TDT4258 Low-Level Programming
HS 2022

Lab assignment 2

Building a Cache Simulator

Handout: Tuesday, 19th September 2023, 00:00
Deadline: Friday, 06th October 2023, 17:00

Teaching Assistants: TBA
Assignment Coordinator: Roman K. Brunner
Professor: Rakesh Kumar

Pre-amble

The labs are here for you to deepen your understanding of concepts taught in the lecture. The goal is that you not only develop a theoretical understanding of the matter, but also develop the technical skills to apply it in practice.

Each lab has a **main project**, but we also provide optional exercises for those who want to go beyond the mandatory exercise. To pass a lab, you only need to **hand in the solution to the main project**. The optional exercises are purely for your entertainment. Some of the optional tasks are easier than the main task; some are harder. We indicate the difficulty at the beginning of the problem description. The easier ones can serve as entry points if you feel that you are not yet ready to tackle the main task. But remember that in the end, all that counts is solving the main task, as the optional tasks do not count towards the pass/fail decision.

We assess the lab assignments on a **pass/fail** basis. To be allowed to sit in the exam, **you must pass all three labs**. As the assignments are part of the evaluation, they are subject to NTNU's plagiarism rules [1]. We have tools at our disposal and will run all submissions through plagiarism checkers. Copying code from current or past students is considered plagiarism. Hence, we advise you to not share code to prevent situations where we have to find out who copied from whom.

While copying each other is disallowed, we still encourage student discussions about your solutions. This will allow you to explore alternative approaches and solutions and learn about the advantages and challenges of particular implementations.

1 Description

In this lab, you will write a simulator similar to the one you used for the last exercise. As a full simulator such as CPUlator is quite extensive work, we will focus on the cache in this lab. So your task will be to write a parametrized cache simulator that reads a memory address trace and provides statistics about cache accesses and hits. You will write the simulator in C. Before you can execute your program, you need to compile it using GCC. To debug any potential misbehaviors, you can use GDB. To help you get started, we provide an outline in `cache_sim.c` on Blackboard.

We strongly advise you to ensure you fully understand how caches and the different cache architectures work. If you need to brush up on your caching

[1] https://i.ntnu.no/wiki/-/wiki/English/Cheating+on+exams

knowledge, you should revisit the topic using the lecture slides, videos, and the course textbook.

Even though we focus on a particular part of a whole system, it is still plenty of work, hence you are strongly encouraged to start work as soon as possible.

2 Main Task: Cache Simulator

Almost all current systems make use of some cache. A cache is a particular type of memory used to gap the difference in performance between the processor core and the main memory. A cache can store a limited amount of instructions and/or data close to the core, thus allowing fast access to the instruction/data present in the cache. A cache that holds both, instructions and data is called a **unified cache** (Similar to memory in the Von Neumann architecture). However, certain caches use an alternate design, called a **split cache**, using two caches: one stores purely data (**data cache**); the other solely instructions (**instruction cache**). This approach should remind you of the memory organization in the Harvard Architecture, which stores data and instructions separately.

We usually deploy software simulators in research and engineering instead of monitoring actual hardware to understand caching behavior. In this exercise, you will investigate a program's memory access and caching behavior. To obtain the results, you have to write a flexible cache simulator that meets the following specifications:

Fixed parameters The following parameters are constant and stay the same for all simulations:

- We are analyzing a program from a 32-bit system. So you are guaranteed that all addresses will always be 32 bits long.
- Data in the cache is organized in blocks of 64 bytes (The size of the cache line is 64 bytes)
- The caches use a FIFO replacement policy for associative caches

Variable parameters The following parameters are passed as command-line arguments to the cache simulator:

- The overall cache size in bytes or kilobytes (Note that the only acceptable values are powers of 2, between 128 and 4096 bytes)

- Cache mapping: The cache can either be a direct mapped or fully associative cache
- Cache organization: The cache can either be **unified** or **split** (see above or lecture slides for more details)

If the cache is configured to use the **split cache** organization, the cache size that has been provided is split up equally between the data and the instruction cache. However, both will use the same mapping (direct/fully associative mapping).

Sample Commandline Parameters

1. Parameters: cache size = 2KB, cache mapping = direct mapped, cache organization = unified cache
   These parameters result in the following single cache configuration (as data and instructions are stored in the same cache):
   - Cache Mapping: Direct mapped
   - Block Size: 64B
   - Cache Size: 2KB
   - Number of Blocks: 32 ($2KB/64B$)
   - Number of bits for block offset: 6 (Index into 64B)
   - Number of bits for the index: 5 (Index into 32 cache line)
   - Number of bits for the tag: 21 ($= 32 - 6 - 5$)
2. Parameters: cache size = 2KB, cache mapping = direct mapped, cache organization = split cache
   These parameters result in the following two separate caches for instructions and data. The trace indicates if an access should go to the data or instruction cache:
   - Parameter: Instruction Cache Value / Data Cache Value
   - Cache Mapping: Direct Mapped / Direct mapped
   - Block Size: 64B / 64B
   - Cache Size: 1KB / 1KB ($= 2KB/2$)
   - Number of Blocks: 16 / 16 ($= 1KB/64B$)
   - Number of bits for block offset: 6 / 6 (Index into 64B)
   - Number of bits for index: 4 / 4 (Index into 16 cache line)
   - Number of bits for tag: 22 / 22 ($= 32 - 6 - 4$)

Expected Results from the Simulator

Your simulator needs to capture a variety of statistics about the caching behavior of the application under test. The required statistics are

1. Number of accesses to the cache: Just count the total accesses that your cache has handled
2. Number of hits: Count the number of addresses that were already present when the program accesses that specific location
3. Hit rate: Relative expression of hits over the total handled instructions

You are free to track additional metrics if it helps debug and test your cache implementation. We are though only examining the results for the three mandatory statistics above.

Summary

In this lab assignment, you are expected to write a cache simulator, which reads a trace file containing memory references and reports three different statistics (accesses, hits, and hit rate). The cache configuration is determined by parameters that are passed as command-line arguments. The command line parameters are the cache size, cache mapping (DM/FA), and cache organization (UC/SC).

Hint: You can write your own trace files with easily predictable hit rates (e.g. a four accesses trace using four different addresses and has a 50% hit rate) to verify your simulator before using the traces provided to you. Your simulator will be tested with trace files different from the one that is already provided to you.

2.1 Trace and Code Files

On Blackboard you find a memory trace file `mem_trace.txt` and an outline for the cache simulator `cache_sim.c`. Testcases together with the expected output can be found on Blackboard as well.

The memory trace file consists of a sequence of memory accesses and serves as input to your cache simulator. For each memory access, the trace file provides whether it is an instruction or data access and the corresponding 32-bit memory address. See the following short excerpt from a trace file:

```
I 8cda3fa8
I 8158bf94
```

D 8cd94c50
I 8cd94d64
D 8cd94c54

The letter at the beginning of the line indicates the type of access, where "I" stands for instruction access, and "D" stands for data access. In the case of a unified cache, both types of accesses go to the same cache, whereas in the split cache, memory accesses marked "I" need to go to the instruction cache and accesses marked "D" to the data cache.

The `cache_sim.c` provides code for:

1. Reading the command-line parameters and initializing the corresponding values
2. Reading the trace file
3. Declaring the cache array and statistics
4. Printing the cache statistics

2.2 Compiling and Running the Simulator

To run your program, you first need to compile it. The most common compiler for C code is GCC. If you are running a Linux machine, you likely already have GCC installed[2] and can execute it in your terminal using

```
gcc -o cache_sim cache_sim.c
```

to generate an executable `cache_sim`. To run the simulator, you need to pass command-line parameters as follows: `./cache_sim 1024 dm uc`.

This would execute your cache simulator with a direct mapped, unified cache with a size of 1024B.

If you don't have a Linux machine at your disposal, here are a few pointers that should help you in setting up your development environment:

- Windows users:
  - MinGW: A native Windows port of the GNU Compiler Collection (GCC): https://sourceforge.net/projects/mingw/

[2]If not: https://www.ubuntupit.com/how-to-install-and-use-gcc-compiler-on-linux-system/

  - Visual Studio Code integration for Compilation and Debugging: https://code.visualstudio.com/docs/cpp/config-mingw
  - Visual Studio Code & Windows Subsystem for Linux (WSL) Setup: https://code.visualstudio.com/docs/cpp/config-wsl
  - Set up a Linux VM, e.g. using VirtualBox: https://www.virtualbox.org/wiki/Downloa
  - Using Docker: https://docs.docker.com/desktop/install/windows-install/ (Docker setup), https://hub.docker.com/_/gcc (GCC Docker Image)

- Mac users:

  - Using Homebrew: Install homebrew (https://brew.sh/) and then GCC (https://formulae.brew.sh/formula/gcc#default)
  - Using Docker: https://docs.docker.com/desktop/install/mac-install/ (Docker setup), https://hub.docker.com/_/gcc (GCC Docker Image)
  - Install XCode and use the shipped clang compiler (compile using `cc` instead of `gcc`). Least preferred version, as we cannot help you if there are any particular differences between the `gcc` and the `clang` compiler.

3 Optional Tasks

- **[EASY]:** Why can a cache respond faster to the reads and writes from the processor than the main memory? How many reasons do you find? Why don't we just use cache if it is faster?

- **[EASY]:** Why do we want to simulate the cache in software instead of running it on actual hardware and just observe how the hardware behaves?

- **[MEDIUM]:** In the main task, we asked you to support direct mapped and fully associative cache. Can you add an implementation for a set associative cache?

- **[HARD]:** Extend your simulator to support physical and virtual addresses. What do you need to change? We suggest first writing down the precise requirements.

4 Submission

Submit your **commented C code file** `cache_sim.c` before the deadline on Blackboard. Aside from comments, make sure your code is well readable, variables are sensibly named and that your code is well structured.

We expect all submissions to meet the following requirements:

1. The submission is on time. We provide enough time to solve the labs, but we expect you to start early.
2. You submit *a single C file* `cache_sim.c`.
3. The submitted file compiles and does not crash when running on the provided test traces.
4. The command-line interface of the program has not been changed.

Failing to meet those requirements can result in failing the lab.

5 Assessment

This assignment will be evaluated on a pass/fail basis. Your program will be judged on correctness and completeness, so please make sure that all the requirements are respected and functional.

Commenting on your code and keeping it tidy is very important. Helping us understand what you did, supports us in assessing your work – we can only give points for what we understand.

6 Similarity Checking and Plagiarism

You must submit your **own work**. You must write your **own code** and not copy it from anywhere else, including your classmates, the internet, and automated tools. Failure to do so is plagiarism. Detailed guidelines on what constitutes plagiarism can be found at:

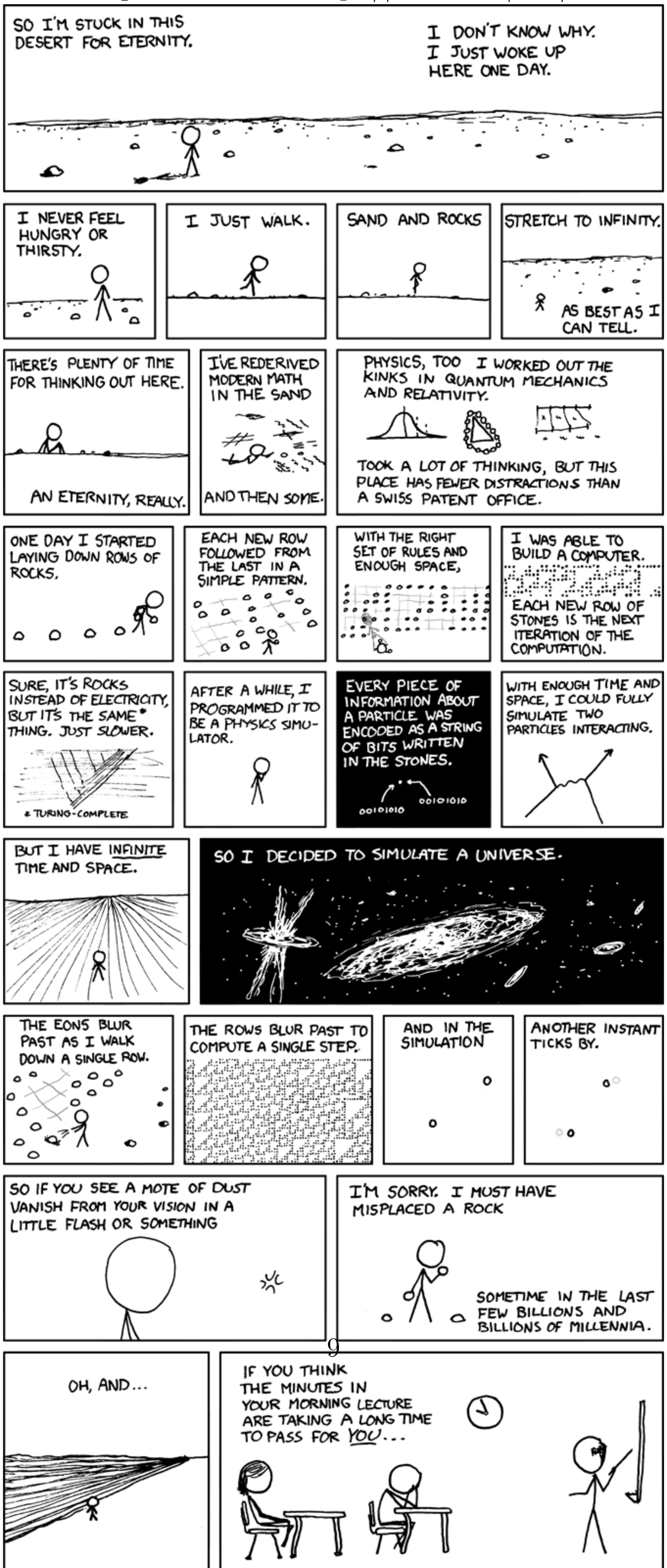
https://innsida.ntnu.no/wiki/-/wiki/English/Cheating+on+exams

We check all submitted code for similarity with other submissions. Plagiarism detection tools have been effective in the past at finding similarities. They have gotten excellent over time, so it is inadvisable to try and outsmart them. So don't do it, not only because we will most likely catch you, but because it is morally wrong and can undermine your academic integrity, even a long time into the future. For more references, see https://www.google.com/search?q=resigns+over+plagiarism+allegations (statistics on the 8th of August: 467'000 results).

7 Questions

If you have any questions about this assignment, we encourage you to ask the question on the course forum on Piazza. By that, you also help other students who have the same questions in the future.

Figure 1: Source: https://xkcd.com/505/


SO I'M STUCK IN THIS DESERT FOR ETERNITY.
I DON'T KNOW WHY. I JUST WOKE UP HERE ONE DAY.
I NEVER FEEL HUNGRY OR THIRSTY.
I JUST WALK.
SAND AND ROCKS
STRETCH TO INFINITY.
AS BEST AS I CAN TELL.
THERE'S PLENTY OF TIME FOR THINKING OUT HERE.
AN ETERNITY, REALLY.
I'VE REDERIVED MODERN MATH IN THE SAND
AND THEN SOME.
PHYSICS, TOO I WORKED OUT THE KINKS IN QUANTUM MECHANICS AND RELATIVITY.
TOOK A LOT OF THINKING, BUT THIS PLACE HAS FEWER DISTRACTIONS THAN A SWISS PATENT OFFICE.
ONE DAY I STARTED LAYING DOWN ROWS OF ROCKS.
EACH NEW ROW FOLLOWED FROM THE LAST IN A SIMPLE PATTERN.
WITH THE RIGHT SET OF RULES AND ENOUGH SPACE,
I WAS ABLE TO BUILD A COMPUTER.
EACH NEW ROW OF STONES IS THE NEXT ITERATION OF THE COMPUTATION.
SURE, IT'S ROCKS INSTEAD OF ELECTRICITY, BUT IT'S THE SAME* THING. JUST SLOWER.
* TURING-COMPLETE
AFTER A WHILE, I PROGRAMMED IT TO BE A PHYSICS SIMULATOR.
EVERY PIECE OF INFORMATION ABOUT A PARTICLE WAS ENCODED AS A STRING OF BITS WRITTEN IN THE STONES.
00101010
00101010
WITH ENOUGH TIME AND SPACE, I COULD FULLY SIMULATE TWO PARTICLES INTERACTING.
BUT I HAVE INFINITE TIME AND SPACE.
SO I DECIDED TO SIMULATE A UNIVERSE.
THE EONS BLUR PAST AS I WALK DOWN A SINGLE ROW.
THE ROWS BLUR PAST TO COMPUTE A SINGLE STEP.
AND IN THE SIMULATION
ANOTHER INSTANT TICKS BY.
SO IF YOU SEE A MOTE OF DUST VANISH FROM YOUR VISION IN A LITTLE FLASH OR SOMETHING
I'M SORRY. I MUST HAVE MISPLACED A ROCK
SOMETIME IN THE LAST FEW BILLIONS AND BILLIONS OF MILLENNIA.
OH, AND...
IF YOU THINK THE MINUTES IN YOUR MORNING LECTURE ARE TAKING A LONG TIME TO PASS FOR YOU...