

Measuring Context Switches in Serverless Environments

Elaine Yao
elainedv111@gmail.com

1 Introduction

Serverless computing is a new type of cloud application model. Instead of deploying virtual machines or containers, the developers only need to write functions without considering resource allocation, scheduling, etc. This simplifies the process of deploying code into production. Some commercial offerings include AWS lambda, Azure serverless and Google Cloud function. The cloud provider will allocate resources on demand and the pricing model of serverless platforms is different. It will bill the customers for the invocations times, functions execution time and the outbound network request. For example, in Google Cloud Function, when the amount of invocation is beyond 2 millions, for each extra million of invocation, it will charge \$0.4. For each gigabyte of outbound network, it will charge \$0.12. As for the execution time, the price is based on the amount memory and CPU resources provisioned for the function. In Table 1 For example, with 128 MB memory allocated and 0.2 GHz CPU frequency, the price for each second of execution time is \$0.00000231. And for \$1, the function deployed on cloud can run at most for a week. As the memory doubles, so does the price and CPU frequency. With more memory, the function will occupy more CPU resources. For exchange, the user has to pay more.

The execution time is measured from the time that the function receive a request, to the time it completes. It may complete through signaling, timeout or other failures. The problem is, user's functions can't be executed continuously due to limited resource. Providers want to invoke more functions at the same hardware to achieve the low resource cost. When resources like memory or CPU is limited, a running function may be interrupted to let another function being executed. Some CPU cycles are spent in OS scheduling, container management, page fault handling, etc. Thus, the function execution time(CPU wall-clock time) is larger than the actual execution time, which is presented in previous studies. [8] For users, who are charged with the function execution time, this phenomenon makes them charged more than they should be. In this project, we'll focus on the context switch in server,

which is one of the main extra time contributor [8].

Context switch [4] refers to the situation when the operating system interrupts the current execution and switch it off to another task in multithreading/processing. It happens when the number of processes/threads is more than the amount of CPU cores. In context switch, the OS has to save context registers for the currently-running process into the kernel stack, and then restore the context registers for the soon-to-be-executing process from kernel stack [2]. In this way, when the OS wants to resume the execution of previous process, it can just fetch the stored context registers.

There are various studies [3, 5–7] proposing benchmarks for measuring the context switch time in Linux systems. Most of them are creating pipes among different threads or processes and using system call to force the context switches. However, these benchmarks written in C can't be directly used in serverless environment as the latter only supports languages like Python, Go, PHP, Node.js, Java, etc. Also, there are other factors that may influence the number of the context switch and the context switch time in serverless environment, for example, the memory configurations. The characteristics of the context switch in the cloud are different from traditional Linux systems and thus the current benchmarks can't be used directly.

Have a sense of the context switch time in serverless environment can help user know about the extra cost they're paying for and may also motivate cloud providers to formulate more mature pricing model. In this work, we aim to answer the following research questions:

1. How to measure the context switch time accurately in a serverless environment?
2. What's the factors that may influence the number and time of the context switch in a serverless environment?

Memory	CPU frequency	Price/s	Time/\$
128 MB	0.2GHz	\$0.00000231	120h
256 MB	0.4GHz	\$0.00000463	60h
512 MB	0.8GHz	\$0.00000925	30h
1 G	1.4GHz	\$0.00001650	16.8h
2 G	2.4GHz	\$0.00002900	9.5h

Table 1: Context switch time by different benchmarks

2 Methodology

2.1 Challenges

Measuring the context switch time in serverless functions has several challenges:

C1 Characteristics of context switch in serverless environments

Context switch is triggered differently in serverless functions compared to in traditional operating systems as it's scheduled by the cloud provider. And the service provider has the tendency to trigger more functions in the same hardware to get more profits. Thus, context switch may happen not only due to performance factors like multi-threading/processing, but also provider's profit considerations. Besides, in Linux setting, the context switch time is related to the number of processes/threads and process/thread size. In the case of serverless functions, it may be influenced by other factors.

C2 Benchmark accuracy

As none of the existing benchmarks can be used in serverless environment, it's challenging for us to reason about the accuracy of new benchmarks proposed. We also have to reason about the potential factors that might lead to variations in the measurement.

2.2 Context switches in serverless environment

In a cloud setting, the cloud provider will allocate multiple users to share the same physical device, in order to maximize the computing resource utilization. When there are multiple users co-locating at the same server, if the number of processes needed to run is larger than the cores inside the server, it's highly likely that there will be context switches.

Figure 1 shows one example scenario. Suppose there are two cores in a physical machine, but there are three processes need to run. Each process belongs to one user. When process 1 is executing on core 1 and process 3 is executing on core 2, No resource is available for process 2. In order to handle the request from process 2 simultaneously, the scheduler may interrupt process 1 and assign process 2 to run in Core 1 by saving the context register of process 1, stopping it, and restoring the registers of process 2 and start running it. After a

while, it switches back to process 1. In this scenario, two context switches happen and add extra execution time to process 1.

To tackle C1, we design the following experiments to analyze the factors influencing context switch in serverless environments. Shahrade *et al.* [8] shows that the execution time in serverless functions is influenced by function invocation frequency, memory size and function execution time. Since invocation frequency only influences the time that the function is called and has no impact on the execution process of the function itself, we choose to observe the context switch time under different memory size and function execution time setting. Recall that in Table 1 with more memory allocated, the function will enjoy higher CPU frequency and thus more CPU computing resources, which might influence the amount of context switches during the function execution.

2.3 Combine various benchmarks

To tackle C2, we first run our own designed benchmarks on local Linux system and compare the results with published benchmarks [1, 3, 6, 7], ensuring the correctness of the proposed ones. Then we deploy the new benchmarks in serverless functions and compare the results. If one of the benchmarks is consistently higher or lower than others, then we'll explore the reasons and this can help us improve our accuracy. The prior benchmarks we are choosing are:

1. Pingpong pipes [1, 7]: In Fig. 2, two threads or processes are created and pinned to a single core. One thread writes the data to another thread through a pipe, which induces a context switch. After the data is read, the read thread transfers the data to write thread/process and this induces another context switch. The total execution time equals the read and write operation time adds two context switch time. In contrast, a single thread and two pipes are created. It passes the data through a pipe to itself. Here the total execution time only consists of the read and write operation time. By subtracting these two time and dividing the result with 2, we get the thread context switch time.

2. Condition var [1]: A shared variable between two threads is used and is protected by mutex to prevent it from being modified by two threads at the same time. A signal is passed between two threads to tell which thread has the access to write the variable. And by passing the signals, the context will switch from one thread to the other. The time for the signal changes is the context switch time.

3. Lmbench [6]: A ring of pipes is created, and a token is passed from process to process with these pipes. Also, the time of passing the token in a single process with this ring of pipes is measured. Subtraction of these two time is the total context switch time of a number of context switches.

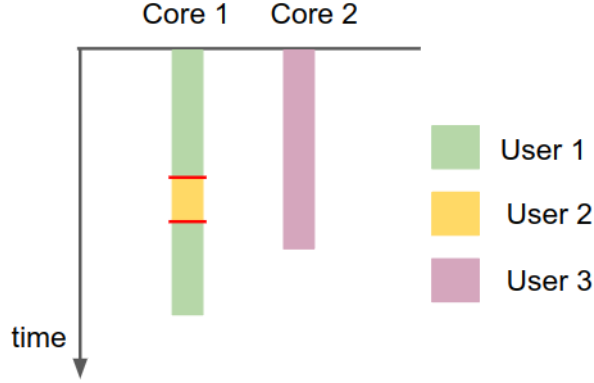


Figure 1: Context switches in serverless computing

2.4 Benchmarks in serverless environment

The aforementioned benchmarks can't be directly applied to serverless environment as they were all written in C. However, most mainstream serverless functions only support programming languages including Python, Java, Ruby and Go. In this project, we choose to write own benchmarks in Python.

Context switches can be divided into thread context switch and process context switch. **Thread context switch**

This design is similar to *Pingpong pipes* [1] in Section 2.3, the only difference that we made is we rewrote the algorithm from C language to Python.

Process context switch

This design is similar to *Lmbench* [6]. However, we add modifications to the original algorithm to make it more self-explainable. In Fig.3, a ring of processes is created and pinned to a single core. Each two of them are connected with a pipe. The parent process will pass its process id(PID) to next process. Process 2 reads the PID through the pipe, add it with its own PID, then write the new PID sum into the next pipe. For the following processes, it will read the sum of previous PIDs, add it with itself, and pass the new sum to next process. At last, the parent process will read the sum of all PIDs from the last pipe. The benefit of this design is, by checking the last value read by Process 1 with sum of PIDs, we can ensure that the value is passed through pipes. This design also enables us to analyze if the number of processes will affect the context switch time. As for the baseline, N pipes and 1 process are created to measure the read and write time. The result for subtracting T1 with T2 is the time for N context switches.

3 Experiments

3.1 Experiment setup

In RQ1, in order to check the accuracy of the benchmark proposed in this project, we first compare the context switch time measured by prior benchmarks and the proposed ones on

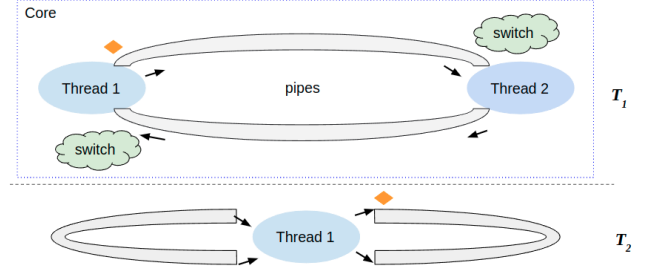


Figure 2: Measuring thread context switches with two pipes

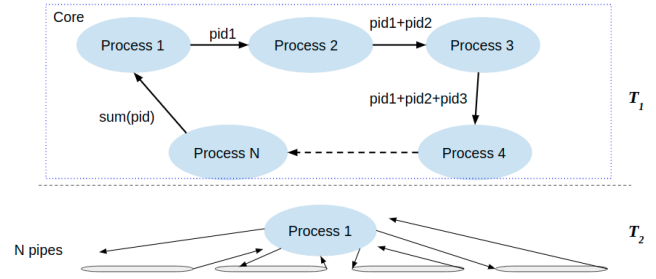


Figure 3: Measuring process context switches with ring pipes

local PC. For local measurements, we use an Ubuntu 20.04-64bit machine with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz processor and 12GB RAM. Then in RQ2, we will use the proposed benchmark to measure the context switch time in Google Cloou Function, which is a mainstream cloud platform to provide serverless computing.

3.2 RQ1: Measuring local context switches

In this research question, we want to compare our benchmarks in Python with previous benchmarks to check its correctness. For each benchmark, we run 200 times and plot the results. Fig.5 shows the context switch time measured by 5 benchmarks on local PC. *ThreadpingC* is the original *Pingpong pipes* [1, 7] written in C. *ThreadCondC* is the original *Condition var* [1] written in C. *ThreadpingP* is the thread benchmark proposed in this project written in Python. These first three benchmarks measure the thread context switch time. *Lmbench* is the *Lmbench* [6] written in C. *ProcRing* is the process benchmark proposed in this project written in Python. The last two measure the process context switch time.

The first two bars show that the thread context switch time measured by benchmarks in C language is around 2.3/2.4 microseconds and the coefficient of variation is very low. The time measured by the benchmarks written in Python, however, is almost 15-20x longer than C programs. This might due to the following reasons. 1) Python's execution speed is slower than compiled language like C, as Python is interpreted at runtime instead of being compiled to native code at compile time. 2) During the slow execution of Python, there might be

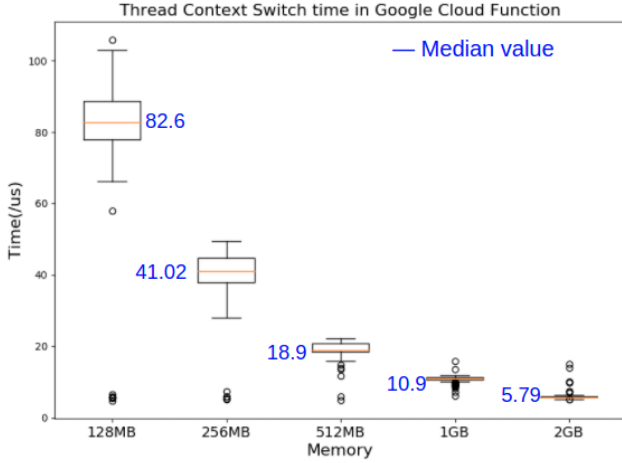


Figure 4: Thread context switches time in Google Cloud Function

more than 2 context switches happening.

As for the first reason,

This may make Python slower than compiled languages like C. Also, there is comparatively large variance in third benchmark. It may due to programming languages itself or some code modifications during my implementation in a new language. But this is acceptable as its Coefficient of variation is lower than 10

Therefore, we show the results for thread context switching for with pingpong method, condition var and process context switching with lmbench and pid sum, and perf (+pidstat/vmstat?) For lmbench, we choose xxx size. We run 100 times and get the median value and variance. And we draw them below in the figures. - bar plot, median and variance

3.3 Measuring context switches in the cloud

3.3.1 Non-parametric analysis among different memories

We first collect 200 data for each memory setting. And then we perform xxx method to check the normality. -> show the results for checking normality

Because it's not normal distribution, we then use non-parametric method to determine the number of experiment times.

We create functions with different memory allocated and measure the context switch time with the Ping pong pipes method. For each memory configuration we run 10 times and get the average of the calculated time. We also notion that the variation induced by Python language has already been considered and the value shown in Table ?? is processed. We notice that if the allocate memory is 128GB, the context switch time is remarkably higher than the other configurations. And we plan to look deeper into this in the future.

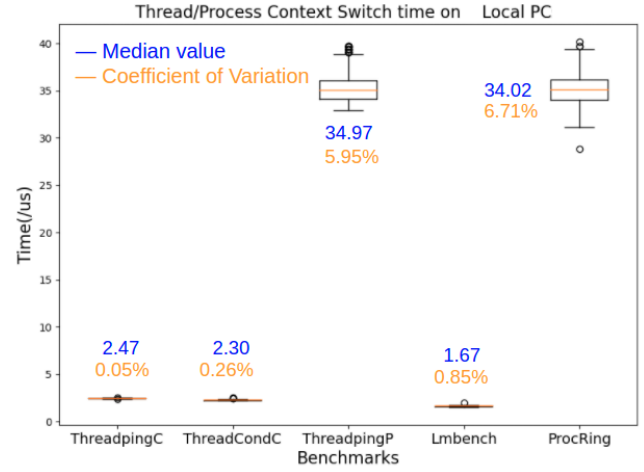


Figure 5: Thread/Process context switches time in local PCs

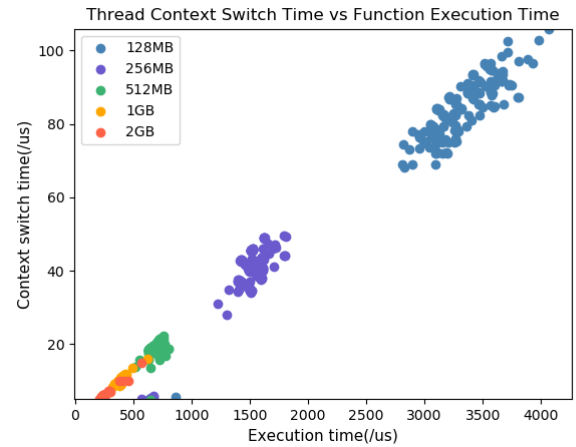


Figure 6: Thread context switch time vs Function execution time

4 Future work

We plan to look at the following problems in the future.

1. Implement condition variable and lmbench in python and deploy the function in the cloud. We aim to measure the context switches in different ways in order to improve its accuracy.
2. Explore the factors that influence the number and the time of the context switch in serverless environment.

5 Related Work

A few studies [3, 5–7] have proposed benchmarks for measuring context switch in Linux system. Ousterhout *et al.* [7] proposes using two pipes to ping pong data between two processes. McVoy *et al.* [6] reduces the cost of system calls for context switch by designing a ring of pipes, and they also consider the influence of data size and cache size on the context switch time [5]. Suo *et al.* [9] measures the context switch on the edge in AI workloads. None of them considers the context switch in a serverless environment.

References

- [1] Measuring context switching and memory overheads for linux threads. <https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/>. Accessed: 2022-2-25.
- [2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.
- [3] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Context switch overheads for linux on arm platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science, ExpCS '07*, page 3–es, New York, NY, USA, 2007. Association for Computing Machinery.
- [4] David Kalinsky. Context switch. *Embedded Systems Programming*, 14(1):94–105, 2001.
- [5] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science, ExpCS '07*, page 2–es, New York, NY, USA, 2007. Association for Computing Machinery.
- [6] Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.
- [7] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer*, 1990.
- [8] Mohammad Shahradd, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.
- [9] Kun Suo, Yong Shi, Chih-Cheng Hung, and Patrick Bobbie. *Quantifying Context Switch Overhead of Artificial Intelligence Workloads on the Cloud and Edges*, page 1182–1189. Association for Computing Machinery, New York, NY, USA, 2021.