Assignment 2: Understanding Data Cache Prefetching

Computer Architecture

Due: Monday, March 27, 2017 at 4:00 PM

This assignment represents the second practical component of the Computer Architecture module. This practical contributes 12.5% of the overall mark for the module. It consists of a programming exercise culminating in a 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 School of Informatics guidelines on Academic Misconduct. 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 have to implement the following data prefetch mechanisms:

1. *Stride Prefetcher*
2. *Stride Prefetcher with Prefetch Buffer*

2 Prefetching

Cache prefetching is a technique to reduce cache miss rate by fetching data from memory to a cache before the data is actually needed.

2.1 Sequential Prefetcher

The simplest hardware prefetcher is a Next-N-Line Prefetcher which brings one or several cache blocks next to the one which 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*. Aggressiveness of Next-N-Line Prefetcher can vary across different implementations. The simplest Next-N-Line Prefetcher requests just 1 next block (N=1). However, a more aggressive prefetcher can prefetch more next blocks.

2.2 Stride Prefetcher

Stride prefetching [1] 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 Next-N-Line prefetch, 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 prefetch is enabled for this load instruction.

Information required for detecting a strided access pattern is accumulated in a *reference prediction table* (RPT). RPT entries are indexed by the address (PC) of a load instruction. An RPT entry contains:

- a load PC tag,
- a previous address accessed by this load instruction on a cache miss,
- a stride for those entries which have established a pattern and
- a two-bit state

RPT structure is shown in Figure 1.

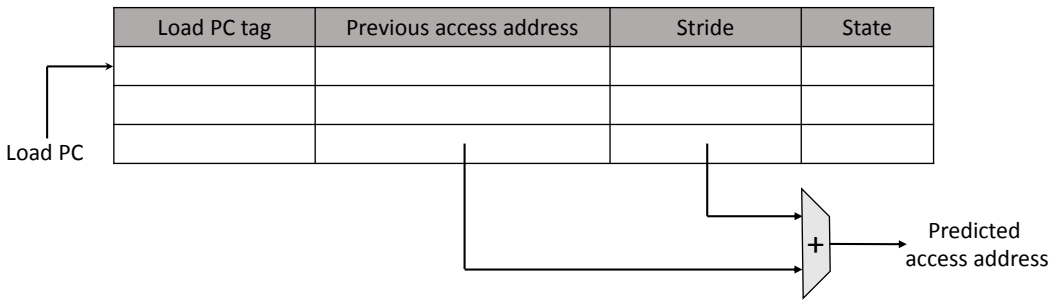

Figure 1: RPT structure.

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 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 the prediction should be stable for a while.
- *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 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, load address accessed by the instruction is predicted to be

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

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

For more information on Stride Prefetcher please look in the original Chen and Baer's paper [1].

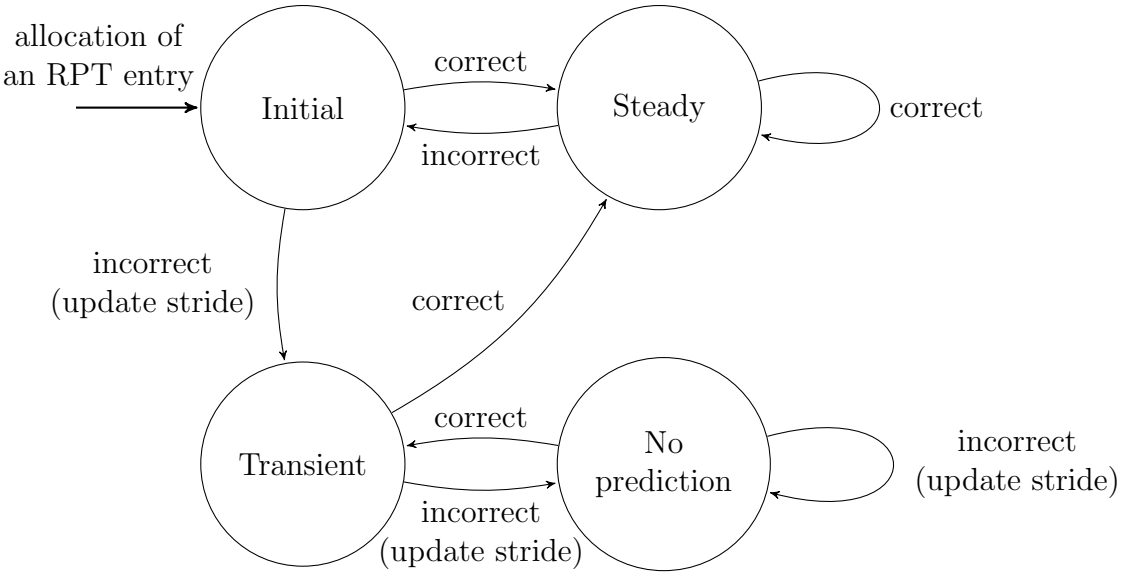

Figure 2: State transition diagram.

2.3 Prefetch Buffer

Aggressive prefetching can cause cache pollution. The prefetch buffer is a tiny cache whose purpose is to minimize such pollution by holding prefetched cache blocks that have

not been accessed. If a prefetch buffer is present, all prefetched blocks are allocated into it. On an access to the cache, the prefetch buffer is checked in parallel. In case a block is found in the prefetch buffer, it is evicted from the prefetch buffer and allocated to the data cache following the normal replacement policy.

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 Next-N-Line prefetcher. You will extend this tool to simulate a Stride Prefetcher and a Prefetch Buffer.

The directions below apply to DICE machines.

The Pin tool and the benchmarks are available in a directory that can be accessed through a DICE machine. Inside that directory there is a README text file that contains guidelines on how to run the simulator with the benchmarks.
To access the guidelines within the directory, type the following commands in the terminal:

1. `CAR_ASSGN2_PATH=/afs/inf.ed.ac.uk/group/teaching/car/assignment2`
2. `cd $CAR_ASSGN2_PATH`
3. `gedit README`

The directory that contains the skeleton source code for a data cache (`dcache_for_prefetcher.hpp`) and next-line prefetcher (`prefetcher_example.cpp`) is further down that path, at:

`$CAR_ASSGN2_PATH/tools/pin-3.0-76991-gcc-linux/source/tools/PrefetcherExample/`

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 arguments. The Pin tool runs the benchmark and while running it, passes all memory instructions to the simulated data cache. A Next Line Prefetcher is implemented inside `prefetcher_example.cpp` in a function called `prefetchNextLine()`.

Once Pin exits, it will generate a set of statistics for the simulated data cache and prefetcher in a file. The file's base name will be given as an argument, the simulator concatenates information to the name regarding the simulation properties (i.e. whether prefetching/ prefetch buffers was used, what was the aggressiveness). The default base name is `data`. The cache simulator, in the file `prefetcher_example.cpp`, allows for 4 different command line arguments:

1. *-pref*: Prefetching enabled/disabled. For example, in order to enable prefetching, use this command line argument `-pref 1`
2. *-prefBuff*: Prefetch buffer enabled/disabled. For example, in order to enable Prefetch buffer, use this command line argument `-prefBuff 1`

3. *-aggr*: Sets aggressiveness. For example, in order to prefetch next line only, use this command line argument `-aggr 1`

4. *-o*: Sets the base 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 the arguments regarding the Prefetch buffer(`-prefBuff`) and the aggressiveness(`-aggr`) will have no impact on the simulation. The given code contains comments on how to use the the three arguments that are necessary for your implementation (i.e. `-pref`, `-prefBuff`, `-aggr`).

In the path `$CAR_ASSGN2_PATH/benchmarks` there are 3 different benchmark programs. 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.

4 Your tasks

1. Implement Stride Prefetcher and Prefetch Buffer.

2. Evaluate and compare the hit ratio for each benchmark on the following configurations:

   (a) No prefetching

   (b) Next Line Prefetcher (example implementation)

   (c) Stride Prefetcher with varying aggressiveness (1-10 cache blocks) without the Prefetcher Buffer

   (d) Stride Prefetcher with varying aggressiveness (1-10 cache blocks) with the Prefetch Buffer

The file `prefetcher_example.cpp` includes comments and guidelines on how you can use the existing classes and their member functions. As an example, a Next-Line Prefetcher is already implemented.

You must run your experiments with all 3 benchmarks. You must vary aggressiveness of Stride Prefetcher from 1 to 10. The experiments must be run with the following parameters:

- Data cache (provided - no changes need to be made):
  - size: 512B
  - block size: 4 bytes
  - associativity: 2
  - replacement policy: LRU
  - prefetched blocks are allocated to LRU position
- Stride Prefetcher:
  - RPT size: 64 entries

  - associativity: fully associative
  - replacement policy: random

- Prefetch buffer:
  - size: 128B
  - block size: 4 bytes
  - associativity: fully associative
  - replacement policy: LRU

5 Marking Scheme

Correctness (40 marks). Includes code functionality and quality for the Stride Prefetcher and Prefetch Buffer.

Report (60 marks). You should write a report (max 5 pages). In the report you should compare the results (using bar/line charts) and answer the following questions:

- Why the different data prefetch mechanisms yield different (or similar) results?
- Why the same data prefetch mechanism yield different (or similar) results when run with different benchmarks?
- Why and how, varying aggressiveness impacts (or not) the hit ratio?
- Are the results what you expected and why (or why not)?

Also, write a discussion section on what you learned from this assignment. It is recommended to make one graph for each benchmark that shows the hit ratio for all 4 configurations when varying aggressiveness.

6 Format of your submission

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 DICE machine, so you must ensure that it works on DICE.

Submit an archive of your source code, along with the results text file, and a soft copy of your report as follows, using the DICE environment:

```
$ submit car 2 prefetcher.tar.gz report.pdf
```

7 Similarity Checking and Academic Misconduct

You must submit your own work. Any code that is not written by you must be clearly identified and explained through comments at the top of your files. Failure to do so is plagiarism. Note that you are required to take reasonable measures to protect your assessed work from unauthorised access. Detailed guidelines on what constitutes plagiarism and other issues of academic misconduct can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

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 Artemy.Margaritov@ed.ac.uk **and** Vasilis.Gavrielatos@ed.ac.uk if you have any issues regarding the assignment.

References

[1] Jean-Loup Baer and Tien-Fu Chen. "An effective on-chip preloading scheme to reduce data access penalty". In: *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*. IEEE. 1991, pp. 176–186.