# *High Performance Programming, Lecture 6*

*Optimization II: ILP, branching, loop unrolling, loop fusion, vectorization/SSE (intro)*

Please **verify that your student e-mail is working!** Did you receive e-mails about the course? If not, please fix it.

Assignment 3 is in Student Portal. Deadline 15.02.

# *Assignment 2. Common problems.*

**Follow instructions exactly!**

Try your code on one of the University's Linux computers:
– do you need to link with the math library (-lm)?
– fix compiler error and warnings.

**Part 1**

– do not print extra empty lines

**Part 2**

– check if the fopen call failed and output an error message in that case: fopen returns NULL in case of error.

**Part 3**

– when nothing has changed, make should say "make: 'january' is up to date."
– if the checking script fails, go to the directory tmpdir_for_checking/A2/part3
– for each malloc you should have a free
– do not use unitialized pointers
– program that you submit should start with an empty list

# *New example codes in Student Portal*

Check new example codes under Example programs / strings:

`gets_string.c` and `argv_gets_test.c`

They present different ways to read data into the string. For example, how to safely read the string on the unknown length.

# *Pure functions (example from Lecture 5)*

https://godbolt.org/z/cYWoQq

# *Optimization overview*

Program performance can be improved by:

- I: Doing less work
- II: Waiting less for data
- III: **Doing the work faster** $\longleftarrow$ this lecture!
- IV: Using less space (to fit a bigger problem)

# *Outline*

Things covered in this lecture:

- Instruction-level parallelism (ILP)
- Branching
- Loop unrolling
- Loop fusion

# *Reading*

Reading for this module:

- Fog 7.12 : "Branches and switch statements"
- Fog 7.13 : "Loops"
- Fog 11 : "Out of order execution"

# *Instruction-level parallelism (ILP)*

Instruction-level parallelism (ILP) allows CPU to perform several instructions simultaneously.

**Only works if the program contains independent instructions**; if each instruction depends on the result of the previous instruction, they cannot run in parallel.

Optimizing a program to make use of ILP typically means rewriting the code to get more independent instructions.

# *Two important concepts*

*Pipelines and vector operations*

Focus on two concepts (two ways of achieving ILP):

**Pipelining**: computations inside CPU are done in stages forming a pipeline. Typically there is more than one pipeline. Attempts to keep every part of the processor busy with some instruction.

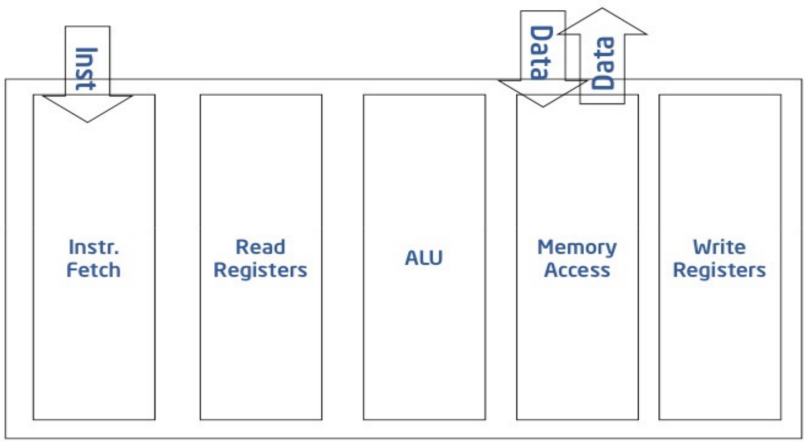
**Vector operations**: modern CPU:s have special arithmetic units that allow performing operations on a vector of operands, in parallel. *(see Lecture 8)*

# *Instruction stages, example*

In a pipelined CPU, the execution of an instruction is separated into several stages. Each stage is performed in a different part of the chip.

The classic RISC pipeline comprises:

- 1. Instruction fetch (IF)
- 2. Instruction decode/register fetch (ID)
- 3. Execution (EX)
- 4. Memory access (MEM)
- 5. Write-back (WB)



# *Observations*

- All steps are performed in succession
  - Steps are waiting for work most of the time
  - Hardware sits idle
- The CPI (cycles per instruction) is at least 5 (more if we have to wait for memory)
- Solution: use **pipelining**
  - Start a new instruction each cycle
  - Each clock cycle corresponds to a pipe stage
  - This maximizes the use of hardware components

# *Pipelines*

```
                  -----> time ----->
        1   2   3   4   5   6   7   8   9  10  11  12
Inst1   IF  ID  EX  MEM WB
Inst2       IF  ID  EX  MEM WB
Inst3           IF  ID  EX  MEM WB
Inst4               IF  ID  EX  MEM WB
Inst5                   IF  ID  EX  MEM WB
Inst6                       IF  ID  EX  MEM WB
Inst7                           IF  ID  EX  MEM WB
```

- Pipelining allows several instruction steps to execute simultaneously
- In this example, up to 5 x faster execution
- Requires that instructions are independent

The downside is that the minimal time to execute a single instruction is five clock cycles (one cycle per stage).

## *Pipeline speedup*

When using a pipeline we can execute one instruction per cycle **if the pipe is full**.

In general a filled pipeline of length N will give a speedup of N.

The same technique is used in fabrication industry: Cars, mobile phones etc.

# *Problems with pipelines*

Hazards (instructions that depend on each other) reduce pipelining

- Instructions must be independent
- If the pipeline detects a hazard it causes a pipeline bubble
  Some "empty" or "no-operation" instructions must then be inserted into the pipeline instead of useful instructions.

# *Hazard*

Example:

```
1: add 1 to R2
2: move R2 to R6
```

To make it work, the CPU would need to wait and not insert the move operation into the pipeline until it was certain that the result of the add was available.

Intel designing CPUs with up to 31 stages!

Today: 7-20 stages and 4-6 pipes

Good things with long and many pipelines:

- + Shorter cycletime (more MHz)
- + ILP

Difficulties:

- - Branch delay more expensive
- - Even harder to find "enough" independent instructions to keep all pipelines full.
- - Bigger power consumption, more complex circuits

# *Instruction scheduling*

- Compilers can reorder the instructions
- Compilers know which resources are available and how long instructions take
- Compilers try to:
  - Tries to minimize pipeline bubbles
  - Keep all resources busy (increase ILP)
- Sometimes we must have bubbles due to hazards
  - Branch prediction hardware helps
  - Out-of-order execution helps

# *Example: the XOR swap*

Consider the two variable swapping codes below:

```
// e.g. x = 1011, y = 0111

 x = x^y;        // x = 1100
 y = x^y;        // y = 1011
 x = x^y;        // x = 0111

 t = x;
 x = y;
 y = t;
```

"XOR swap" allows the swap to be done without using any additional register. However, strictly sequential order needed (ILP not possible). The second way, using a temporary variable t, is better from an ILP point of view since then two instructions can be executed in parallel.

# *Branch prediction*

Most CPUs have *branch prediction* capabilities; they attempt to guess the outcome of a branch, so that execution can continue without pipeline stalls.

Branch prediction can make a program run fast in spite of the existence of branches, especially if the branches are very predictable.

**Branch prediction does not solve all problems with branches**; it is still always better to avoid branches in inner loops.

Depending on workload, number of branches and predictability of branches, the program will still be slowed down due to branches.

# *Branch prediction*

Typical timings:

branch instruction if predicted correctly: 0-2 clock cycles
branch misprediction: 12-25 clock cycles

The number of branches should be kept small in the critical parts of the program.

# *Example technique: bounds checking*

```
const int size = 16; int i; float list[size];
...
if (i < 0 || i >= size) {
cout << "Error: Index out of range";
}
else {
list[i] += 1.0f;
}
```

The two comparisons can be replaced by a single comparison (type conversion generates no extra code at all):

```
if ((unsigned int)i >= (unsigned int)size) {
cout << "Error: Index out of range";
}
```

# *Example technique: simpler loop control condition*

`string_loop.c`

(Compiler explorer link: `https://gcc.godbolt.org/z/zoT7YP`)

```
#if FAST

void lowercase(char* p) {
   int i;
   for (i = 100; i>0; i--)
        *(p++) |= 0x20;
}

#else

void lowercase(char* p) {
   while (*p != 0)
        *(p++) |= 0x20;
}

#endif
```

# *Branch prediction example*

See `branch_prediction` directory. See function f in `testfuncs.c`.
In the main function we assign value to an array:

```
a[i] = (rand() % 1000) * 0.001; // value between 0
    and 1
// a[i] = (rand() % 1000) * 0.0005; // value between
    0 and 0.5
```

Try to run code when elements of an array a are between 0 and 1. Then let elements of the array will be between 0 and 0.5. Do you get any performance improvement?

# *How to get more independent instructions? How to reduce number of branches?*

Techniques:

*Loop unrolling*: rewrite loop to do more work in each loop iteration

*Loop fusion*: turn two (or more) loops into one

# *Loop unrolling*

Loop unrolling: rewrite loop to do more work in each loop iteration. If statements in loop are not dependent on each other, they can be executed in parallel.

```
for(i = 0; i < N; i++) {
  // do something related to i
}
```

$\longrightarrow$

```
for(i = 0; i < N; i+=3) {
  // do something related to i
  // do something related to i+1
  // do something related to i+2
}
// Note: in case N does not divide evenly by 3,
// we need to stop the loop earlier and
// take care of remainder part separately.
```

# *Loop unrolling*

The number of loop iterations that are grouped together is called the *unroll factor* or sometimes *unroll depth*.

The **optimal unroll factor can depend on computer architecture**, number of pipelines in the CPU, vectorization capabilities, etc. Software 'tuning' often involves experimenting with unrolling.

There is no computational gain if the computational time is determined by the computation inside the loop.

Loop unrolling can often be done automatically by the compiler, but not always. See examples in Lab 3.

# *Loop fusion*

Loop fusion – turn two (or more) loops into one

```
for(i = 0; i < N; i++) {
  // do "work A" related to i
}
for(i = 0; i < N; i++) {
  // do "work B" related to i
}
```

$\longrightarrow$

```
for(i = 0; i < N; i++) {
  // do "work A" related to i
  // do "work B" related to i
}
```

# *Tweak your code to allow optimizations*

**Avoid if-statements in your innermost loops!**

Sometimes the programmer knows something that allows the code to be rewritten with no (or at least fewer) if-statements inside loop.

# *Performance cost of using "virtualization" tools*

So-called "virtualization" tools such as VMware or VirtualBox allow running a Linux "virtual machine" on e.g. a Windows computer.

However, **there is a performance cost**.

$\longrightarrow$ Avoid virtualization tools if you want the best possible performance.

An alternative is **Cygwin**, which provides Linux-like tools for Windows, **running directly in Windows**, not via "virtualization". If using Cygwin on a 64-bit Windows computer, make sure you are using the 64-bit Cygwin version, otherwise you will get worse performance.

# *Performance cost of using "virtualization" tools*

*Example: mmul code for matrix size 400*

CPU used: Intel Core i7-2670QM @ 2.2 GHz. Compiled using -03.

Timings in milli-seconds (matrix size 400):

| | *Loop order* | | | | | |
|---|---|---|---|---|---|---|
| | ijk | ikj | jik | jki | kij | kji |
| Native Linux, Ubuntu, gcc 5.3.1 | **95** | **34** | **93** | **522** | **37** | **523** |
| Cygwin 64-bit, gcc 5.3.0 | 97 | 35 | 103 | 542 | 38 | 541 |
| Cygwin 32-bit, gcc 5.3.0 | 99 | 58 | 102 | 589 | 61 | 566 |
| VMware Player, Ubuntu, gcc 5.3.1 | 102 | 38 | 105 | 1116 | 45 | 1182 |
| VirtualBox, Ubuntu, gcc 5.3.1 | 115 | 46 | 115 | 916 | 49 | 900 |

Test code `mmul_loop_order.c` available in Student Portal.

# *Groups for assignments*

If it turns out that your group does not work well, e.g. the other student is not contributing or collaboration is not working, tell me about it. Then we can change the group division.

# *That's all*

Questions?