CSE 221 Project Report

Yudong Wu
A53216872
yuw466@eng.ucsd.edu

February 8, 2017

1 Introduction

The goal of this project is to measure the performance characteristics of a HP's workstation in Opera group when running ubuntu 16.04.4. There is 3 people in our group: Chengcheng, Yihong and Yudong. We finished this system measurement project by C, a typical system implementation languages. Our code is compiled by gcc 5.4.0, with optimization flag **-O0**.

2 Machine Description

Our experiment machine is an HP Pavilion Elite HPE workstation, and the system running on this machine is Ubuntu 16.04.1. The specs of this machine and the operating system will be stated below.

2.1 CPU

The processor on this machine is an Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz with 4 cores. This CPU supports constant tsc feature, which makes cycle counting much more trivial. When running this project, we disabled 3 of the 4 cores, and made only a single core available for running our measurement benchmark. The specs summery for this core is listed below:

- The frequency of this CPU should be 2.80GHz (read from model infomation). After a simple measurement, we found out that the frequency of the clock of this core is 2.793291 GHz, and the clock period of this core should be

$$\frac{1.000000}{2.793291GHz} = 0.358001ns$$

- This core has 64K on core L1 cache, which 32K of them are data cache, and the rest 32K are instruction cache
- This core has 25k on core unify L2 cache
- The L3 cache is 8M, and it is shared by all cores. Since we only enable single core when running, we can seem it as a private 8M L3 cache for this core.

2.2 Memory

This machine has totally 8G main memory.

2.3 Buses

2.4 Disk

2.5 NIC

2.6 Operating System

When running this project, the operating system running on the machine is normal Ubuntu 16.04.1. The system is running under 64 bit mode.

3 CPU Operations

This section mainly reports our estimation and measurement result about the cpu operations and os service like system calls and context creation and switch.

3.1 Overhead Measurement

When performing measurement, we are using the following code to measure the overhead of a piece of code:

```
#define START_COUNT(cycles_high, cycles_low)       \
    asm volatile ("CPUID\n\t" "RDTSC\n\t" \
                  "mov %%edx, %0\n\t" \
                  "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::\
                  "%rax", "%rbx", "%rcx", "%rdx")

#define STOP_COUNT(cycles_high1, cycles_low1)      \
    asm volatile ("RDTSCP\n\t" \
                  "mov %%edx, %0\n\t" "mov %%eax, %1\n\t" \
                  "CPUID\n\t": "=r" (cycles_high1), "=r" (cycles_low1):: \
                  "%rax", "%rbx", "%rcx", "%rdx")
```

In the source code, we use RDTSC and RDTSCP to read out the timestamp counter register on the cpu. CPUID instruction serves as a barrier to ensure only cycles runs between RDTSC and RDTSCP instructions will be counted. So when we want to measure the overhead of some functions (code pieces), we can just put START_COUNT before the target code pieces and STOP_COUNT after, then we can easily get the cycles ticks during executing the code pieces we want to measure.

3.1.1 Methodology

When measuring the overhead of the measurement code we mentioned above, we use the following code snipes to perform the overhead measurement:

```
        START_COUNT(high, low);
        STOP_COUNT(high1, low1);
```

We just measure nothing between two count marcos, so that this measurement will response the overhead of the execution of thos two marcos.

When performing the measurement, we will run the measurement code in a 10 round loop, then we take the arithmatic mean of the cycles responsed as the final result.

3.1.2 Estimation and results

The estimation and the measurement result will be list in the following table:

Table 1: **Measurement Overhead: Estimation and Experiment Results**

Hardware Overhead Estimation	Software Overhead Estimation	Total Overhead Estimation	Expr. Results
6 (inst) or 44 (cycles)	0	6 (instructions)	92

When estimation, the first evidence for us to perform the estimation is the code we write and generate, following is the acutall assembly code after compilation:

```
1     4005f8: 0f a2                cpuid
2     4005fa: 0f 31                rdtsc
3     4005fc: 89 d7                mov    %edx,%edi
4     4005fe: 89 c6                mov    %eax,%esi
5     400600: 89 7d d0             mov    %edi,-0x30(%rbp)
6     400603: 89 75 d4             mov    %esi,-0x2c(%rbp)
7     400606: 0f 01 f9             rdtscp
8     400609: 89 d7                mov    %edx,%edi
9     40060b: 89 c6                mov    %eax,%esi
10    40060d: 0f a2                cpuid
11    40060f: 89 7d d8             mov    %edi,-0x28(%rbp)
12    400612: 89 75 dc             mov    %esi,-0x24(%rbp)
```

We can found out: there are 4 instructions between RDTSC and RDTSCP instruction: two of them are inline aseembly we write in code, and the rest two are compile-time generated code caused by register saving, so the totally overhead for the measurement should be 4 instructions + 2 instructions (RDTSC and RDTSCP themselves).

The second issue for estimating the hardware overhead is that, how many cycles each instruction will take, or in other words, what is the cpu's CPI (cycles per instruction). We haven't found out the most related information, so we performed the following estimation: for those 6 instructions, we assume that each cycle the cpu can commit one instruction, and the pipeline stages is around 20; as those 6 instructions don't have any data dependencies that can not be solved by data forwarding, the pipeline will not stall,and for those 6 instructions, from first instruction enter the pipeline to the last instruction commit, the cycles consumed should be:

$$20 + 4 + 20 = 44 cycles$$

After the experiments, the result shows that we underestimated the overhead by around 50%, but as we don't know much about the internal of the Intel CPU, so that this kind of mistakes is somewhat reasonable.

3.2 Loop Overhead

When performing estimation, it is reasonable that using repeated experiments to reduce the impact of variance, so before all other overhead measurement, we need to determin the overhead of the loop first.

3.2.1 Methodology

When measuring the loop overhead, we simply measures the cycles diff before and after a simple do nothing loop; in order to reduce the impact of variance, we also run this test 10 times, then take the arithmatic means of 10 samples as the final measurement result. We also perform our measurement upon loops rounds from 0 to 50, in order to see the overhead of different loop scale.

3.2.2 Estimation and results

Our Estimation and the measurement results are showed in the **Figure 1**:

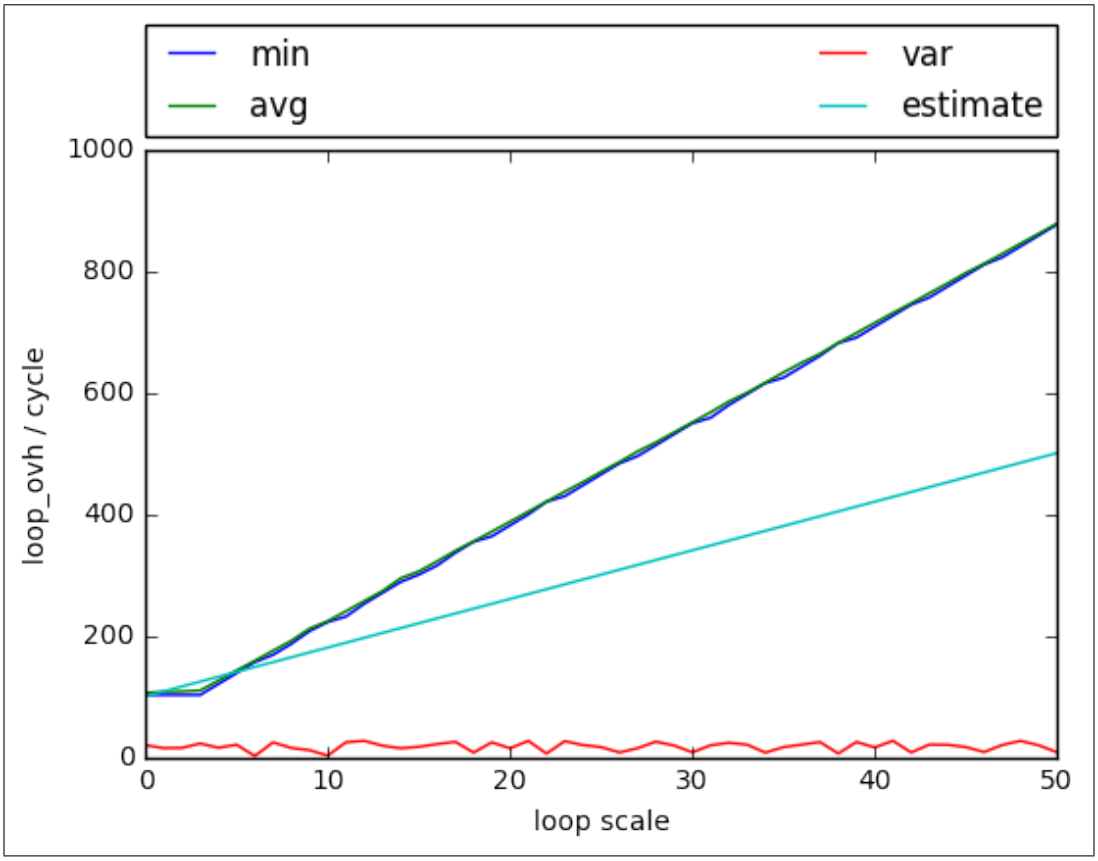

Figure 1: Loop Overhead: Estimation and Experiment Results

As shown in the graph, when estimating, we estimate the overhead increasing by 8 cycles when loop scale increase by 1. We made this estimate based on the instructions of the loop after compile:

```
4006a1: c7 45 c4 00 00 00 00   movl   $0x0,-0x3c(%rbp)
4006a8: eb 04                  jmp    4006ae <main+0xf8>
4006aa: 83 45 c4 01            addl   $0x1,-0x3c(%rbp)
4006ae: 8b 45 c4               mov    -0x3c(%rbp),%eax
4006b1: 3b 45 c0               cmp    -0x40(%rbp),%eax
4006b4: 7c f4                  jl     4006aa <main+0xf4>
```

As shown in above, there will be 4 instructions for each round of the loop, as the **Section 3.1.2** shows that, each instruction will taken more than one cycle tick in average. We finally made a assumption that each instructions will take up 2 cycles on average.

The result shows that, we still underestimated the increament ratio of the overhead, and in fact, each each loop round increase will increase the overhead by 18.

3.3 Procedure Call

3.3.1 Methodology

When measuring the overhead of a procedure call, we just put our cycles counting code before and after the procedure call; also, we reduced the variance by using the arithmatic mean of the response overhead came from 10000 times measurements.

3.3.2 Estimation and results

3.4 System Call

3.4.1 Methodology

When measuring the system calls, basically we want to measure the cycle ticks between the begin and the return points of the system call. This means that we required the kernel return the control to the process after the system call service routine finished, so that what we can measure is some 'light weight' system call, like **getpid()** or **kill()**. However, getpid() will be cached by the kernel, so that only the first call of the getpid() can trap into the kernel space, so that in the end, we choose **kill()** system call to be the system call we measured.

For specific implementation, the measurement process will first start counting cycles, then send itself a sigusr1 signal by invoke **kill()** call. In the signal handler, the process will stop counting cycles, and this period of cycle ticks will be the overhead of the **kill()** system call. We repeated the measurement routine for 10000 times, then take the arithmatic mean of the cycles cost as the final result of overhead.

3.4.2 Estimation and results

3.5 Process and Kernel Thread Creation

3.5.1 Methodology

When measuring Task, or process creation time, basically we need make some assumptions. The first one is that we assume that when the measurement process fork a child process, after fork syscall, the os will schedule either the measurement process or the child of the measurement process to be executed, however, we are not sure which process (parent, or child) will be executed first. Under this assumption, we can find out that basically what we want to get is the cycles between the point just before fork call and the point fork return.

In order to get the results we want, we implement the following measurement sequence:

1. We initialize a pipe for child and parent process to comunicate
2. We start counting cycles before we call fork
3. Both at the parent and the child process start point after fork return, we stop counting cycles
4. Child process passing the end counting to the parent via the pipe we initialized at **1**
5. We calculate both cycles spend for parent fork return and child fork return, then choose the smaller one (as the smaller one indicates that this process executes before the other one)

For the kernel thread creation, we use a similar method. The measurement sequence is listed at following:

1. We initialize a global variable for child process to store the end clock cycle count
2. We start counting cycles before we call pthread_create
3. Both at the main and the peer thread start point after pthread_create finished, we stop counting cycles
4. peer thread passing the end counting to the main thread via the global variable we initialized at **1**

5. We calculate both cycles spend for main thread pthread_create return and pthread_create start execution, then choose the smaller one (as the smaller one indicates that this thread executes before the other one)

3.5.2 Estimation and results

The estimation and the measurement result will be list in the **Table 2**:

Table 2: **Process/Thread Creation Time: Estimation and Experiment Results**

Process or Kthread	HW Overhead Estimation	SW Overhead Estimation	Total Overhead Estimation	Expr. Results
Process	NA	NA	$7.16\mu s$	...
Kernel Thread	NA	NA	$3.58\mu s$	

When estimation, it is hard for us to do the hard ware and software estimation, as there are a lot of issues hard to estimate. For example, for fork(), the kernel will create a new process control block for the new process, and then set up a bunch of member's value inside the pcb; after that, it will at least initialize the child process's address space, by creating a bunch of page tables; all these operations might cost more then 20000 cycles. Compared with process creation, thread creation is much more light weight, as thread doesn't need to construt a new address space. Finnally we made a pridiction that, process creation will cost 20000 cycles and thread will cost half of the process cost, 10000 cycles; or in other word, $7.16\mu s$ for process and $3.58\mu s$ for thread.

The results shows that

For the question how do they compare,

3.6 Process and Kernel Thread Context Switch

3.6.1 Methodology

For the overhead measurement of the context switch, the basic idea to measure the overhead is to count the cycles spend on switch from one process (thread) to another process (thread). In order to continue our measurement, we need the same assumption we made in **section ??**, that the measuring processes (threads) are the only two processes (threads) that to be scheduled. In the real system, this is impossible, but we can estimate this situation by increasing the priority of the process (threads).

For specific how to count the cycles of a context switch, we need to force a context switch happen in one process (thread) the another (thread) process can stop the counting afterwards. To achieve this goal, we implement the following measure sequence by using blocking reading pipe. To be more precisely, at parent process, we first start a pipe for communication, then fork new process; after that, parent process will first start counting cycles, then write to the communiting pipe, then call wait() to wait the child process; at the same time, child process will read from the pipe, then stop counting the cycles, and finally send the stop cycles back to the parent process via the pipe then exit; after that, parent will wake up and get the value and count the tick.

This method uses read and wait to synchronize. Either child or parent process executing first, the final result will be the period start from parent writing to the pipe, then context switch to the child, and finally the the child read from pipe.

The method for thread is similar, the differences are:

1. We use pthread_join() instread of wait()
2. We use global variable, instead of pipe, to pass the final tick to the main thread;

3.6.2 Estimation and results

For the question about difference between process context switch and kernel context switch, as the results show that, the overhead of thread context switch around 4 times less then process context switch. The main difference between thread and process context switch is that, when thread context switching, kernel need neither to construct the virtual memory space to the thread switch to, nor to flush the tlb, which means that thread context switch is much less expensive then process context switch.

4 Memory

5 Networking

6 File System