Practical Work: Memory Effects

Master CSMI
Compilation & Performance
Bérenger Bramas

December 2, 2020

1 Summary

In this work, you analyze the behavior of the memory accesses and some mechanisms in the caches and the CPU.

2 Ressources

- The notes from CnP course

3 Practical work organization (always the same)

In the practical work, you will obtain the code from my repository and push it to your repository. Therefore, you will have to clone one branch per session and push it to your own repository. It is mandatory that you **commit and push** frequently (after each question and at the end of the session, at least) such that I can easily look at what you have coded at the end of the session, how you did progress (and potentially compare it with the latest version you will have).

It is required that you filled the report.md file to let me know what you did.

In the rest of the document, we consider you have a repository named *cnp-tp-2020* on *git.unistra.fr* that is private but that I can access in read.

3.1 Get the practical work

Consider you are in your project directory do the following:

```
# Clone my repo
git clone https://git.unistra.fr/bbramas/csmi-tp-2020.git --branch=TP7 csmi-tp7
# If you use SSH, use:
# git clone git@git.unistra.fr:bbramas/csmi-tp-2020.git --branch=TP7 csmi-tp7
# Go in the newly created directory
cd csmi-tp7
```

3.2 Add your repository as remote

You will push on your own repository:

```
# Rename my remote
git remote rename origin old-origin
# Add your own remote
git remote add origin https://git.unistra.fr/[YOU LOGIN HERE]/cnp-tp-2020.git
# If you use an SSH key:
# git remote add origin git@git.unistra.fr:[YOU LOGIN HERE]/cnp-tp-2020.git
# Push the current branch and active the tracking
git push -u origin TP7
```

3.3 During the session and while you work on the project

After each question or important modification push the current changes:

```
# No matter where you are in the project directory
git commit -a -m "I did something"
git push
```

3.4 When you are done

You have fully finished your work (at most D+14 H-2):

```
git commit -a -m "I did something"
git push
```

3.5 Important!

Do not forget that you have to fill the report.md file and commit it. Remember to commit regularly to keep track of your work and let me see a history of it if I need it. Do not share any code with someone else, as I am here to answer all questions and support all of you. Remember that you have questions to answer in the moodle before the end of the session and that you must push your branch at the end of the session too. Additional credits can be obtained if you make some modifications after the session to improve your solution. Changes can be made until two weeks after the session minus two hours, ie you must push before the beginning of the n+2 practical work. Do not remove code from the test functions as I use them to evaluate your code. Therefore, if you need you can add extra functions for your own testing/debugging. If some of them are showing interesting things about your code, simply leave a comment in the code and the report.md.

3.6 Compilation

To compile, we use CMake:

```
cd TP7
mkdir build
cd build
cmake ..
make # Will make all
make something # Will build only something
VERBOSE=1 make # Will show the commands used to compile (including the flags)
```

Make sure you compile in release! This can be done using *ccmake* **or** *cmake..*$-DCMAKE_BUILD_TYPE =$ *Release*

4 Reminder

The memory is organized in with multiple caches (the closer to the CPU, the faster but also the smaller). The data from the main memory are moved to the caches when they include a variable that the CPU needs to process/read. The data are moved by ”line” of 64 bytes and replaced using a LRU heuristic. These two mechanisms promote the locality effects (temporal and spatial).

Remember to pin the process using ***taskset*** **or some of the effects we want to investigate will vanish because of process preempting. Also, make sure to compile in RELEASE.**

5 Memory bound

The data moves between the caches are moved by line of 64 bytes. Therefore, when a value in an array, $array[idx]$, is used a complete cache line is move to the L1 cache. As long as this cache line is not evicted, accessing any variable in it can be done with zero data transfer (locality principle).

In the file memoeff.cpp update the *memory_bound* function to measure this effect. To do so, we will compare the iteration on an array while accessing all the values (already done) vs. iterating one cache line after the other.

Remark, as you can notice the array *arr* is 64 bytes aligned, and thus each $k * 64 bytes/sizeof(longint)$ starts a new cache line. Run with *taskset* for more accurate results.

6 Instruction level parallelism

As we have seen in class, modern CPUs are able to do operations in parallel, but these operations must be independent. In the file memoeff.cpp update the *ilp* function to measure this effect. To do so, you will create two loops that will iterate *nbRepeat* times. In the first one, you will increment $arr[0]$ with $coef$ and then multiply $arr[0]$ with $coef$ (so both operations are dependent because they work on the same variable). In the second loop, you will increment $arr[0]$ with $coef$ and then multiply $arr[1]$ with $coef$. The amount of work is the same in both loops but you should notice different execution time. Run with *taskset* for more accurate results.

7 Missprediction

As we have seen in class, modern CPUs use a pipelining mechanism to work on different operations at the same time. However, when there is a conditional jump (such as a *if* statement), the CPUs has to decide which of the execution is the more probable to happen. To make a prediction it uses an history of the previous times the test was found, and for example and could decide to look at the last 3 tests and take the execution path that has the majority. However, usually we do not code with this in mind, and miss-prediction can be expensive. You will update the *missprediction* function to show these effects. The idea is to compare two loops. In the first one a is incremented by $arr[i]$ every 1023 times over 1024, else b is incremented. In the second loop, a is incremented by $arr[i]$ every 2 times over 3, else b is incremented. Run with *taskset* for more accurate results.

8 False sharing

We know that a complete cache line is moved even if a thread works on a single byte. When threads works on different bytes that belong to the same cache line the cache coherency system but perform updates and moves in order to maintain valid memory values. Update the *falsesharing* function such that in the second part the threads work on contiguous values. Do not execute with *taskset* (but you could use $OMP\_PROC\_BIND = true$).

9 Cache effect

In this example, we will study the impact of re-using values that are in cache vs. loading a new cache line. More precisely, we will also look at the how the cache lines are evicted based on the k-way property.

Start by looking at your CPU to know the associativity: $getconf -a|grepCACHE$ (look for $LEVEL1\_ICACHE\_ASS$

You will implement three variants of a code that perform the same amount of work but on different parts of the memory. We always do *nbRepeat* repetitions, we work on *kway* cache lines twice and work on all values of the cache lines (*nbLongIntInCacheLine*). The variants are given by:

- variant 1: Each cache line entry can store up to *kway* different lines. So here, we will work on *kway* lines twice and the lines can stay in the cache (they are never evicted).

```
for(long int repeat = 0 ; repeat < nbRepeat ; ++repeat){
    for (long int i = 0; i < kway; i++){
        for(long int idxElement = 0 ; idxElement < nbLongIntInCacheLine ; ++idxE
            arr[i * chunk_jump_long_int + idxElement] += 3;
        }
    }
    for (long int i = 0; i < kway; i++){
        for(long int idxElement = 0 ; idxElement < nbLongIntInCacheLine ; ++idxE
            arr[i * chunk_jump_long_int + idxElement] += 3;
```

- variant 2: In this case, we work on very close lines that will be loaded in different cache entries.

```
    for(long int repeat = 0 ; repeat < nbRepeat ; ++repeat){
        for (long int i = 0; i < 2*kway; i++){
            for(long int idxElement = 0 ; idxElement < nbLongIntInCacheLine ; ++idxE
                arr[i * nbLongIntInCacheLine + idxElement] += 3;
```

- variant 3: In this case, we work on a single lines (all the cache block will be stored in the same cache entry, so it will evict the cache lines all the times).

```
    for(long int repeat = 0 ; repeat < nbRepeat ; ++repeat){
        for (long int i = 0; i < 2*kway; i++){
            for(long int idxElement = 0 ; idxElement < nbLongIntInCacheLine ; ++idxE
                arr[i * chunk_jump_long_int + idxElement] += 3;
```

Output on my machine is:

```
>> all in the same way twice : 7.8e-07
>> all in different way : 8.79819
>> all in the same way conflict : 77.4788
```

10 Keep time to answer the test, which includes questions on the other functions