



F. Wermelinger
Office: Pierce Hall 211

Lab 3

Experiment with false sharing and cache thrashing

Issued: February 20, 2023
Due: March 10, 2023 11:59pm

In this lab you experiment with false sharing that can happen in multi-threaded programs. In a second task we experiment with cache thrashing which can happen in both, single- or multi-threaded codes. The two phenomena are not the same. The topics of this lab are related to lectures 3 and 9.

Submission instructions: please see the lab submission section in the syllabus.^a

^a<https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#lab-submission>

Task 1: False Sharing

In this task we want to reproduce the strong scaling plot that was shown in lecture 9 based on the example to approximate π . The task was to approximate the integral

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx \quad (1)$$

using a shared memory code. The problem of false sharing lies in *simultaneous* access of the same cache line by two (or more) different threads. The cache coherency protocol must ensure that the requested data for every cache line access is up to date (a cache line is marked “dirty” if it has been modified for somebody with exclusive access). Hence, when two or more threads simultaneously modify the same cache line, the cache coherency protocol will be under high pressure to ensure consistency for each of the cache line copies owned by the individual threads. As a consequence, performance degrades under the high load on the cache coherency protocol.

a) The core computational loop used for this problem had the following structure

```
1  #pragma omp for nowait schedule(static)
2  for (size_t i = 0; i < nsteps; ++i) {
3      const double x = (i + 0.5) * dx;
4      tpi[tid] += 4.0 / (1.0 + x * x);
5  }
```

where `tpi` is an array in shared memory. Two threads with ID's `tid` and `tid+1` are likely to share the same cache line for their access into the array. If you were to implement this loop as shown, why will you *not* observe performance degradation due to false sharing if you compile optimized code?

Hint: Look at the update rule in line 4 and think about how the compiler performs this update. What type of memory will it use?

- b) Reproduce the strong scaling plot that has been shown in the lecture, where the effect of false sharing has been demonstrated. Implement the TODO labels in the file [code/t1/false_sharing.cpp](#). You may use the provided `plot.sh` script to generate the strong scaling plot from your experiment data.

To build the code you can run

```
$ make
```

or to run an experiment

```
$ make clean run
```

which will clean up the working directory, compile the code and run an experiment. See the Makefile for the details of the targets. To clean up an experiment run `make clean`. *Remember that you should perform this work on a compute node and not a login node.* If you test the code on your laptop, you need to update line 18 in [false_sharing.cpp](#). Include the generated plot and data files in your final lab submission. Do not submit any executable code that was not already present in the lab handout.

Task 2: Cache Thrashing

In lecture 3 we studied caches in recent CPU architectures. The 3 C's stand for the three types of misses you can observe in a cache

1. compulsory misses (cold misses)
2. capacity misses
3. conflict misses

Conflict misses are the hardest to understand and we will use this task to deepen our understanding of their impact on performance.

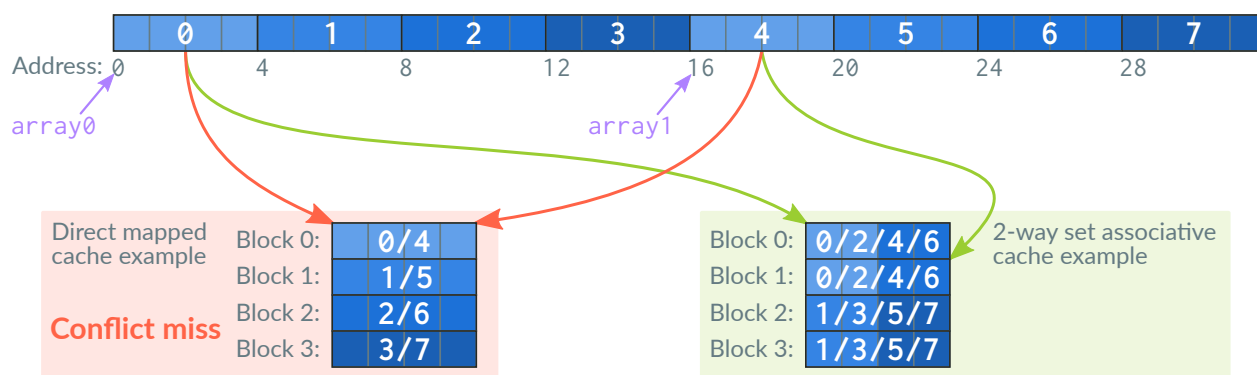


FIGURE 1: Conflict miss sketch from lecture 3

Figure 1 shows the sketch we have discussed in the lecture and how different cache blocks (the bold white indices) map into an n -way set associative cache (the one on the right in the figure is 2-way set associative). An n -way set associative cache offers a good trade-off between manufacturing cost and small latency, which is the reason this is the technology used in almost all L1 caches of recent CPUs. The two cache locations indicated by “Block 0” and “Block 1” in the 2-way set associative cache in the figure above compose what is called a *set* and only certain cache blocks can map into such a set. If one location in the set is already taken by another cache block, there would be $n - 1$ other locations the cache block could map to. If all n spots are taken and another cache line must be brought in, then an existing cache block must be evicted due to this *conflict miss*. If you happen to have a bad alignment of the data you are working with, it is possible that you continuously exhaust the n available map locations in any set, causing conflict misses which *severely* degrade the performance of your application.

- Check out an (interactive) compute node on the cluster and figure out the associativity n of the L1 data cache on the CPU you have access to.

Hint: Navigate to the directory `/sys/devices/system/cpu/cpu0/cache` and inspect the data in the four directories to find the answer.

- The provided skeleton code [code/t2/associativity.cpp](#) is a small benchmark to analyze conflict misses. The code allocates a large coalesced memory buffer to hold n_{arrays} , each holds m elements of type `double`, where m is a power of two. This is shown in figure 1 by `array0`, `array1` and so on until n_{arrays} . We use an index k to denote the array ID. For example $k = 0$ corresponds to `array0` and so on. In the `run()` function we continuously increase the number of such arrays that we use when we measure the performance. The idea is that for some k you should observe something happening to your performance measurements.

Spend some time with the code and make sure you understand what it does. In the `measure_flops()` function there is one `TODO` that you must complete. That is, complete the two innermost for-loop indices and figure out the correct indexing pattern into the buffer array such that you recover the behavior shown in figure 1.

When done, compile the code with

```
$ make
```

and run it with

```
$ ./associativity
```

This will generate two data files that you can plot with the provided `plot.sh` script. Append the generated plot in your final lab submission. *Note that this application runs single threaded unlike in the previous task of false sharing.*

Briefly comment on your observations and journal them in a markdown file named `observations.md` inside your lab root directory. For what k do you observe anomalous behavior of your experiment data? Why does the padded version behave differently?

Hint: *The missing loop boundaries will depend on m and k .*