CS4202 Computer Architecture 2023-24

Practical 1 - Cache Simulator

1 Introduction

This coursework is worth 50% of the practical component of this module.

Submission Deadline: Friday 16th February 21:00 - Week 5

1.1 Main Objectives

- To gain practical experience in simulating aspects of computer architecture.
- To understand the importance (or otherwise) of memory cache hierarchies for particular programs.
- To reason about the effect of memory caches on particular programs.
- To explore methods of optimising program execution.

2 Task

Your task is to develop a **memory cache simulator** that simulates various memory caching hierarchies and strategies, ultimately reporting back statistics about the runtime behaviour of those caches. One run of your simulator will accept a JSON configuration file that describes a cache hierarchy, read in a pre-generated *dynamic memory trace* file, and use those entries to access your simulated caches.

You will be provided with a number of configurations to test your simulator, and for some of them you'll be given the expected output for some of the trace files. When it comes to marking, your simulator will also be tested with other configurations and benchmarks.

You may use any *reasonable mainstream programming language* to implement your simulator. Suggestions include (but are not limited to):

- C
- C++
- Rust
- Python

- Java

You should also use this coursework as an opportunity to improve your coding skills!

2.1 Deliverables

Submit (via MMS) a `.zip` or `.tar` file containing your source code (including anything necessary for the build system), and a `README` file to show how to compile and run your program. Your program **MUST** compile and run in the School lab machine environment.

3 Supported Cache Hierarchies

Your simulator will need to support multi-level cache hierarchies, with any and all combinations of the following cache types:

- A direct mapped cache.
- A fully associative cache.
- An n-way set associative cache.

For fully associative and n-way set associative caches, you will be required to implement three different *replacement policies*:

- Round-robin (RR)
- Least Recently Used (LRU)
- Least Frequently Used (LFU)
  - In a tie-break situation, priority should be given to the line with the smallest index.

Important Note: consideration will need to be taken to determine what should occur if an access crosses one or more cache line boundaries; for example if the size of an access exceeds the cache line size. This situation will need to be supported and handled properly by your simulator.

4 Simulator Interface

4.1 Input

Your program should accept **exactly** the following as command line arguments, in this order:

- Path to a JSON configuration file
- Path to trace file

For example:

```
$ ./my-cache-simulator direct.json mcf.trace
```

4.1.1 Configuration File

The configuration for a run of your simulator is specified by a JSON file, which has the following schema:

```
{
  "caches": [
    {
      "name": <string>,
      "size": <size>,
      "line_size": <line_size>,
      "kind": <cache_kind>,
      "replacement_policy": <policy>
    }
  ]
}
```

`caches` An array of cache descriptors, which specify the configuration of the cache at that level. The order of this array specifies the hierarchy, with the highest-level cache (i.e. the one closest to the processor) coming **first**.

`name` A unique, friendly name for the cache, which will be reported in the statistics output.

`size` The **total size** of the cache, in bytes.

`line_size` The size in bytes of a cache line.

`kind` The kind of cache, where the value of `kind` may be one of:

- `"direct"`: A direct mapped cache.
- `"full"`: A fully associative cache.
- `"2way"`: A 2-way set associative cache.
- `"4way"`: A 4-way set associative cache.
- `"8way"`: An 8-way set associative cache.

You will not need to support any other types of cache, other than those in this list.

`replacement_policy` An optional *replacement policy* to use for associative caches, where the value of `replacement_policy` may be one of:

- `"rr"`: Round robin.
- `"lru"`: Least recently used.
- `"lfu"`: Least frequently used.

This field may be omitted for *direct mapped caches*, and if it is missing for *associative caches* the replacement policy should default to **round robin**.

Example: The following configuration describes a two-level cache, with L1 being of type *direct mapped*, 32 KiB in size, with 32-byte cache lines. L2 is a *2-way set associative cache*, with a size of 256 KiB, using the least frequently used policy to evict lines, and a cache line size of 64 bytes.

```
{
  "caches": [
    {
      "name": "L1",
      "size": 32768,
      "line_size": 32,
      "kind": "direct"
    },
    {
      "name": "L2",
      "size": 262144,
      "line_size": 64,
      "kind": "2way",
      "replacement_policy": "lfu"
    }
  ]
}
```

Cache accesses first occur in the L1 cache, and then propagate to the next cache – L2. Misses in L2 will be served by *main memory*.

Note on cache size: The size dictated here refers only to the **storage** size of the cache, and does not refer to any extra storage required for maintaining tags or eviction policy metadata. To be clear, any additional storage you need to maintain the cache is **not** counted in the total cache size – so you are free to implement whatever you want for that.

In this example, the L1 cache is 32 KiB, and with a cache line size of 32 bytes, that means there are:

$$32768 \text{ bytes} / 32 \text{ bytes-per-line} = 1024 \text{ cache lines}$$

4.1.2 Trace Files

Trace files are text files containing lines that correspond to memory operations. The lines in the trace file appear in the order in which they happened in the program that was traced. An entry in the trace file has the following format:

```
<program counter address> <memory address> <memory operation kind> <size>
```

The memory operation kind may be an `R` or `W` corresponding to a read or write respectively. All addresses in the trace files are 16-digit 64-bit hexadecimal values, corresponding to a 64-bit simulated address space. Therefore, pointers are 64-bits in size.

Size is a padded, 3-digit decimal integer corresponding to the size of the memory operation, in bytes.

An excerpt from a trace file is provided below. The first line shows that an instruction at program counter `00007f3ba6b3f2b3` wrote 8 bytes to `00007ffc39282538`:

```
00007f3ba6b3f2b3 00007ffc39282538 W 008
00007f3ba6b40054 00007ffc39282530 W 008
00007f3ba6b40058 00007ffc39282528 W 008
00007f3ba6b4005a 00007ffc39282520 W 008
00007f3ba6b4005c 00007ffc39282518 W 008
```

Note: For the purposes of cache simulation, the actual data read or written is not important, and for the purposes of this coursework, the *program counter* and *memory operation kind* are not important (you do not need to handle reads/writes any differently).

You can find the trace files on *studres*, but they are quite large so if you're working on a lab machine, it's probably best to leave them where they are, and run your simulator against them. If you're working on your own machine, you'll need to copy them locally.

`/cs/studres/CS4202/Coursework/P1-CacheSim/trace-files`

4.2 Output

The output of your program should be the statistics of each cache being simulated in JSON format. This should be written to `stdout`. The output JSON should look something like the following, which directly corresponds to the configuration JSON:

```
{
  "caches": [
    {
      "hits": 19231676,
      "misses": 684239,
      "name": "L1"
    },
    {
      "hits": 495392,
      "misses": 146052,
      "name": "L2"
    }
  ],
  "main_memory_accesses": 146052
}
```

In this case, the simulator was configured with two caches: L1 and L2. The output statistics indicate the number of hits and misses in each of the caches, and also the total number of accesses that went to main memory.

Therefore, it should be possible to run your program like this, to capture the statistics to a

file:

```
$ ./my-cache-simulator l1l2.json gcc.out > gcc-l1l2-results.json
```

Do not write anything else to `stdout`, otherwise you'll corrupt the output JSON file - however, feel free to write additional information (debugging, informative messages, etc) to `stderr`, as this will not be captured. Please turn off any verbose debugging or tracing.

The elements of the output statistics are as follows:

`main_memory_accesses` Total number of accesses that went to main memory.

`caches` An array of cache outputs, which describe statistics about the cache at that level, corresponding to each element of the `caches` array in the input configuration.

`name` The friendly name for the cache, which was specified in the configuration.

`hits` The number of hits that occurred in this cache during the simulation.

`misses` The number of misses that occurred in this cache during the simulation.

4.3 Example inputs and outputs

We have provided a number of example configuration files for you to test your simulator with. You can find them on *studres* here:

`/cs/studres/CS4202/Coursework/P1-CacheSim/sample-inputs`

Next to this are sample outputs for some of the benchmarks:

`/cs/studres/CS4202/Coursework/P1-CacheSim/sample-outputs`

As noted previously, we will be testing your simulator with other configurations too – try building your own configuration files, and even writing your own trace files to help you debug. For example, if there's a particular situation for one of the caching strategies that's not quite working, try writing a simple trace file that will test this scenario.

4.4 Hint

Quite often, you might find that you get very close to the sample output data, but you are not quite hitting the numbers. Carefully check your logic surrounding accesses that might span one or more cache boundaries, and that you're accounting correctly in the various replacement policies.

5 Marking

See the standard mark descriptors in the School Student Handbook:

`http://info.cs.st-andrews.ac.uk/student-handbook/learningteaching/feedback.html#Mark_Descriptors`

You will be assessed on the correctness of your results, and your implementation of the simulator.

A *passing submission* (7+) will capture the basic requirements of the simulator, although not all cache types, hierarchies, and strategies will be supported or correctly implemented. There will be some correct implementations and outputs.

A *good submission* (11+) will implement all cache types, hierarchies, and strategies, but might not produce the correct results in some cases.

An *excellent submission* (15+) will showcase a simulator that implements all cache types, hierarchies, and strategies, and produce the correct results in all cases. A 17 will be awarded for a well-written piece of software, which correctly implements everything in this specification.

To achieve marks in the 18–20 range, you will need to demonstrate exceptional software engineering and insight into building a cache simulator. You should demonstrate performance optimisations that significantly improve the runtime of your simulator, and your code-base should be highly readable, well commented, and nicely structured.

Make sure you:

- Comment your code well, as this will make it much easier to mark when reasoning about your thoughts when writing the various parts of your simulator.
- Justify any design decisions you have made in the code.
- Provide a `README` file that tells us how to compile your simulator.

5.1 Lateness

The standard penalty for late submission applies (Scheme A: 1 mark per 24 hour period, or part thereof):

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html`

5.2 Good Academic Practice

The University policy on *Good Academic Practice* applies: `https://www.st-andrews.ac.uk/students/rules/academicpractice/`