Lab 6

Objective:

Continuing on from our Lab 5 ...

- The purpose of this lab is to introduce us to the idea of microbenchmarking, i.e., testing the performance of very small units of computation. To do so, we shall use more precise tools than getrusage() (used in Lab 5).

Submission – Participation Activity 10

- At the end of this lab session, you need to submit your data sheet and graph (last 2 pages of this lab) to CourSys as Lab6.pdf under Participation Activity 10.
- Make sure you write your name and student number on its top right corner.

Part 1: Benchmarking a Loop

- Login to a CSIL machine in the Linux environment, download Lab6-files.zip and unpack it into sfuhome/CMPT295. Change directory into Lab6 where we will find the usual makefile and a collection of source files.
- When we open main.c, we will see something similar to the benchmarking code in Lab 5: an array of N integers will be initialized, and a function sumPlus() will be benchmarked 20 times using getrusage().
- The function sumPlus() is written in assembly. It computes the sum of all of the positive integers within an array of integers A[n]. Have a look at the code.
- Build the executable, run it and have a close look at the results.

Woah! What is happening? Our testing does not seem to be working.

The reason for these strange results is that we are measuring something very small: the time (number of clock cycles) it takes to compute the sum of the elements contained in a very small array, which is in the order of nanoseconds, with a tool (getrusage()) that is meant to measure something much larger, i.e., computation ranging from milliseconds to seconds.

Part 2: Inline Assembly

So, let’s change our measuring tool! Let’s include microbenchmarking code within our C code.
First, let’s remove the “old” benchmarking tool in main.c by commenting out the two instances of getrusage() and replacing the line that currently computes the number of clock cycles per call to sumPlus() at the end of the loop with this one:

\[
\text{cycles}[i] = \text{end\_time} - \text{start\_time};
\]

To complete the change, we need to add code that will capture the start\_time as well as the end\_time so that their difference can be computed properly.

Still in main.c, insert the following code before the call to sumPlus():

```c
asm volatile ( "cpuid\n\t" "rdtsc\n\t" "movl \%eax, \%0\n\t" : "=r" (start_time) : : "rax", "rbx", "rcx", "rdx" );
```

What does it all mean?

- “asm volatile” tells the compiler we are going to write some inline assembly code in our C program. The volatile keyword is sometimes optional: it suggests to the optimizing compiler that it would be a bad idea to move this code (code motion) elsewhere.

- The parentheses after “asm volatile” contain four arguments, each separated by colons (“:”).
  - The first argument:
    ```c
    "cpuid\n\t" "rdtsc\n\t" "movl \%eax, \%0\n\t"
    ```
    is a string containing assembly instructions. Let’s go over each of them in this order: first “rdtsc”, then “movl \%eax, \%0” and finally “cpuid”.
  
  - Let’s start with rdtsc, the second instruction in this first argument as it is the core instruction of the microbenchmarking process. We learn from the web page [https://www.felixcloutier.com/x86/RDTSCP.html](https://www.felixcloutier.com/x86/RDTSCP.html) that this instruction reads the current value of the processor’s time-stamp counter (i.e., 64-bit value of the clock cycle counter) into the \%edx:\%eax registers, i.e., the higher 32 bits of this counter value are copied into \%edx and the lower 32 bits are copied into \%eax registers, clearing the high-order 32 bits of each
of %rax and %rdx. This time-stamp counter is incremented every clock cycle by the processor.

- The next assembly instruction we shall look at in this first argument is
  
  ```
  movl %%eax, %0
  ```

  This instruction copies the value stored in register %eax into the global variable start_time, represented here by %0.

- These two instructions seem to be doing what is needed in order to time the call to sumPlus(), so why would we require the cpuid instruction? The reason why we need this instruction is that when rdtscp is executed, it is placed in the CPU pipeline along with all the other instructions, which means that it could be run before or after some of the work we are timing. Therefore, the serializing1 instruction cpuid (https://www.felixcloutier.com/x86/CPUID.html) is issued to clear the pipeline and allow the timing instructions (the instructions contained in this first argument as well as the call to sumPlus) to execute one after the other. This way, we can truly time the execution of sumPlus().

- The second argument
  
  "=r" (start_time)

  is a comma-separated list of output variables. Here, we only have one output variable, namely start_time, which is seen as the 0th output variable of the list and is referred to by "%0" found in the instruction “movl %%eax, %0“, which was explained above. This is how the movl instruction ends up copying the value stored in register %eax (i.e., the starting time-stamp counter) into the global variable start_time.

- The third argument is a comma-separated list of input variables, empty in this case.

- The last argument
  
  "rax", "rbx", "rcx", "rdx"

  is a comma-separated list of registers that should be treated as temporary registers by these assembly instructions. Because cpuid writes to registers %rax, %rbx, %rcx and %rdx, this argument warns the compiler not to leave anything important inside these registers, in case their values get overwritten. (The effect of this argument is akin to

1 Definition: arranging in a sequence.
pushq’ing and popq’ing these registers onto the stack in order to preserve their content.)

- Place another round of inline assembly code ("asm volatile") after the call to `sumPlus(...)` in `main.c` to measure the end_time. Don’t forget to replace start_time with end_time.
- Rebuild and run the code multiple times to get a sense of how long `sumPlus()` takes for \( N = 100 \).
- It may be the case that some samples are rather large compared to the rest of the samples. It may be due to context-switching: to achieve the appearance of parallelism, the operating system switches rapidly between computing tasks (called “processes”). The large number of clock cycles may include the execution time of several processes currently run by the processor, not just the execution time of our program. Also, if the first few samples are larger than the rest, this may be due to cache misses, which we shall discuss in class during the coming lectures. In order to ignore these large samples (which would throw off our timing calculation), add the following code in `main.c`, after the computation of `cycles[i]` in the main loop:

```c
if (cycles[i] >= 2000) {
    printf(“Sample %d completed in %d clock cycles – DISCARDED.\n”, i+1, cycles[i]);
    i--;
    continue;
}
```

- **Note:** Depending on the speed of the computer we use, we may have to tweak this limit of 2000 (on my machine) either upward (perhaps 4000 to 5000 on CSIL machine) or downward.
- Let’s rebuild our code once again and run it 11 times. Discard the slowest 3 and fastest 3 average times. Then record the middle 5 average times in the table found on the penultimate page of this lab. Repeat for \( N = \{150, 200, 250\} \) and record the data in the table. Compute the averages of each data set.

**Part 3: Optimization**

The `sumPlus()` source uses the branch instruction `jle endif` to test if \( A[i] \) is greater than 0. When the CPU fails to predict the branch, perhaps close to half the time (50-50 chance that an element of \( A \) is negative), CPU clock cycles are wasted. In this part of the lab, we shall optimize the code and see just how much of a penalty is paid.
• Open `sumPlus.s` and replace the line `jle endif` with `cmovl %r8d, %ecx`. This is a *conditional move* instruction which will move the value of `%r8d` into `%ecx` only if the content of `%ecx` is less than 0. The idea is:

```assembly
if A[i] (i.e., %ecx) < 0
    %ecx = %r8d (i.e., 0)
sum += %ecx
```

This means that we need to store 0 into `%r8d`.

• Add a line before the loop that zeroes `%r8d`. Which instruction we use to do this is up to us.

• Re-build and microbenchmark the code for the same values of `N` as in Part 2, i.e., 11 times for each value of `N`, discarding the slowest 3 and fastest 3 average times, then recording the middle 5 average times in the table.

• Complete the table and then plot both datasets on the graph paper at the end of this lab. Draw a straight line through each dataset, and compute its slope. The slope should have units of clock cycles per element, i.e., CPE.

• This is the last step of Lab 6. Don’t forget to submit your data sheet and graph (last 2 pages of this lab) to CourSys as `Lab6.pdf` under *Participation Activity 10*. Make sure you write your name and student number on its top right corner.

---

*Thank you to Brad Bart for having inspired this lab.*
### Data Sheet

Data set: **Version 1 - Branch**

<table>
<thead>
<tr>
<th>N</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>μ</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data set: **Version 2 – Conditional move**

<table>
<thead>
<tr>
<th>N</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>μ</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>