Lab Assignment 4: Data Caches

1 Objective

The objective of this assignment is to investigate the impact of data prefetchers on cache performance. All work on this assignment is to be done in groups of two. This assignment involves a competition aspect; you are encouraged to start early!

This handout is organized as follows: Section 2 outlines the questions that students should answer in this assignment. Section 3 describes how to get the simulator, how to compile it, and how to run the benchmarks. Section 4 describes the prefetchers that students are required to implement. Section 5 provides hints to help with coding in the `sim-cache` simulator. Sections 6 to 7 describe the prelab and in-lab deliverables, and a high-level overview of the marking scheme.

2 Problem Statement

In this assignment, you will use the SimpleScalar cache simulator (`sim-cache`) to investigate the performance of three data cache prefetchers: (a) the **next-line** prefetcher, (b) the **stride** prefetcher, and (c) an **open-ended** prefetcher of your own design. Details about all three prefetchers are provided in Section 4.

In your report, you have to compare the performance of these prefetchers, when applied to a L1 data cache, with a baseline system with no prefetching. You are required to implement the three prefetchers inside `sim-cache`, run several benchmarks and answer the questions below:

Question 1: Provide a micro-benchmark and a configuration file that you use to verify that your implementation of the next line prefetcher is correct. Explain your choice.

Question 2: Provide a micro-benchmark and a configuration file that you use to verify that your implementation of the stride prefetcher is correct. Explain your choice.

Question 3: Using the configuration files provided with the simulator and statistics collected from the simulator, estimate the average memory access time for data accesses for benchmark `compress` for the configurations with no prefetcher, L1 data next line prefetcher and L1 stride prefetcher. In your calculations, assume the following hit times: $T_{access-L1Data} = 1$, $T_{access-L2} = 10$, $T_{hit-Memory} = 100$. Include a table with the following heading:

Config	L1 Miss Rate	L2 Miss Rate	Average access time

The table above will have three rows, each row for a different value of the column **Config** (i.e., baseline, next-line, stride). Fill in the rows with the values you collect with `sim-cache` and the configuration files provided with the simulator. For each configuration, compute the value in the last column using the formula for average memory access time.

Question 4: For benchmark `compress`, study the performance of the stride prefetcher when varying the number of entries in the RPT. Use the configuration files provided, changing **only** the number of entries in the RPT. Provide a graph that plots on the x axis the number of entries in the RPT and on the y axis a metric of your choice that measures the performance of the prefetcher. Explain your graph and your conclusions.

Question 5: If you were asked to include more statistics in the `sim-cache` simulator to study the performance of prefetchers in general, which statistics you would consider adding? (No implementation necessary, only an explanation is sufficient.)

Question 6: Provide a micro-benchmark and a configuration file that you use to demonstrate the performance of your open-ended prefetcher. Explain your choice.

3 Preparation

In this section, we briefly describe how to install and compile the `sim-cache` simulator, and how to run the benchmarks.

a) You can obtain the simulator source code and set it up in your ug account using the following Unix commands (assumming you have a working directory called ece552).This sequence of commands extracts the simulator files into your working directory. Do not leave the Unix permissions to your code open.

```
cd ~/ece552   # or some other working directory
cp /cad2/ece552f/simplesim-3.0d-assig4.tgz ./
tar -zxf simplesim-3.0d-assig4.tgz
cd simplesim-3.0d-assig4
```

b) You can build the `sim-cache` simulator using the provided Makefile by typing:

```
make sim-cache
```

You may safely ignore any warning messages you see during compilation.

c) You can run the benchmarks as follows (all two rows in one line):

```
./sim-cache -config cache-config/cache-lru-stride.cfg
    /cad2/ece552f/benchmarks/compress.eio

./sim-cache -config cache-config/cache-lru-stride.cfg
    /cad2/ece552f/benchmarks/gcc.eio

./sim-cache -config cache-config/cache-lru-stride.cfg
    /cad2/ece552f/benchmarks/go.eio
```

In the commands above, the simulator is run with a configuration file that specifies the parameters for the simulation. In the `cache-config` directory, you can find the following configuration files:

- `cache-lru-nextline.cfg` - configuration to be used for the runs with a next-line prefetcher at the L1 data cache level
- `cache-lru-stride.cfg` - configuration to be used for the runs with a stride-prefetcher at the L1 data cache level

- `cache-lru-open.cfg` - configuration to be used for the runs with the open-ended prefetcher that you design

These configuration files specify the geometry of the caches, the replacement policies and the prefetcher configuration. All these configuration files specify a two-level hierarchy with a separate L1 instruction and L1 data cache and a unified L2 cache. The size and the geometry of the caches are the same for all configuration files; the prefetcher configurations vary for the L1 data cache. Do not change the geometry or size of the caches when reporting results for the benchmarks. You are free to use other configurations for your microbenchmarks.

d) *CAUTION!* When you are redirecting the output of the simulator, use the -redir:sim flag of sim-cache. Do NOT redirect the output using the pipe character `|` or the redirect character `>` as this may cause variation in the simulated instruction count. To redirect the output of the simulated program, use the `-redir:prog` flag.

e) Please refer to the handout from Lab Assignment 1, if you need further instructions about how to create a microbenhmark.

3.1 Configuration Files

The `cache-config` directory contains the configuration files used for this assignment. The configuration files specify a cache hierarchy with a 16KB L1 instruction cache, 16KB L1 data cache and a 256KB unified (data and instructions) L2 cache. Note that the configuration files specify a maximum number of instructions to be simulated. It is important to use these configuration files when you report your statistics since different configurations produce different results.

The cache config parameter has the following format:

```
<name>:<nsets>:<bsize>:<assoc>:<repl>:<pref>

<name>   - name of the cache being defined
<nsets>  - number of sets in the cache
<bsize>  - block size of the cache
<assoc>  - associativity of the cache
<repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random
<pref>   - prefetcher type, 0 - no prefetcher, 1 - next line prefetcher, 2 -
           open-ended prefetcher, any other number num - stride prefetcher with
           num entries in the RPT
```

4 Data Cache Prefetchers

Hardware prefetchers predict memory accesses based on past history and bring the corresponding data/instructions into the caches before the processor demands it.

In this assignment, you will implement three different data prefetchers: 1) a next line prefetcher, 2) a stride prefetcher and 3) an open-ended prefetcher. The next sections explain how the first two prefetchers work.

4.1 Next Line Prefetcher

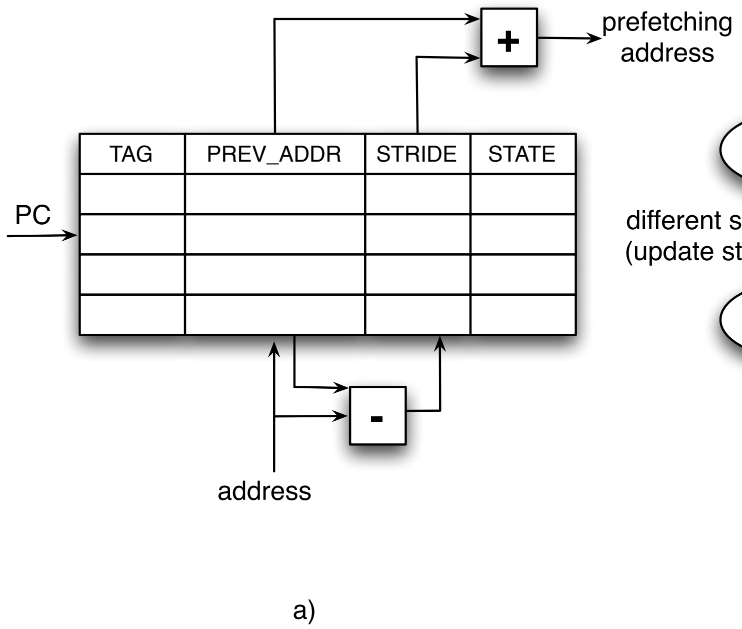
Next line prefetching (NLP) tries to take advantage of the spatial locality in applications. NLP is one of the simplest forms of prefetching: upon a memory access to address `ADDR`, a prefetch request to memory address `ADDR + cache_line_size` is generated if the cache line is not already present. This will bring into the cache the block subsequent in memory to the cache line currently being accessed, hence the name of the prefetcher. For this assignment, you are required to implement a next line prefetcher for the L1 data cache. Thus, any L1 data cache access is a candidate for generating a prefetch.

4.2 Stride Prefetcher

The stride prefetcher described in this section is a simplified version of the basic stride prefetcher described in "Effective Hardware-Based Data Prefetching for High-Performance Processors" by Chen and Baer [1]. Several commercial processors implement some version of this prefetcher (e.g., Intel Skylake [2], IBM Power 4 [4], and likely IBM Power 10 [3]). If you are interested in reading more, the paper is posted on Quercus.

Stride prefetchers try to predict future references based on the past history for the same memory instructions (i.e., loads/stores) that exhibit a constant stride in their memory addresses.

The core hardware structure employed by a stride prefetcher is a `reference prediction table (RPT)` that keeps track of data access patterns in the form of effective address and stride for memory operations (i.e., loads and stores).



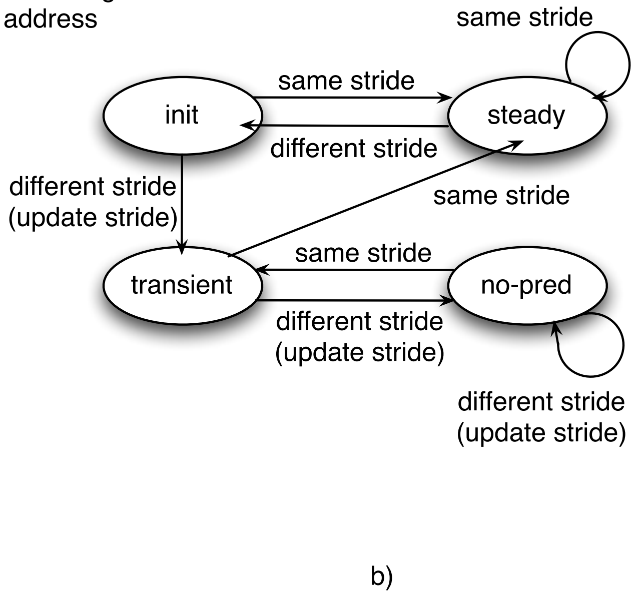

same stride
init
same stride
steady
different stride
different stride (update stride)
same stride
transient
same stride
no-pred
different stride (update stride)
different stride (update stride)
b)

Figure 1: (a) Reference prediction table (RPT). (b) RPT state transition

4.2.1 Reference Predictor Table (RPT)

Figure 1(a) shows the design of the RPT, which keeps track of previous addresses and associated strides for memory instructions. Prefetches are generated based on how stable the observed stride pattern is for a particular memory instruction. The RPT is organized as a direct-mapped cache structure. The RPT entry, indexed by the instruction address (i.e., PC), includes the address of the last data access, the stride information, and the regularity status for the instruction (i.e., an indication of how stable the stride pattern is). Each entry has the following format:

- `tag` – the tag corresponding to the address of the memory instruction (i.e., PC)
- `prev-addr` – the last address referenced by the corresponding instruction
- `stride` – the difference between the last two addresses that were generated by the corresponding PC (can be negative)
- `state` – a two-bit encoding (four states) of the past history

The four states and the transitions between them are depicted in Figure 1(b). They are:

1. `initial`: set at first entry in the RPT or after the entry experienced a different stride from the steady state one.
2. `transient`: corresponds to the case when the system is not sure whether the current stride is stable
3. `steady`: indicates that the prediction should be stable for a while.
4. `no-prediction`: disables the prefetching for this entry for the time being.

4.2.2 The RPT Mechanism

RPT records the address of the memory instruction, computes the stride for that access, and sets a state controlling the prefetching by comparing the previously recorded stride with the one just computed. The stride information is obtained by taking the difference between the addresses of the two most recent accesses for the same instruction.

When a load/store instruction is encountered for the first time, the instruction is entered in the RPT in the initial state. When the stride is obtained for the first time, i.e., at the second access, the state is set to transient since it is not known yet whether the pattern will be regular or irregular. When a further stride is computed and, if it is equal to the one previously recorded, i.e., if the same stride has occurred twice in a row, then the entry will be set to the steady state. It will remain in the steady state until a different stride is detected. At that point, the state is reset to initial.

If a different stride is computed while in the transient state, the entry is set to the no-prediction state since the pattern is not regular and erroneous prefetching should be prevented. In the presence of irregular patterns, the state of the entry will either stay in the no-prediction state or oscillate between the transient and no-prediction states until a regular stride is detected.

The RPT is updated for an instruction that accesses a memory location at address `addr` as follows:

Scenario 1. There is no corresponding entry in the RPT. The instruction is entered in the RPT, the `prev-addr` field is set to `addr`, the `stride` to 0, and the `state` to initial. This may involve replacing an existing entry (with a different tag) that maps to the same RPT entry.

Scenario 2. There is a corresponding entry. The new stride is computed as `addr - prev-addr`. Depending on the current state of the entry and whether the new stride is equal to the stride in the RPT the transitions depicted in Figure 1b) are performed. The following table further explains what happens in each transition. In the table, by stride condition "false" we mean the new stride is different than the stride recorded in the RPT, and by "true" we mean the same stride is observed. Once the transition to the new state is performed, the `prev-addr` is set to `addr`.

STATE	STRIDE CONDITION	NEW STATE	NEW STRIDE
initial	true	steady	no change
initial	false	transient	update
transient	true	steady	no change
transient	false	no-prediction	update
steady	true	steady	no change
steady	false	initial	no change
no-prediction	true	transient	no change
no-prediction	false	no-prediction	update

Following the update, a prefetch is generated depending on the state of the entry. If the entry is in one of the states `init`, `transient` or `steady`, a prefetch for the next address in the stride pattern is generated, provided the cache line/block is not already present in the cache (i.e., a prefetch for address `addr + stride`).

NOTE: At initialization, assume that all tags in the RPT are zero to indicate that RPT entries are empty. The executable to be simulated starts at higher addresses, so this assumption is safe. When indexing into the RPT, discard the lower bits in the PC that are always zero.

4.3 Open-Ended Prefetcher

In addition to the next-line and stride-prefetchers, you need to implement a prefetcher of your own design (the design can be taken from research or can be your own creation). Significant research exists in the area of data prefetchers and can be used to get you started. You can search for “data prefetcher” or “memory prefetcher” using `scholar.google.com` for relevant articles. In addition to considering prefetch schemes, you are also free to consider modifications to the replacement policy. There are no constraints on the hardware overhead used to implement your prefetcher. However, you should justify the feasibility of your design in your report. To receive marks on the open-ended prefetcher, your prefetcher must achieve an average L1 data cache miss rate less than 2.1% across the three provided benchmarks. In addition, up to two bonus points will be available for students with the top ranking prefetchers.

5 Coding Hints

In this assignment, you will use the `sim-cache` simulator. `sim-cache` is a functional simulator for the cache hierarchy and data/instruction TLBs. For the purpose of this assignment, we do not simulate the TLBs (the configuration files used declare the data and instruction TLBs as “none”), so no need to worry about it.

The files that you are most likely to work with are `sim-cache.c`, `cache.c`, `cache.h`. Please note that the code in the `cache.[ch]` files is also used in the timing simulator part of the SimpleScalar infrastructure. Since this assignment involves only the functional cache simulator, you can safely ignore any timing/latency parameters in these files.

`sim-cache` works similarly to the way `sim-safe` works in that it executes the program instruction by instruction. The main difference between the two is that `sim-cache` models the behavior of the cache hierarchy as the program executes. This is done by modeling cache accesses when instructions are fetched for execution and when executing memory instructions (e.g., load, stores). To achieve this, a cache access is simulated in the main loop of the simulator (in `sim_main`) starting at the L1 instruction cache. In addition, all macros used for accessing memory (such as `READ_BYTE` or `READ_WORD`) involve cache accesses as well.

The `sim_check_options` function parses the configuration parameters and creates the data structures corresponding to the specified memory hierarchy. To understand how caches work in the simulator, you may want to take a look at the data structures in the `cache.h` file and understand the code in the `cache_create` function. The same code is used to create the different levels in the cache hierarchy. Note that, as part of the initialization, a function pointer is passed as argument to the `cache_create` function. This function is used whenever an access to the lower cache level is needed (e.g., upon a cache miss or a write back).

Once you understand the data structures used for simulating the caches, you may want to read the code inside the `cache_access` function. This is the core function of the cache simulation. For implementing the data prefetchers, the skeleton of the code is provided. You need to fill in the two prefetch methods to actually generate the prefetch address and perform the prefetch operation. A prefetch can be simulated by a call to the `cache_access` function with the prefetch address and the corresponding cache structure.

You shouldn't need to modify the `sim-cache.c` file for this assignment. The simulator already has code to account for cache hits/misses, miss rates and prefetch statistics. Your code should be implemented only in the `cache.c` and `cache.h` files. You need to run the simulator with the benchmarks and configuration files provided and report on your findings.

6 Prelab

The prelab is worth 1/6th of the overall lab mark. Please complete the following steps before coming to the lab:

- Read all necessary background on data prefetching (textbook, lecture slides).
- Answer the following questions:
  1. Would a next line-prefetcher work well for an instruction cache? When would it issue useless prefetches?
  2. Can data prefetching be harmful for the performance of a system? Provide an example.
  3. How could you address the issue of harmful prefetches, assuming you cannot turn off your data prefetcher?
- Write most of your code, along with some microbenchmarks, before coming to the lab.
- Finally, be prepared to answer any high-level questions about the simulator, your code, and the lab material.

7 Lab Deliverables

At the end of this assignment you should submit the following files using the `submitece552f` command:

1. **cache.h** and **cache.c**: In `cache.c` and `cache.h`, identify all modifications with the comments

```
/* ECE552 Assignment 4 - BEGIN CODE*/

... your code in here...

/* ECE552 Assignment 4 - END CODE*/
```

2. **report.pdf**: a maximum 4-pages report (single-spaced with 12-point font size). Make sure your report can be viewed on the ug machines through `xpdf` or `acroread`. Non-readable reports will not be marked.
   - Provide succinct answers to all questions from Section 2. Make sure to include brief explanations for the mathematical derivations used to arrive at the answers.
   - Describe your open-ended date prefetcher implementation. Reason about how realistic your data prefetcher is in terms of area overhead and access time. Feel free to use CACTI from Lab assignment 2, to get real area and access time numbers.
   - Include a brief statement of work completed by each partner.
3. **mbq1.c, mbq2.c, and mbq6.c**: the microbenchmark files for Questions 1, 2 and 6. All microbenchmarks must include comments that explain how they verify your implementations. Failure to include appropriate explanation in the micro-benchmarks and in the report will result in a grade of 0 for the micro-benchmark portion of the assignment.
4. **q1.cfg, q2.cfg, and q6.cfg**: the configuration files for Questions 1, 2 and 6.

The submit command should be similar to the following:

```
submitece552f 4 cache.c cache.h report.pdf mbq1.c mbq2.c mbq6.c q1.cfg q2.cfg q6.cfg
```

You can view the files that you have submitted via the command:

```
submitece552f -l 4
```

Do not leave the Unix permissions open to your code, micro-benchmarks or configuration files. You can submit your code early to determine how your open-ended prefetcher ranks compared to your classmates.

8 Due Date and Marking Scheme

This assignment is due on **Friday November 24, 2023 at 6:00pm**. It is worth **6**% of the total course mark. The pre-lab will constitute 1/6th of the overall mark.

To receive marks on the open-ended prefetcher, your prefetcher must achieve an average L1 data cache miss rate less than 2.1% across the three provided benchmarks. Marks will be based on creativity, performance and the feasibility of your proposed implementation. Simple modifications to the size and algorithm of the stride prefetcher will not be viewed as creative. Up to 2 bonus points will be awarded to the top performing open-ended prefetchers. During the week prior to the due date, you can submit your code each day; each night the TA will run the open-ended prefetcher and post a ranking of the results on Piazza. You can continue to update and resubmit your code until the deadline. Please plan ahead and start early!

An automarker will be used to compile and run your implementation and verify the statistics generated. Therefore, your implementation is required to follow some strict rules. To begin with, use the simulator package specifically provided for this assignment as described in Section 3.

Things to watch for:

- run the benchmarks as specified in the handout with the configuration files provided
- initialize properly all the data structures as specified in the handout (e.g., fields in the RPT)

- pay attention to the generation of prefetches; a prefetch is generated if and only if the calculated prefetch address is not already in the local cache
- for the stride prefetcher, the stride can be negative
- do not change the existing stats; you can add more stats, if you are interested in studying something in particular, but do not modify the existing ones
- the only files to modify are `cache.c` and `cache.h`
- do not add additional files, they will not be taken into account; your simulator should compile with the provided `Makefile` by typing `make sim-cache`
- same automatic comparison of your submissions will be done to check for copied code; changing variable names and formatting will not defeat the checker.

9 Questions

Please post clarification questions on Piazza. Do not post solution code or microbenchmark code. This constitutes cheating and will not be tolerated.

References

[1] Jean-Loup Baer and Tien-Fu Chen. 1995. Effective Hardware-Based Data Prefetching for High-Performance Processors. IEEE Trans. Comput. 44, 5 (May 1995), 609-623.

[2] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. 2020. Reverse Engineering the Stream Prefetcher for Profit. In Proc. of Security of Software/Hardware Interfaces (SILM) Workshop.

[3] William Starke, Brian Thompto. 2021. IBM's POWER10 Processor. IEEE Micro 41, 2 (March 2021).

[4] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. 2002. POWER4 System Microarchitecture. IBM Journal of Research and Development. 46, 1, 5-25.