Technische Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Carsten Trinitis

Course IN2075: Microprocessors
Winter Term 16/17

Exercise 2: Caches

In these exercises, we are investigating the cache hierarchy of microprocessors.

Exercise 2.1: Cache Line Length

The following idea can be used to write a micro-benchmark for cache line length determination.

- Create an array of `N` bytes (e.g. `N = 1024*4096`).
- Write a routine containing a loop which accesses the array with 'stride' `i`, that is, it reads every i^{th} byte.

Hint: Sum up the values and print the result in order to avoid the compiler optimising away your code (or tell the compiler to omit optimisation). Check the assembly output to make sure that you are measuring the correct code!

- Measure the time the loop takes for different values of 'stride' `i`, calculate the average clock cycles required for one read access, and plot the results.

Can you determine the size of the cache line by interpreting the graph?

Exercise 2.2: Cache Sizes

By using a different micro-benchmark, the sizes of the different caches can be obtained. The idea is as follows:

- Create an array of `N` memory blocks (each block is as long as a cache line).
- Measure the time required to do 1 million (or more, if applicable) random accesses to this array. Calculate the average number of clock cycles needed for each access (cycles per instruction, CPI).

The problem here is to do the random accesses. Apparently, you cannot generate random numbers during runtime as this would spoil the measurements. An efficient way to generate random accesses is *pointer chasing*:

- each memory block contains a pointer to the next memory block. The order in which the blocks are accessed can be determined beforehand, outside the measurement loop. You can either use a random sequence (however, make sure that each element is visited exactly once), or the following pseudo random rule:
  - Memory block at address `x` points to `(x + stride) % N`
    What is a good value for `stride`? (why?).

Hint: You should also ensure that '`stride`' and '`N`' do not have a common divisor. In this case, not all elements of the array will be accessed, which also spoils the measurement. The easiest way to do this is to check if element 0 is accessed more than once within the first `N` accesses. In this case, the measurement can be aborted and no output should be printed. Accessing the array `N` times before the measurements also makes sense, because then it is already in cache (if it fits in). Thus "cold-start" cache misses are avoided.

Plot the CPI-values for different `N`. Can you determine the size of the caches from the graph? What other properties of the caches can you identify?

Exercise 2.3: likwid

Use the `likwid` performance tools to identify the cache hierarchy.

- Hava a look at, download and install `likwid`, a set of performance tools, from [`https://github.com/RRZE-HPC/likwid`](https://github.com/RRZE-HPC/likwid)

- Execute the '`likwid-topology`' tool

Hint: Use command line parameter '-c' to get additional information about the caches .
Compare the report from l`ikwid-topology` with your own findings in Exercises 2.1 and 2.2!

Play around with the `likwid` tools and find out more about your processor, also in combination with the code you wrote in exercise 1!

Exercise 2.4: hwloc/lstopo

Install `hwloc` on your Linux computer and generate a graphical view of your cache hierarchy. Does it match your previous results?

Prepare a short presentation on what you have found out on your computer!

General remark: Please DO form teams of three to four, if you get stuck, consult your fellow team members!