and other issues of academic misconduct can be found at:

https://www.cs.utah.edu/undergraduate/current-students/
policy-statement-on-academic-misconduct/ Discussing high-level ideas behind the practical assignments is permitted and encouraged, but the code you turn in must be your own. You may not copy code from others or the internet, and you may not allow another students to copy your code. Use of tools such as AutoPilot and ChatGPT is not permitted. Any violation of the above is considered cheating and may result in a failing grade. All submitted code is checked for similarity with other submissions using software tools such as MOSS. These tools have been very effective in the past at finding similarities and are not fooled by name changes and reordering of code blocks.

8 Reporting Problems

Send an email to u1418754@umail.utah.edu **and** u1418984@utah.edu if you have any issues regarding the assignment.