Performance Report on EC2 Instances

Vineet Kumar, Phuc Xuan Nguyen

October 25, 2011

1 Introduction

Amazon Elastic Compute Cloud(EC2) is a web service that provides resizable computing power through the cloud. There are lots of disputes online regarding the correctness of the "rated" hardware performance of Amazon EC2. Some articles claims that the actual performance of the hardware is much less than the advertised number. We are interested in testing to see whether such claim is accurate, and if not, what the reason might be. As a later part of the project, we are also interested in the overhead of the linux containers(LXC) on a virtualized platform.

The aim of this project is to measure performance on Amazon's EC2 instances. For the first portion of this project we measure the overhead of CPU, scheduling and OS services. All the code for our tests was written in C. We used gcc 4.6.1 with no optimizations to run our code. Collecting machine description information, measuring procedure call overhead and measurment overhead were done by Phuc Xuan Nguyen. Measurement of system call overhead, process creation, kernel thread creation, process context switch and kernel context switch were done by Vineet Kumar. We think we spent in total of around 7 working days for this portion of the project.

2 Machine Description

We are aiming to measure the performance on the Amazon's t1.micro instances.

- 1 Elastic Computing Unit
- Processor: Intel(c) Xeon(R) CPU E5430 @ 2.66Ghz.
  - 12M L2 Cache, 1333 Mhz FSB
- Memory: 592MiB
- Netword card speed
  - Between EC2 Instances: 100MB/s

- Disk: Amazon Elastic Block (EBS)
  - Size: 7.9GB
- Operating System: Ubuntu Oneric 11.10

3 CPU Operation

For all our experiments we use RDTSC counter. To obtain time we divide this by the CPU frequency. Also of all the data seen, we discard the best and worst 10% of data and then take mean values.

3.1 Measurement overhead

3.1.1 Experiment

We are using RDTSC as a fine-grained counter to measure the performance. In order to calculate the overhead of RDSTC, we run the following experiment.

Function 1:

- Get initial clock counter
- Repeat N times:
  - Run RDTSC
  - Perform a random function f
- Return the difference between the current and the initial clock counter.

Function 2:

- Get initial clock counter
- Repeat N times:
  - Perform a random function f
- Return the difference between the current and the initial clock counter

To measure the for loop overhead, we set up two procedures: (1) using a for loop to execute a function foo() 50 times, and (2) just execute foo() 50 times in a row. Through experiment we find that the amount of times does not affect the interested results.

Table 1: System call overhead

System Call	Time - cached(μs)	Std. dev- cached	Time - uncached(μs)	Std. dev - uncached
getpid()	0.00358	1.71%	7.89	5.74%
write to devnull	0.00341	1.06%	1.45	53%

3.1.2 Results

We find that the variance becomes insignificant when N is around 10000. We avoid the possible compiler optimization by running the random function f.

We calculate the difference in the result of Function 2 and Function 1 and divide that by N to find the overhead of RDTSC. In the t1.micro instance.

- Without RDTSC: average 6.00 cycles ~ 2.25 μs
- With RDTSC: average 48 cycles ~ 18.04 μs
- The overhead of RDSTC: ~ 15.789μs

After running the for-loop experiment 10000 times, we find that

- Without for loop: average 12.6 cycles ~ 4.7μs
- With for loop: average 21.8 cycles ~ 7.89μs

So the overhead of the for loop is the difference between these, which is around 9.2 cycles ~ 3.38μs.

3.2 Procedure call overhead

To find out the procedure call overhead, we perform two simple operations (int x = 1+1; int y = x;) in 9 different scenarios: no procedure call and procedure calls with the 0-7 parameters. Figure ? describes the increment in overhead.

The result, gathered after running 1,000,000 iterations, is shown in Figure 1.

3.3 System call overhead

To measure system call overhead, we need to do measurements on a system call that does not do much work. We do our experiments by calling "getpid()" and by writing one byte to the device devnull. We notice that for both these experiments if we run a tight loop within a single process, the system call gets cached and thus does not give us correct overhead measurements. Thus, we handle this issue by running the test within a context of different process. We run the test for 10,000 iterations. Table 1 shows the results:

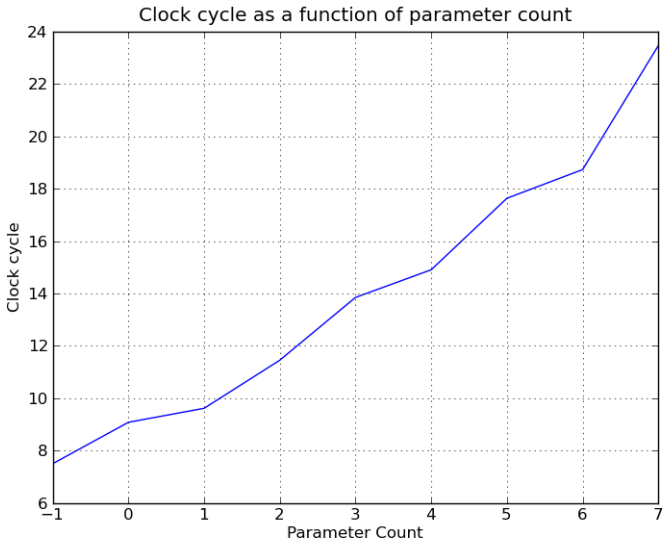

Figure 1: Clock cycle(param)* -1 means no procedure call

Table 2: Process and kernel thread creation overhead

	Time(μs)	Std. deviation
Process creation	264.73	18.48%
Kernel Thread Creation	1.89	35

3.4 Task creation time

We measure the task creation time by calling the timer before a fork() is issued and immediately inside the child process. We repeat this process for 10, 000 iterations. To measure the creation time for a kernel thread - we use posix thread attributes to tie a user thread to a kernel level thread. We repeat these experiments 10,000 times. We use the following test methodology :

1. Repeat N times
2. Start timer
3. Fork a process
4. Stop timer inside the child process

3.5 Context switching time

We measure context switch time by passing a token across pipes. We create a total of 4 pipes to accomplish a 2 way communication and measure the round

Table 3: Context Switching overhead

	Time (μs)	Std. deviation
Process Context Switch	2.19	27.8%
Kernel thread context switch	2.65	38.5%

trip time. This roundtrip time per process contains time needed to conext switch twice and time for a read and write call using pipes. We however, do not remove the pipe overhead - that needs to be done. We run our experiments for N iterations

This is the test methodology we use for measuring a process context switch time

1. Create 2 pipes for communication between process 1 and process 2 (pipe1)
2. Create 2 pipes for communication between process 2 and process 1 (pipe2)
3. Repeat N times
   (a) Start timer
   (b) Repeat (c) -(d) for some iterations
   (c) Process 1 writes to pipe1, process 2 reads it. Note that these will be blocking reads. This causes Process 2 to start running
   (d) Process 2 writes to pipe2 and process 1 reads it. This will again cause a context switch
   (e) Stop timer

For Measuring kernel thread context switch time we use posix threads bound to kernel by setting scope attribute to PTHREAD_SCOPE_SYSTEM