Quantifying Context Switch Overhead of Artificial Intelligence Workloads on the Cloud and Edges

Kun Suo
Kennesaw State University
ksuo@kennesaw.edu

Yong Shi
Kennesaw State University
yshi5@kennesaw.edu

Chih-Cheng Hung
Kennesaw State University
chung1@kennesaw.edu

Patrick Bobbie
Kennesaw State University
pbobbie@kennesaw.edu

ABSTRACT

Context switching is the fundamental technique for providing flexible and efficient utilization of CPU resources in the multitasking system. However, context switching also introduces non-trivial overhead due to its complicated activities, resulting in not only significant performance degradation of applications, but also dramatic system low efficiency. In this paper, we perform a comprehensive and empirical study on the performance and overhead of context switches in modern artificial intelligence workloads, which identifies unrevealed and important impact on cloud servers and edge/IoT devices. Our observations and root cause analysis cast light on optimizing the system stack of modern operating systems to more efficiently support artificial intelligence workloads on the cloud and edge systems.

CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Software design engineering**; • **Hardware → Board- and system-level test**;

KEYWORDS

Context switch, Artificial intelligence, System, Edge, Cloud

ACM Reference Format:
Kun Suo, Yong Shi, Chih-Cheng Hung, and Patrick Bobbie. 2021. Quantifying Context Switch Overhead of Artificial, Intelligence Workloads on the Cloud and Edges. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3412841.3441993

1 INTRODUCTION

Process is one of most fundamental innovations of the operating system (OS) which allow developers to focus on the application and business logic itself instead of worrying about where to deploy their codes and how to tweak the underlying system such as CPU scheduling, memory management, I/O access, etc. In order to execute many processes, context switching is the cornerstone of the OS for multitasking. With context switching, the OS can execute as many tasks as possible on limited cores by splitting the CPU time slices and storing as well as recovering temporary data. Although many hardware context switching techniques, such as CPU gate [6], task register [7], etc., have been proposed in recent years, most modern operating systems still provide traditional software context switching to support all types of platforms from cloud servers to edge IoTs.

Context in the OS can be roughly divided into two categories: process context and interrupt context. Generally, in the user space, the process context includes application data, user stack, and shared storage. In the kernel area, the process context contains process control block (`task_struct`), memory management information (`mm_struct`, `vm_area_struct`, etc.) and kernel stack. Besides the above, data in the registers such as general registers, program registers (IP), stack pointer (ESP), etc., is also part of the process context. The interrupt context includes parameters and data passed by the hardware during interrupts. In general, there exist two types of interrupt context switches: (1) context switch of a process interrupted by interrupts; (2) low-level interrupt handlers interrupted by high-level interrupts. In operating systems, the process context switch and first kind of interrupt context switch are widely existed while the second are less common.

Despite the conveniences and benefits, context switch, which is one of the most time-consuming operations in the OS, also introduces additional overhead and inefficiency to the entire system. Recent studies [8, 9] reported that mitigating context switch overhead can achieve 4.7 times speedup in workloads such as image processing and reduce the CPU utilization by up to a factor of 30. For inefficiency of many compute-bound workloads such as artificial intelligence (AI) applications, the expensive process switching as well as the frequent data saving and restoring can be easily identified as the main culprit. Indeed, as an example of image recognition using Inception-V3 [40] on the Raspberry Pi 4B, it would introduce up to 4500 process switches each second, leading to high system cost and long execution latency. In this paper, our investigation reveals that the causes of context switch overhead and the low-efficiency of context switch in AI workloads are multifaceted and even more complicated:

First, the traditional high-performance chips (e.g., CPU) or customized high-speed accelerators (e.g., GPU, ASIC, FPGA, etc.) require the OS kernel to quickly process the AI algorithms or models.

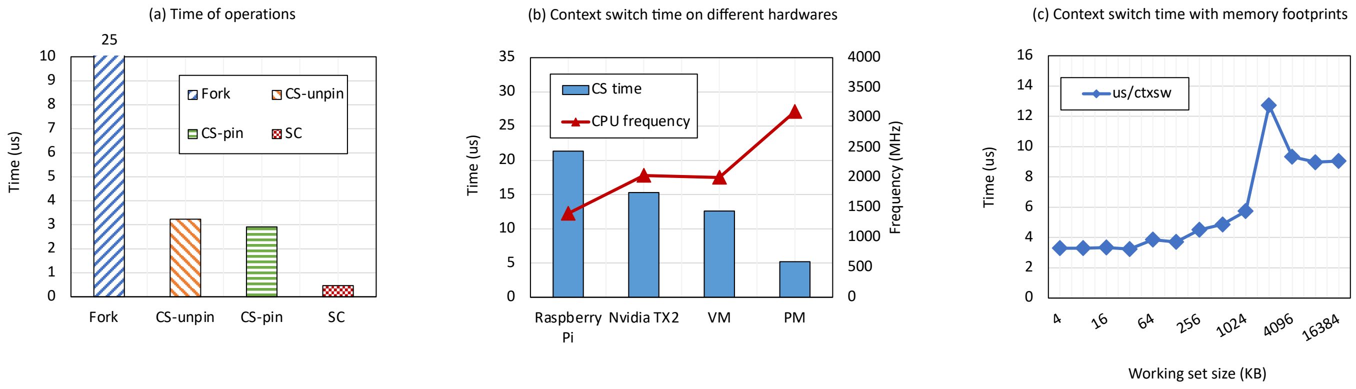

Figure 1: Context switch overhead measurement. Here 'CS' denotes context switches and 'SC' denotes system calls.

However, as stated above, context switches would slow down the application execution with significant overhead involved. More critically, we measured the cost of context switch in our controlled testbed and have found many factors including hardware platform, CPU frequency, working set data size, number of processes, etc., contributed to the overall context switch overhead. Therefore, the maximum application performance is highly limited by the characteristics of underlying system stacks.

Next, we measured the context switches and system utilization on traditional cloud servers and general edge platforms such as Raspberry Pi. We observed that the application data size and neural network models are closely related to the process switches. Further, although most modern hardwares provide multi-core processor to separate CPU cores for parallel processing. Unfortunately, as AI workloads are computation intensive and produce poor scalability, the overall application performance is still limited by the processing capability of a single CPU core and largely affected by the context switch overhead. For instance, we observed that the CPU utilization of the image processing using Nvidia SPADE [30] would drop by up to 40% when process switching goes from 2500 per second to 4000 per second.

Lastly, our research also evaluated context switches overhead and application performance on heterogeneous cloud and edge architectures. The combination of CPUs and accelerators (GPUs, TPUs, ASICs, etc.) allows many AI workloads to be processed on separate devices for parallel processing. Unfortunately, context switch is still observed to produce more overhead with accelerators. For instance, object segmentation using ICNet [42] on Nvidia Jetson TX2 introduces an order of magnitude higher than that on Raspberry Pi. Our investigation finds that this severe issue is largely due to the inefficient interplay and interrupt between the CPUs and hardware accelerators.

To summarize, this paper made the following contributions:

- We perform a comprehensive and empirical performance study of context switch in AI workloads and identify the above-stated new, critical bottlenecks on the cloud and edge platforms.
- We further deconstruct these bottlenecks to locate their root causes. To the best of our knowledge, this is the first research to study the context switch overhead impact on AI workloads in the modern platforms.
- We believe our observations, insights and root cause analysis will cast light on optimizing the OS kernel to well support future AI workloads on cloud and edge systems.

The rest of this paper is organized as follows. Section 2 introduces the background of context switch overhead, analyzes the root causes of context switch overhead, and reviews the related work. Section 3 presents experimental results and analysis of context switch overhead in AI workloads on cloud servers and edge IoTs, respectively. Section 4 concludes this paper and proposes our insight.

2 BACKGROUND & RELATED WORKS

In this section, we introduce the background of context switch processing (under Linux), analyze its potential overhead and discuss the existing optimizations for context switching.

2.1 What Does the Context Switch Overhead Include?

During the context switching, the CPUs will frequently execute between the registers and process runqueues instead of running the real working processes. So what does the overhead of process context switching include in the operating systems? Generally, the direct overhead contains global page table directory switching, kernel stack switching, and hardware context switching such as loading data into the registers before the process resumes. For instance, the base pointer (bp) and stack pointer (sp) register need be updated to load the bottom and top stack address of the process being executed. In addition, the direct cost also contains the translation lookaside buffer (TLB) refresh and the scheduler execution time. In comparison, the indirect cost of context switches mainly lies in the shared data between multi-core caches. After switching to a new process, the CPU speed will be slower as all levels of caches are not hot. The indirect cost would be slight when switching on a single CPU. For context switching on multiple CPUs, all data in L1, L2, and L3 cache have changed and the memory I/O would be delayed significantly.

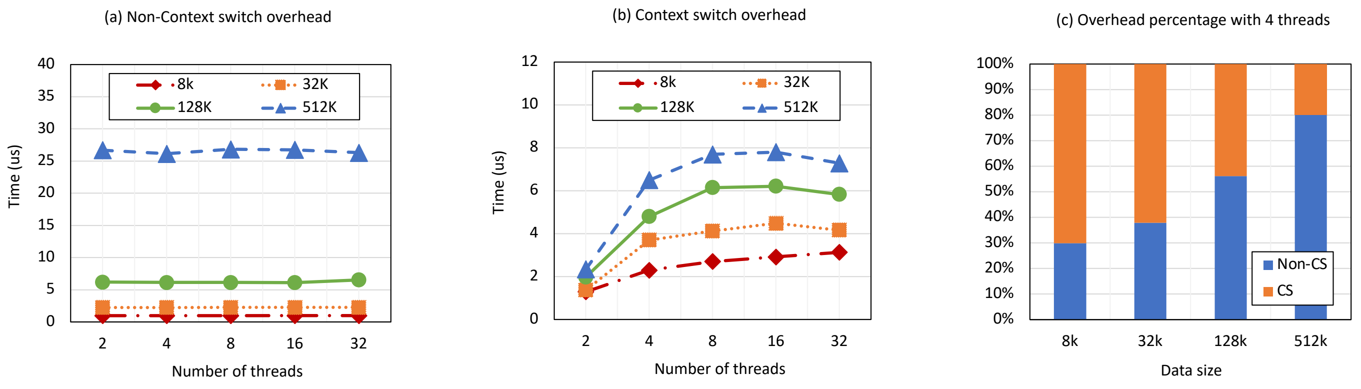

Figure 2: Context switch and non-context switch overhead evaluation using lmbench.

2.2 How Much is the Overhead of Context Switch?

Here we analyzed the potential impact of context switch using different methods. Details of the testbed and application settings can be found in Section 3. First, we measured the CPU time cost of one context switch using pipe. We created two processes in Linux and passed a token between them. The receiver process is blocked when reading the token while the sender process is blocked after sending the token and waiting for the response. The token is transferred a certain amount of times in this way and we calculate its average single switch time when the program finishes. To make the evaluation accurate and consistent, we manually set the processes with maximum priority under real-time scheduling policy `SCHED_FIFO`. After thousands of such communications, the average time for each context switch on our testbed is about 3.5µs. We also tried to measure the overhead by pinning the sender and receiver process on the same CPU core. As Figure 1(a) shows, the overhead under CPU pin decreased by around 10%. This is due to the fact that the CPU migrations and inefficient cache reuse might introduce extra cost. In addition, we further compared the overhead of context switches with other kernel activities such as system calls. Here we created a file with 3MB fixed size and executed the read system call. We found that many system calls' overhead was far less than the context switch. The system call only switches between the user mode and kernel mode of one process while the context switch is directly switched from one process to another which introduces more overhead. Besides, we also compared context switch cost with the `fork`, one of the most expensive operations in the OS which spends bulk of time copying file descriptors and VM map entries. As shown in Figure 1(a), although it is far less than `fork` cost, the context switch overhead is still significant due to its high volume and cannot be neglected in system design and service deployment.

Next, we evaluated the overhead of context switch using micro-benchmark *contextswitch* [2] on various different hardware platforms. Here we select *timectxsw* to evaluate the cost of context switching between two processes. On the edge or IoT devices, we adopted Raspberry Pi 4B and Nvidia TX2. As depicted in Figure 1(b), the context switch time on Nvidia Jetson TX2 is 28.3% less than that on Raspberry Pi. With faster CPU speed, the delay introduced by context switch could be greatly reduced. In addition, we also measured the context switch time on cloud servers. As Figure 1(b) shows, the context switch overhead on VMs is 2.4× of that on physical machines even though the CPU frequency of physical nodes is just 50% faster. The page table in VMs usually cannot be updated by the guest OS but through hypervisors, which generate additional context switches during the VM execution. Another observation from Figure 1(b) is that the context switch overhead of VM is much less than edge devices even though the CPU frequency is similar. This is due to the VM has much larger L1 and L2 cache than the IoT hardwares which can also reduce the delay of frequent context switches. Besides, we also explore the cost of context switch under different data sizes. Here we adopted *timectxswws* to measure the overhead of context switching using various working sets on Nvidia Jetson TX2. As shown in Figure 1(c), the context switch cost is insignificant from L1 (64KB) to L2 (2MB). Instead, the time jumps to 13µs when the data size outreaches L2 cache size.

Lastly, we used tool `lat_ctx` in `lmbench` [1] to measure the entire application overhead. `lmbench` is a multi-platform open source benchmark used to evaluate the comprehensive performance of the system. The idea is to create many parent-child processes. The parent process writes data in the pipe and waits for the child process to complete the response. The child process reads data from the pipe and waits for the parent process to send data. Different from the above programs, user can precisely control the intensity of process switching and the number of processes of switching through parameters. For instance, in the specific operation of `lat_ctx`, there are two cyclic loops to control process switching strengths (the number of processes) and frequency (the amount of data exchanged between parent and child processes). As Figure 2(a) shows, non-context switching overhead is consistent with different number of processes or threads but proportional to the amount of data. This is because the main overhead of non-context switching comes from the memory copy. For context switching overhead, it grows significantly as the number of processes increases. More context switches would happen as more processes execute on a CPU. In addition, we also found context switch overhead is positively correlated with the amount of data. During the data transferring, the processes are in blocking state and they will be waked up once there is new data. When the amount of data is large, many process switches will occur during such waiting and wake-up process, which would

Table 1: Specifications of hardware platforms.

Specification	**Raspberry Pi Model 4B** (General edge architecture)	**NVIDIA Jetson TX2** (Heterogeneous edge architecture)	**Cloud virtual machine** (General cloud architecture)	**Physical server** (Heterogeneous cloud architecture)
CPU	A quad-core cortex-A72 ARM v8 64-bit SoC 1.5GHz processor	A quad-core 2.0GHz 64-bit ARM v8 A57 processor; a dual-core 2.0GHz superscalar ARM v8 Denver processor	A quad-core 2.0GHz Intel scalable processor (vCPU)	2×12-Core Intel Xeon E5-2670 v3 3.1GHz processor
GPU	No GPU	56-core 1.33 TFLOPS NVIDIA Pascal	No GPU	4x NVIDIA GM200GL Tesla M40
Cache	32 KB L1 data cache 48 KB L1 instruction cache 1MB L2 cache	64KB L1 data cache 128KB L1 instruction cache 2MB L2 cache	128KB L1 data cache 128KB L1 instruction cache 2MB L2 cache	768KB L1 data cache 768KB L1 instruction cache 6MB L2 cache
Memory	4 GB LPDDR4-2400 RAM	8 GB 128-bit LPDDR4 1866 MHz	16 GB RAM (vMemory)	504 GB DDR4 DIMMs RAM
Storage	32 GB MicroSD Card	32 GB eMMC 5.1 disk	80GB HDD	4TB HDD

increase the overall overhead. As depicted in Figure 2(c), another observation is that non-context switch overhead is dominant when the amount of data among processes is large while context switch takes the majority of overhead when the data size is small. Note that all the manual tests absolutely perform zero calculations which might have high cache hit rate in data and instruction cache. In the real world, switching between two tasks could result in higher penalties due to memory competition and pollution.

2.3 Related Works

Context Switch Cost Analysis. Much effort [12, 13, 24, 26, 28, 41] has been dedicated to analyzing factors that affect context switch cost and application performance. Li et al. [24] quantified the indirect cost of context switch by measuring the impact of program data size and access stride, and they found that the OS background interrupt handling has a significant influence on the context switch overhead. Liu et al. [27] characterized the behavior of context switch misses and reported that reordered misses occur due to the interaction between cache replacement policy and the temporal reuse pattern of an application. Liu et al. [26] presented an analytical model that reveals the mathematical relationship among the temporal reuse behavior of a thread, cache parameters, and the number of context switch misses the thread suffers from. They reported that applications with a flatter stack distance profile are more likely and vulnerable from suffering context switch misses. In industries, there also exist many manual solutions including providing dedicated resources, preemptive scheduling, prefetching the hot data, etc. Different from the above works, this paper focuses on analyzing the context switch overhead of AI workloads and their behavior differences on cloud servers and edge devices.

Optimizing Context Switch Overhead. Many researchers have revealed that poor performance of modern applications suffered from the inefficiency and complexity of context switches. Davis et al. [13] proposed integrating the differing context switch costs into fixed priority preemptive scheduling with two mixed criticality schemes. To address the compatible issues, they further invented a heuristic priority assignment policy aimed at reducing the number of high cost context switches. CSALT [28] presents a context-switch aware TLB algorithm to address the increased TLB miss rates, data cache contention and context switches. Jammula et al. [20] found that a virtual machine that switches between four workloads can cause some of the workloads a slowdown of as much as 54%. Therefore, they designed elastic time slicing algorithm which can make better decisions to minimize the context switch penalty for the most affected workloads and achieve substantial performance improvements. Many other works, including priority assignment [14], dynamic reconfiguration [22], elastic scheduling [17, 21], hardware customization [10, 45], cache prefetching [26], focus on optimizing the system execution stack and improving data processing. These studies are orthogonal and complementary to our work.

Infrastructure Efficiency on Edge AIs. Recently, a few studies start to explore how to optimize underlying infrastructure performance [19, 23, 34–39, 43, 44] to improve the efficiency for modern applications such as AI or ML workloads. Morabito *et al.* [29] proposed to consolidate IoT edge computing with lightweight virtualization including containers and unikernels. Son *et al.* [32] proposed a framework with SDN and NFV functionalities at edge environments, which greatly improve the end-to-end delay, response time, network traffic, and power consumption in different scenarios. Several other papers studies computation offloading [33], frequency scaling [11], cloud-edge assisted [25] to improve the energy efficiency, application performance and system security of edge computing. For instance, Li *et al.* [25] proposed to divide DNN model into two parts and deployed the initial layers computation on edge servers and higher layers on the cloud. DeepDecision [31] and MCDNN [15] take an optimization-based offloading approach with constraints such as network latency and bandwidth, device energy, and cost. In this work, we mainly focused on the context switch overhead of cloud and edge infrastructure for the AI workloads.

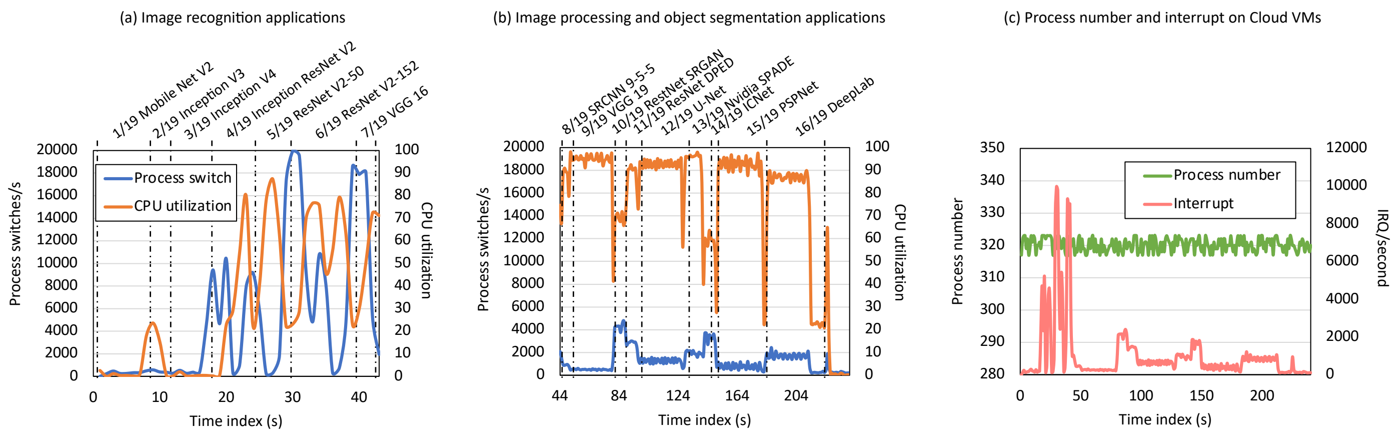

Figure 3: Process context switches, CPU utilization, process and interrupt measurement of AI workloads on cloud VMs.

Table 2: Specifications of AI workloads.

ID	Task	Neural Network / Model
1	Object Recognition	MobileNet v2
2	Classification	Inception v3
3	Classification	Inception v4
4	Facial Recognition	Inception-ResNet v2
5	Classification	ResNet-50 v2
6	Classification	ResNet-152 v2
7	Classification	VGG-16
8	Super-Resolution	VGG-19
9	Super-Resolution	ResNet-SRGAN
10	Image Deblurring	SRCNN 9-5-5
11	Image Enhancement	ResNet-DPED
12	Bokeh Simulation	U-Net
13	Semantic Image Synthesis	Nvidia-SPADE
14	Image Segmentation	ICNet
15	Image Segmentation	PSPNet
16	Image Segmentation	DeepLab v1

3 EVALUATION OF CONTEXT SWITCH OF AI WORKLOADS ON CLOUD SERVERS AND EDGE IOTS

3.1 Experimental Settings & Methodology

Hardware Platforms. We conduct our study on several representative hardwares. For general purpose edge device, we choose Raspberry Pi 4B with quad-core CPU and 4GB of RAM. For edge hardware with heterogeneous accelerators, we use Nvidia Jetson TX2, which has a quad-core A57 processor, a dual-core Denver processor, a 56-core GPU, and 8GB of RAM. To highlight the context switch impact on performance difference between edge devices and traditional servers, we evaluate the workloads on a VM in the university data center, which is configured with a quad-core vCPU and 16GB of memory. We also select a physical server equipped with 24-core Intel Xeon E5-2670 v3 CPU and 504 GB memory. All processor cores have a three level cache hierarchy that the L1 and L2 caches are private to each core, while the last level cache (LLC) is shared among all cores. We ran Ubuntu-16.04 on Linux-4.4. Table I summarizes the key architectural parameters of the hardware systems.

AI Framework. TensorFlow is developed by Google and was released in 2015, which has integrated most of the common units into the machine learning framework. Typically, to implement machine learning (ML) capabilities on devices such as mobile or embedded devices, ML models are required to be hosted on a cloud server and accessed through RESTful API services. TensorFlow Lite is an open source deep learning framework for mobile and edge device and was presented in 2017. As a successor of TF Mobile library, it provides better performance and smaller binary size due to optimized kernels, pre-fused activations and fewer dependencies. The version of TensorFlow Lite we used in this paper is 1.14.

AI Workloads. AI Benchmark [3] is an open source python library for evaluating AI performance of various hardware platforms, including CPUs, GPUs and TPUs. The benchmark relies on TensorFlow ML library, and provides a lightweight and accurate solution for assessing inference and training speed of key deep learning models. AI Benchmark is currently distributed as a Python pip package and can be executed on any operating systems. The version of AI Benchmark we are using is 0.1.2. In total, AI Benchmark consists of 42 tests and 16 sections, as shown in Table 2. The tests cover all major deep learning tasks and algorithms, and are useful for researchers, developers, hardware vendors, and end-users to evaluate AI applications on their systems and platforms.

Methodology. We execute workloads in AI Benchmark to evaluate the AI and ML performance on edge devices and cloud servers. Data from these workloads was captured using Linux nmon [4] and the output files were then converted into HTML files using nmonchart [5]. These files were further edited to highlight the duration of each test that was executed. The CPU statistics of each device were obtained and their efficiencies were analyzed. The process context switch was measured and calculated by counting number of processes that are enqueued or dequeued from the run queue on each core in every second.

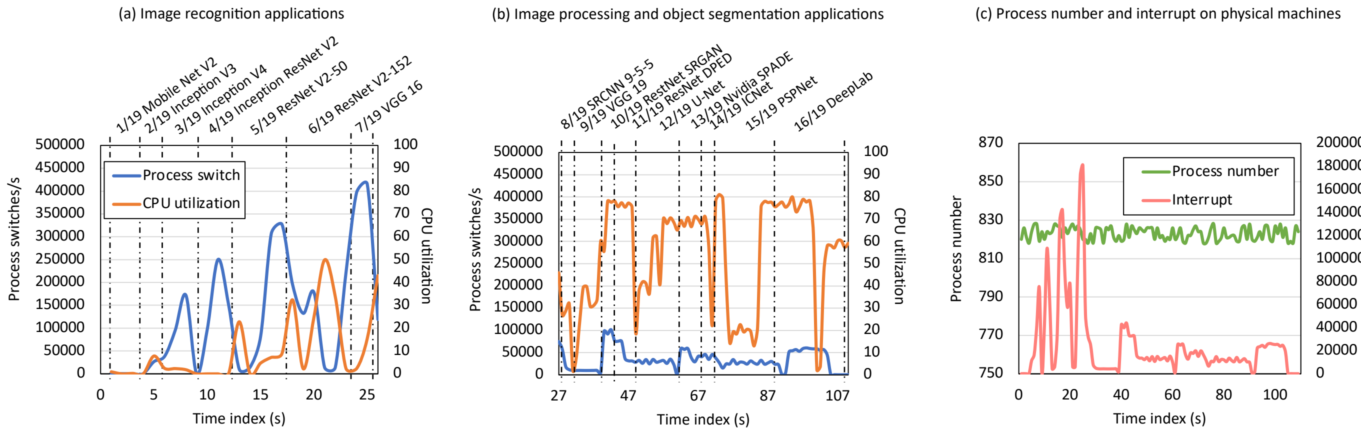

Figure 4: Process context switches, CPU utilization, process and interrupt measurement of AI workloads on physical servers.

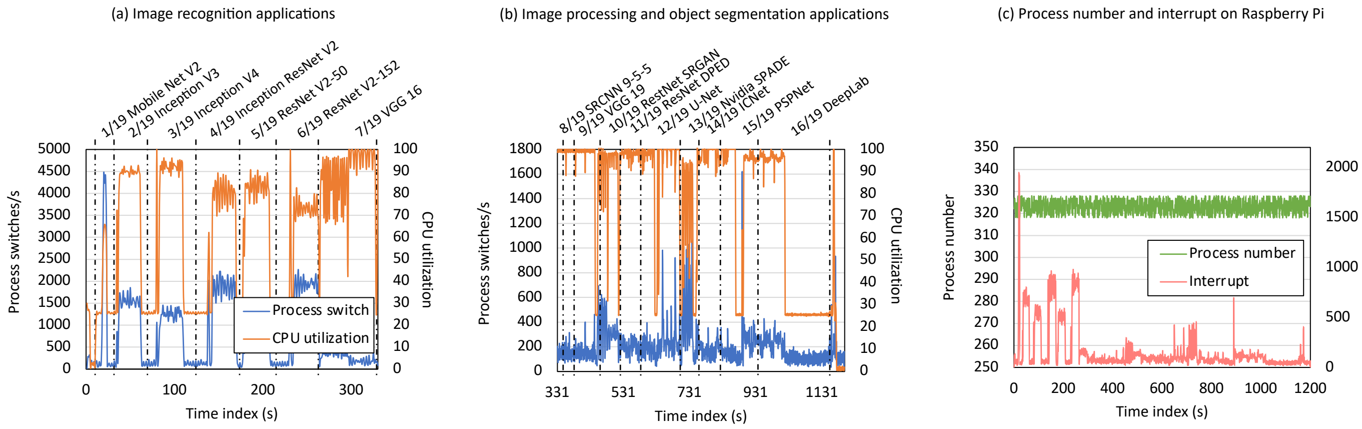

Figure 5: Process context switches, CPU utilization, process and interrupt measurement of AI workloads on Raspberry Pi.

3.2 Cloud Performance Results and Analysis

First, we measured the CPU utilization and context switches using workloads in AI benchmarks residing on traditional data center servers. Figure 3 shows the data on cloud VMs while Figure 4 shows the data on physical machines. More specifically, Figure 3(a) and 4(a) illustrate the image recognition applications which has smaller input data size from 224×224 to 346×346 while Figure 3(b) and 4(b) show the image processing and object segmentation workloads which input data size ranges from 256×256 to 1024×1024. As Figure 3(a) shows, the traditional lightweight neural networks, such as MobileNet [18], Inception [40], etc., can reach around 20k process switches each second on VMs. In comparison, as depicted in Figure 3(b), the context switches of image processing and object segmentation benchmarks reaches up to only 5k per second in VMs. From the statistics, we can conclude that the neural network model and the input data size have a significant impact on the system context switch as well as its overhead. Such the overhead cannot be simply neglected. For AI workloads introducing many context switches as shown in Figure 3(a), the CPU utilization was negatively correlated with the process switches. In other words, too many context switches introduced by more complicated models, deeper neural network depth and smaller input data could cause more overhead and reduce the entire system resource utilization. For computation-intensive workloads in Figure 3(b), even though the absolute number of context switching performed on the CPU is not high, the overhead of context switching still needs to be considered as all the work is performed on the CPU and the utilization is nearly 100% during execution, in which the overall performance is sensitive to any minor overhead.

Figure 4 shows the tracing of process context switches and CPU utilization on physical servers with accelerators (e.g., GPUs). The observed patterns are similar to those shown in Figure 3. There are two major differences between physical servers and the VMs. First, the physical servers have dedicated hardwares and less context switching overhead as illustrated in Figure 1. The contradiction between the context switch overhead and system utilization is not that remarkable compared to virtual environment. For instance, the CPU utilization of ResNet V2-152 [16] is only 50% in physical servers while it is over 80% on VM host. Second, the physical server offloads most of the computations onto the GPUs which introduces conspicuous number of interrupts during the interaction between CPUs and GPUs, as shown in Figure 4(c). The physical server can

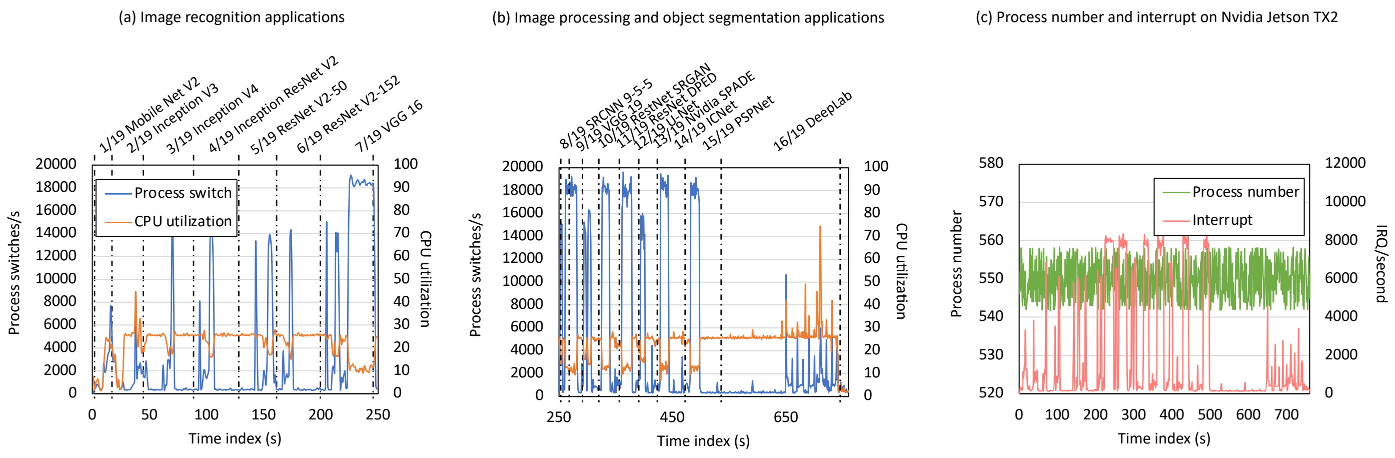

Figure 6: Process context switches, CPU utilization, process and interrupt measurement of AI workloads on Nvidia TX2.

reach 400k and 100k process switches each second for image recognition applications and image processing or object segmentation applications, respectively. Although the utilization of CPU is low, too frequent context switching could still increase entire overhead and the impact on system cannot be ignored as well. As depicted in Figure 3(c) and 4(c), another observation is that the process number does not change dramatically during the benchmark execution as most of AI workloads are functions nested and do not fork many processes, which implies most of context switches are system interrupt related.

3.3 Edge/IoT Performance Results and Analysis

Next, we investigated the potential impact of context switch overhead on edge and IoT devices. Figure 5 and Figure 6 show the process switches and CPU utilization of AI benchmarks on Raspberry Pi 4B and Nvidia Jetson TX2. As depicted in Figure 5(a), we observed that the CPU utilization of most image recognition applications increases to 100% when number of context switches goes up. As the most of AI workloads are computation-intensive and the edge platform has limited capacity of compute power, too much overhead from context switch will not only restrict the deployment of AI on IoTs, but also reduce the performance of AI workloads. For image processing and object segmentation workloads, as shown in Figure 5(b), the statistic pattern is similar with that in cloud VMs. Due to the less powerful architecture and slower CPU frequency, the process switch per second is much less on the edge platforms. However, as the context switching goes up, we can still observe the reduction of CPU utilization and performance fluctuation (e.g., Nvidia SPADE [30]).

For edge devices with accelerators (GPUs, FPGAs, TPUs, etc.), the CPU utilization will be alleviated significantly during execution while the process context switch increases dramatically at the same time. As shown in Figure 6(a), we found that the process switches each second goes 4 times in image recognition applications compared to the general edge devices shown in Figure 5. For image processing and object segmentation workloads, it increases with an order of magnitude compared to the computation with CPUs. Our investigation shows that such difference comes from lots of interrupts on the edge GPUs. During the execution, we observed thousands of `tegra186_timer0` interrupts which involves dramatic interactions between the CPUs and the accelerators, as depicted in Figure 6(c). For Nvidia Jetson TX2, `cudaMemcpy` is frequently processed for data transmission, which introduces thousands of interrupts and context switches each second. Due to its massive amounts, limited resources on the edges and dominant overhead with smaller data set, such the overhead must be handled seriously for edge/IoTs. Similar with Figure 3(c) and 4(c), the process number also does not change dramatically during the benchmark execution on the edge platforms.

4 CONCLUSIONS & INSIGHTS

We have presented the performance study of context switches on cloud and edge systems with representative AI workloads. First, we systematically studied the overhead of context switches under different system settings. We identified such the overhead are highly related with hardware platforms, application behaviors, data set sizes and number of process threads. These findings further urge us to investigate different AI workloads on data center machines and IoT devices with traditional or heterogeneous architectures. To the best of our knowledge, this paper is the first to explore the overhead of context switches in AI workloads and its footprint on the performance on cloud and edge platforms. Our analysis and findings could help the users develop more efficient applications and deploy more appropriate models on their scenarios and guide the optimization of AI workloads on various environments.

All the findings and insights inspire us to develop a more efficient system stack for AI workloads by considering the following several questions: (1) Is it feasible to eliminate the context switches overhead by adjusting the factors (e.g., application behaviors, data set size, etc.)? (2) How can the kernel perform a better isolation between different computation devices especially with heterogeneous accelerators to mitigate the context switch overhead? This is particularly important and challenging when processing massive small data inputs. (3) Can the system apply different policies flexibly, such as interrupt coalescing, multicore scheduling, etc., to mitigate context switches overhead of AI workloads with various patterns?

ACKNOWLEDGMENTS

We are grateful to our reviewers for their comments and suggestions on this paper. This research was supported by Seed Funding from Kennesaw State University.

REFERENCES

[1] 2005. *LMbench*. http://lmbench.sourceforge.net/.

[2] 2015. *Contextswitch*. https://github.com/tsuna/contextswitch.

[3] 2019. *AI benchmark*. http://ai-benchmark.com/.

[4] 2020. *nmon for Linux*. http://nmon.sourceforge.net/.

[5] 2020. *nmonchart for Linux*. http://nmon.sourceforge.net/docs/sampleC.html.

[6] 2020. *Task management*. https://www.cs.umd.edu/~hollings/cs412/s02/proj1/ia32ch7.pdf.

[7] 2020. *Task register*. https://www.scs.stanford.edu/05au-cs240c/lab/i386/s07_03.htm.

[8] N. Asmussen, M. Roitzsch, and H. Härtig. 2019. M^{3}x: Autonomous Accelerators via Context-Enabled Fast-Path Communication. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*.

[9] S. Awamoto, E. Focht, and M. Honda. 2020. Designing a Storage Software Stack for Accelerators. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[10] A. Bourge, O. Muller, and F. Rousseau. 2016. Generating Efficient Context-switch Capable Circuits through Autonomous Design Flow. In *Proceedings of ACM Transactions on Reconfigurable Technology and Systems (TRETS)*. ACM New York, NY, USA.

[11] Ying Chen, Ning Zhang, Yongchao Zhang, Xin Chen, Wen Wu, and Xuemin Sherman Shen. 2019. TOFFEE: Task offloading and frequency scaling for energy efficiency of mobile devices in mobile edge computing. In *Proceedings of IEEE Transactions on Cloud Computing (TC)*.

[12] F. David, J. Carlyle, and R. Campbell. 2007. Context Switch Overheads for Linux on ARM Platforms. In *Proceedings of the 2007 workshop on Experimental computer science*.

[13] R. Davis, S. Altmeyer, and A. Burns. 2018. Mixed Criticality Systems with Varying Context Switch Costs. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE.

[14] R. Davis, S. Altmeyer, and A. Burns. 2018. Priority Assignment in Fixed Priority Pre-emptive Systems with Varying Context Switch Costs. In *Proc. RTSS Workshop on Open Problems*. 11–12.

[15] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of ACM MobiSys*.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:cs.CV/1512.03385

[17] F. Hermenier, A. Lèbre, and J. Menaud. 2010. Cluster-wide Context Switch of Virtualized Jobs. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 658–666.

[18] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:cs.CV/1704.04861

[19] Hang Huang, Jia Rao, Song Wu, Hai Jin, Kun Suo, and Xiaofeng Wu. 2019. Adaptive resource views for containers. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[20] N. Jammula, M. Qureshi, A. Gavrilovska, and J. Kim. 2014. Balancing Context Switch Penalty and Response Time with Elastic Time Slicing. In *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 1–10.

[21] N. Jammula, M. Qureshi, A. Gavrilovska, and J. Kim. 2015. Reducing Cache-Associated Context-Switch Performance Penalty Using Elastic Time Slicing. *TECHNICAL JOURNAL* (2015), 23.

[22] T. Lee, C. Hu, L. Lai, and C. Tsai. 2010. Hardware Context-switch Methodology for Dynamically Partially Reconfigurable Systems. *Journal of Information Science and Engineering* 26, 4 (2010), 1289–1305.

[23] Jiaxin Lei, Kun Suo, Hui Lu, and Jia Rao. 2019. Tackling parallelization challenges of kernel network stack for container overlay networks. In *Proceedings of 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[24] C. Li, C. Ding, and K. Shen. 2007. Quantifying The Cost of Context Switch. In *Proceedings of the 2007 workshop on Experimental computer science*.

[25] He Li, Kaoru Ota, and Mianxiong Dong. 2018. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. In *Proceedings of IEEE Network*.

[26] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker. 2008. Characterizing and Modeling The Behavior of Context Switch Misses. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 91–101.

[27] F. Liu and Y. Solihin. 2010. Understanding the Behavior and Implications of Context Switch Misses. *ACM Transactions on Architecture and Code Optimization (TACO)* 7, 4 (2010), 1–28.

[28] Y. Marathe, N. Gulur, Jee H. Ryoo, S. Song, and L. John. 2017. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 449–462.

[29] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. 2018. Consolidate IoT edge computing with lightweight virtualization. In *Processing of IEEE Network*.

[30] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. 2019. Semantic image synthesis with spatially-adaptive normalization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2337–2346.

[31] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. 2018. Deepdecision: A mobile deep learning framework for edge video analytics. In *Proceedings of IEEE INFOCOM*.

[32] Jungmin Son, TianZhang He, and Rajkumar Buyya. 2019. CloudSimSDN-NFV: Modeling and simulation of network function virtualization and service function chaining in edge computing environments. In *Proceedings of Software: Practice and Experience*.

[33] Haijian Sun, Fuhui Zhou, and Rose Qingyang Hu. 2019. Joint offloading and computation energy efficiency maximization in a mobile edge computing system. In *Proceedings of IEEE Transactions on Vehicular Technology*.

[34] Kun Suo, Jia Rao, Luwei Cheng, and Francis CM Lau. 2016. Time capsule: Tracing packet latency across different layers in virtualized systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*.

[35] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. 2018. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*.

[36] Kun Suo, Yong Shi, Xiaohua Xu, Dazhao Cheng, and Wei Chen. 2020. Tackling Cold Start in Serverless Computing with Container Runtime Reusing. In *Proceedings of the Workshop on Network Application Integration/CoDesign (NAI)*.

[37] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. An Analysis and Empirical Study of Container Networks.. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*.

[38] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. vnettracer: Efficient and programmable packet tracing in virtualized networks. In *Proceedings of IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*.

[39] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis CM Lau. 2017. Preserving I/O prioritization in virtualized OSes. In *Proceedings of the Symposium on Cloud Computing (SoCC)*.

[40] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[41] D. Tsafrir. 2007. The Context-switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-nothing Loops). In *Proceedings of the 2007 workshop on Experimental computer science*. 4–es.

[42] Hengshuang Zhao, Xiaojuan Qi, Xiaoyong Shen, Jianping Shi, and Jiaya Jia. 2018. Icnet for real-time semantic segmentation on high-resolution images. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 405–420.

[43] Yong Zhao, Kun Suo, Luwei Cheng, and Jia Rao. 2017. Scheduler activations for interference-resilient smp virtual machine scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware)*.

[44] Yong Zhao, Kun Suo, Xiaofeng Wu, Jia Rao, Song Wu, and Hai Jin. 2019. Preemptive multi-queue fair queuing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[45] X. Zhou and P. Petrov. 2006. Rapid and Low-cost Context-switch through Embedded Processor Customization for Real-time and Control Applications. In *2006 43rd ACM/IEEE Design Automation Conference*. IEEE, 352–357.