Characterization of serverless workloads for improved performance on different CPU architectures

by

Kevin Vikas Jain

(Under the Direction of Kyu Hyung Lee)

Abstract

With advancements in cloud technology, serverless applications are gaining popularity as they relieve developers from server management, scaling, and load balancing. While Intel CPUs dominate the cloud market, AMD's CPUs rise in popularity is increasing heterogeneity in cloud servers. This diversity makes it challenging for developers to optimize performance for specific hardware. Cloud service providers lack mechanisms for automatic hardware selection to ensure maximum performance. This results in unpredictable performance differences and cost fluctuations across different machines. In my thesis, I evaluate the performance of nine serverless functions on Intel and AMD CPUs. I found the performance of an application is governed by two major factors - namely, (1) the type of optimization flags used and (2) the number of cores utilized by the running code. I validate my findings by benchmarking industry-standard tests with four different configurations. Identifying performance differences on these platforms can enable cloud providers to efficiently provision servers based on workload types, enhancing their service offerings.

Index words: [Serverless, Benchmarking, FaaS, Performance Analysis, CPU Architecture]

Characterization of serverless workloads for improved performance on different CPU architectures

by

Kevin Vikas Jain

B.S., University of Mumbai, India, 2019

A Thesis Submitted to the Graduate Faculty of the
University of Georgia in Partial Fulfillment of the Requirements for the Degree.

Master of Science

Athens, Georgia

2023

©2023
Kevin Vikas Jain
All Rights Reserved

Characterization of serverless workloads for improved performance on different CPU architectures

by

Kevin Vikas Jain

Major Professor:	Kyu Hyung Lee
Committee:	Wenwen Wang Le Guan

Electronic Version Approved:

Ron Walcott
Dean of the Graduate School
The University of Georgia
August 2023

Acknowledgments

I would like to thank my advisors, Dr. Kyu Hyung Lee and Dr. Wenwen Wang who provided me with constant support and guidance to make this research possible. With their expertise on the subject, our weekly meetings were fun and it allowed me to learn a lot throughout the whole process.

I am grateful to my parents who supported my dreams to pursue a master's degree, without them none of this would have been possible. I would also like to thank my friends who motivated me to finish my project on time and make my master's program a memorable one.

Contents

Acknowledgments **iv**

List of Figures **vi**

1 Introduction **1**

2 Background **3**
- 2.1 Cloud services 3
- 2.2 FaaS 4
- 2.3 CPU competition 9
- 2.4 Serverless Benchmarking 9

3 Literature Review **11**

4 Methodology **13**
- 4.1 Research design and strategy 13

5 Results **20**
- 5.1 Serverless Apps 20
- 5.2 Livermore Loops 25
- 5.3 SPEC-CPU 2017 26

6 Discussion **31**

7 Conclusion **32**

Bibliography **33**

List of Figures

2.1	Virtualization	3
2.2	Different types of cloud services	4
2.3	FaaS architecture	5
2.4	Creating a new function on AWS Lambda	5
2.5	Provisioning resources to a function on AWS Lambda	6
2.6	Performance of "sentiment-analysis"	7
2.7	AWS Lambda Costs	7
2.8	Hidden CPU architecture (Google Cloud Functions)	8
2.9	Memory fragmentation during container execution	9
4.1	Comparison of our setup with Intel i5 and AMD Ryzen 5	13
4.2	OpenWhisk Architecture	14
4.3	Example command to run "perf"	15
4.4	Example output by "perf"	15
4.5	List of SPEC CPU 2017 Benchmarks	18
4.6	List of LFK Loops	19
5.1	Performance of serverless functions	20
5.2	Instructions Executed per Cycle	21
5.3	Context Switches	21
5.4	CPU Migrations	22
5.5	Branch Misses	22
5.6	Page Faults	23
5.7	L1-dcache-load-misses	23
5.8	L1-icache-load-misses	24
5.9	Performance of LFK benchmark	25
5.10	SPEC-CPU Single Core O0 Optimized Benchmarks	26
5.11	SPEC-CPU Multiple Core O0 Optimized Benchmarks	27
5.12	SPEC-CPU Single Core O3 Optimized Benchmarks	27
5.13	SPEC-CPU Multiple Core O3 Optimized Benchmarks	28
5.14	SPEC-CPU 2017 Performance Overview	29

Chapter 1

Introduction

Serverless computing is a form of utility computing that simplifies the process of deploying code into production. A serverless function is a piece of business logic that is both stateless and transient and is designed to be triggered by a specific condition. It is used in conjunction with microservices and monolithic applications while allowing developers to lower costs and move their ideas to market faster while having the ability to scale to customer needs on demand. Serverless functions, that are provided by Function-As-A-Service platforms, have found many use cases, ranging from Data Analytics, CI/CD operations, file conversions, log aggregation, image/video manipulation, and supporting dynamic website content.

Previously, the market share of cloud servers was dominated by Intel CPUs, which means cloud providers used CPUs from one manufacturer for their services, although the latest trends show that AMD CPUs are rapidly gaining popularity due to their fast processors at cheaper rates [8]. Intel and AMD are continually upgrading their CPU architectures and introducing new SIMD (Single Instruction Multiple Data) instruction sets to improve the number of instructions executed per cycle [22]. This caused a rise of heterogenous hardware being integrated into the servers of cloud service providers [42]. For example, programs that use SIMD instructions, running on a recent Intel Skylake microprocessor with 512-bit wide SIMD instructions can be much faster than running on the older Intel Broadwell microprocessor with 128-bit wide SIMD instructions.

Traditionally, cloud providers improve the speed and efficiency of their services by adopting the use of Domain Specific Architectures (DSA), for example - automatically using Graphical Processing Units (GPU) for code written in CUDA and Tensor Processing Units (TPU) for running Tensorflow code. However, no such solution for choosing execution hardware for general-purpose code exists yet [27]. Furthermore, serverless platforms mask the execution environment from developers, which hinders the ability to write code that optimizes performance for the underlying hardware.

A lot of research has been conducted that is focused on benchmarking the performance of functions on proprietary and open-source serverless platforms [29], [41]. They try to understand the efficiency of serverless platforms with a wide variety of use cases. They have also probed into the implications of function chaining on the performance of industry applications [51]. Such insights have helped developers structure their applications well ensuring high performance at lower costs. However, with the growing

heterogeneity in the cloud, no research exists that tries to understand the performance differences of serverless applications on different CPU architectures.

This opens up an interesting area of research because CPUs from Intel and AMD have the same x86 ISA but have different architectures. We present our research which has the following contributions -

- Profile the behavior of 9 serverless functions on both CPUs and note any interesting trends. Each application will stress one or more aspects of a system - CPU cores, memory, cache, and type of instructions used.
- Characterize various serverless workloads based on their performance, we use "perf" to record system metrics like context switches, instructions executed per cycle, cache misses, and page faults to understand architectural differences.
- Profile the performance of industry-standard benchmarks like SPEC CPU and LFK loops to confirm that shifts in performance are consistent from executing serverless functions to executing applications on bare metal. SPEC CPU will have 34 applications and LFK loops have 24 kernels, each of which will be compiled with optimization flags enabled and disabled, both profiled while executing them on single and multiple cores of the CPU.
- Advocate methods for cloud service providers to use these profiling techniques to automatically chose the execution hardware based on code performance thereby improving quality of service or for developers to use our findings to tweak their applications to ensure high throughput.

We find that the performance of an application is governed by factors like the type of optimization flags used (applications compiled with O0, O1, O2, and O3 flags) and the number of cores utilized by the running code (single or multiple). Generally, we found that Intel outperforms AMD when applications run on a single core, although the performance shifts in favor of AMD when the same code is compiled to utilize multiple cores with SIMD instructions being used to parallelize complex arithmetic operations.

This thesis is structured as follows. Following this introduction, in Chapter 2, we dive deeper into the background of serverless applications. In Chapter 3, we review the existing literature on performance analysis of serverless apps in different environments. Chapter 4 presents our methodology, methods of data collection, and analysis. In Chapter 5, we discuss the results and explain the performance metrics. In Chapter 6 we discuss the future scope of this research. Finally, we provide a conclusion in Chapter 7.

Chapter 2

Background

2.1 Cloud services

Cloud computing has evolved through the years, providing services that cater to varying user needs. Traditionally, companies would set up their servers, connect them to the network, install and manage the required software & take care of security. This can be a challenging task and not every company might have the resources or the expertise to do that, making a huge barrier to entry. However, with the rise of virtualization technologies (fig 2.1) [18], cloud computing was born which facilitates the sharing of hardware resources among multiple users [19].

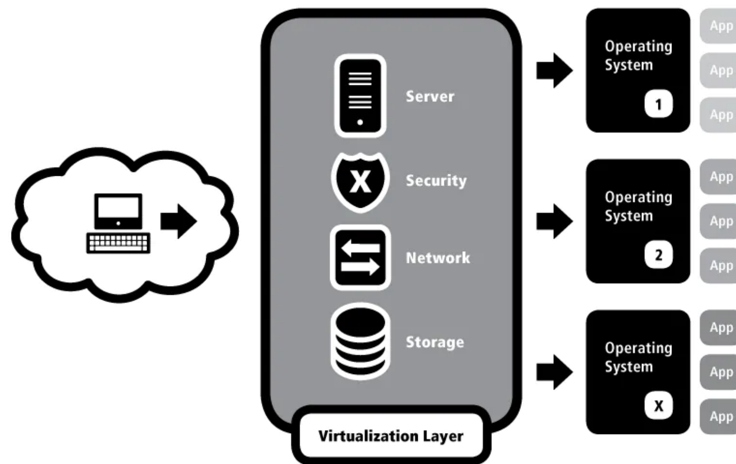


Figure 2.1: Virtualization

Since then, cloud service providers have developed various service offerings for different use cases (IaaS, PaaS, SaaS, FaaS) (fig 2.2). Infrastructure-as-a-Service (IaaS) is a pay-as-you-go model wherein a third-party vendor provides users with infrastructure services, such as storage and virtualization, via the internet [38]. Platform-as-a-Service (PaaS) enables users to upload their source code, which the platform then packages, builds, deploys, and continuously monitors for microservices, ensuring the availability of

at least one running instance [13]. Software-as-a-Service (SaaS), on the other hand, is a comprehensive service that offers entire applications managed by the provider, accessible through a website or desktop app, or via an application programming interface (API).

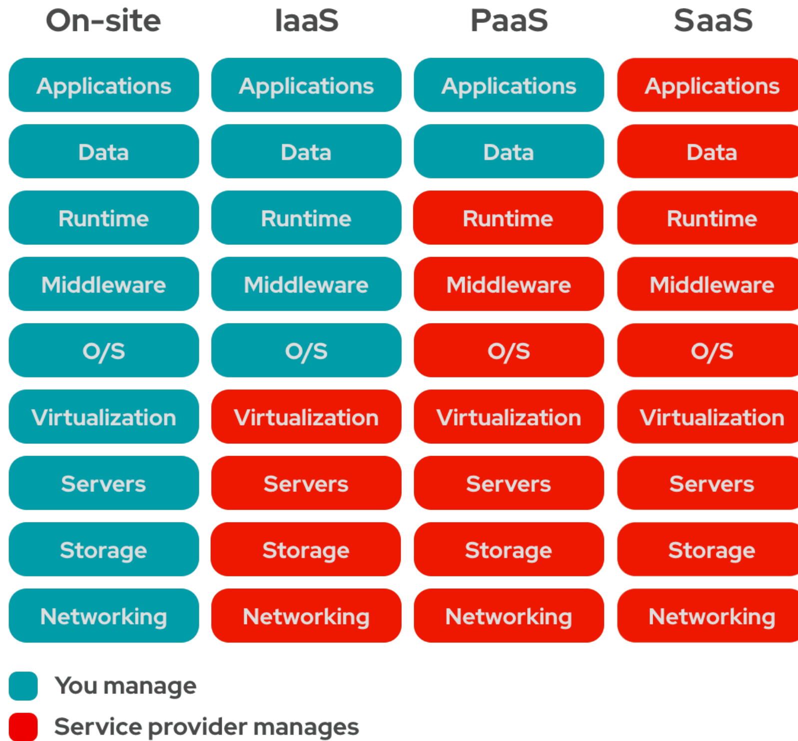


Figure 2.2: Different types of cloud services

2.2 FaaS

2.2.1 Introduction

Lastly, Function-As-A-Service (FaaS) is an event-driven cloud computing paradigm (fig 2.3) where a piece of code is executed based on an event trigger. Consumers of this service only need to focus on writing and shipping code, whereas deployment, execution, and management are taken care of by cloud providers. This design philosophy enables a separation of concerns, allowing developers to quickly scale their applications without being bogged down by the complexities of managing servers, operating systems, web services, and networking components. [20].

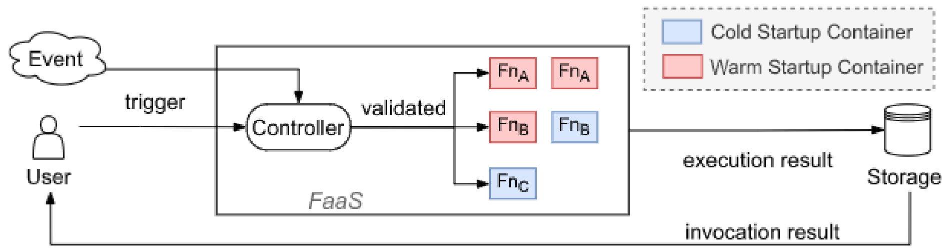


Figure 2.3: FaaS architecture

2.2.2 Creating a function on AWS Lambda

Here is an example of deploying a new function on AWS Lambda [6].

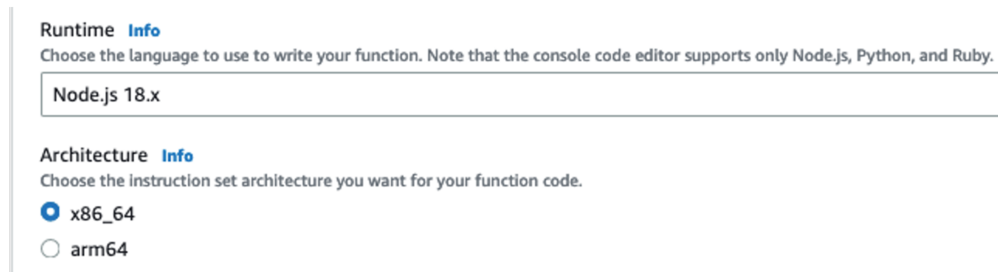


Figure 2.4: Creating a new function on AWS Lambda

Lambda encourages the use of high-level interpreted languages like **Node.js, Python & Ruby** as they don't need to be compiled, resulting in reduced startup latency. It facilitates code to be deployed on **x86_64** and **ARM** architecture machines.

Figure 2.5: Provisioning resources to a function on AWS Lambda

While using AWS Lambda, users are restricted to provisioning the memory allocation for their function, with a minimum threshold of 128 megabytes and a maximum threshold of 10 gigabytes. The amount of CPU allocated is directly proportional to the memory configuration. Lambda's self-regulating functionality enables automatic termination and subsequent restarting of the application, should it exceed the designated resource threshold. FaaS offerings are workload neutral and assume developers know their application's resource demands in advance.

2.2.3 Cost analysis of AWS Lambda

To measure the costs of running a function on AWS Lambda, we deployed a test function called **sentiment-analysis** which evaluates the sentiment of a given piece of text.

sentiment-analysis		
Memory (Mb)	task-clock (ms)	
128	414.42	
512	312.18	-25%

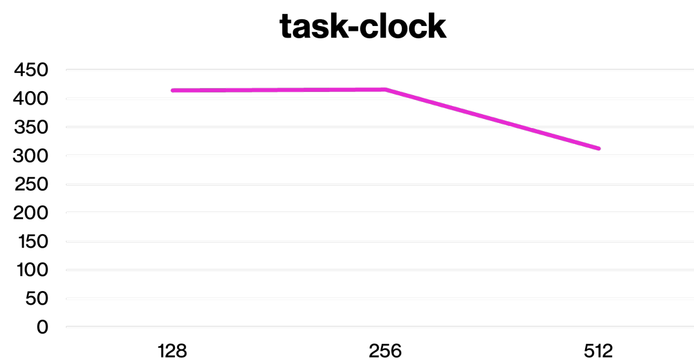


Figure 2.6: Performance of "sentiment-analysis"

sentiment-analysis		
Memory (Mb)	task-clock (ms)	Cost ($/month)
128	414.42	10.63
512	312.18	28

Figure 2.7: AWS Lambda Costs

Results indicated that increasing the allocated memory of the system yields a noteworthy **25% reduction** in execution time (fig 2.6). However, is this cost-effective?

It was found that for a **25% reduction** in execution time, developers incur **164% more** in monthly costs (fig 2.7). Serverless can get expensive and thus demands developers to know how their application performs in resource-constrained environments [12].

2.2.4 Hidden runtime environments

While deploying serverless functions, we also noticed that cloud providers often hide the details of the underlying machine that the code is running on, this is an issue as it makes it difficult for developers seeking

to optimize their code for specific CPU configurations (fig 2.8). Writing platform-independent code is a good practice but it prevents advanced developers from taking full advantage of the underlying hardware. This poses an issue as cloud service providers don't leave much room for improving code performance.

```
GCP
[
  {
  model: 'unknown',
  speed: 2778,
  times: { user: 0, nice: 0, sys: 0, idle: 0, irq: 0 }
  },
  {
  model: 'unknown',
  speed: 2778,
  times: { user: 0, nice: 0, sys: 0, idle: 0, irq: 0 }
  }
]
```

Figure 2.8: Hidden CPU architecture (Google Cloud Functions)

2.2.5 Comparing cloud service providers

In this table, we compare serverless offerings from three major cloud service providers - AWS Lambda, Google Cloud Functions, and Azure Functions. We compare configuration options available for running a serverless function and also the monthly cost incurred in running them with 128Mb and 512Mb of memory for 10 million invocations/month.

Features	AWS Lambda	Google Cloud Functions	Azure Functions
Supported Architecture	x86_64, **arm64**	x86_64	x86_64
Supported Languages	Node.js, Python, Ruby, Java, Go, Rust, .NET	Node.js, Python, Ruby, Java, Go, .NET, PHP	Node.js, Python, Java, .NET
Minimum Memory (allocated to a function)	128Mb	128Mb	128Mb
Maximum Memory	10Gb	32Gb	**1.5Gb**
Runtime Environment	Intel Xeon CPUs	**Hidden**	**Hidden**
Cost (128Mb)	$10.63	**$9.77**	$10.28
Cost (512Mb)	$28	$29.06	**$29.96**

Google and Azure functions don't support running functions for ARM64 architecture and also don't expose the details of the underlying hardware on which their code is running. Azure Functions

doesn't allow functions to consume more than 1.5Gb of memory and gets the most expensive when using their service for higher memory needs, they also don't support popular languages like Go, Rust & Ruby. Currently, the best price-for-value option for running serverless functions is AWS Lambda (in comparison).

2.2.6 Other issues

The users only have to pay for the number of resources they use, however, it comes at a cost. It takes away fine-grain control of resources and can impede building performant systems. Issues like the short lifespan of functions, stateless activities, and resource contention due to the co-location of functions (fig 2.9) can make it difficult to measure the performance of functions over time.

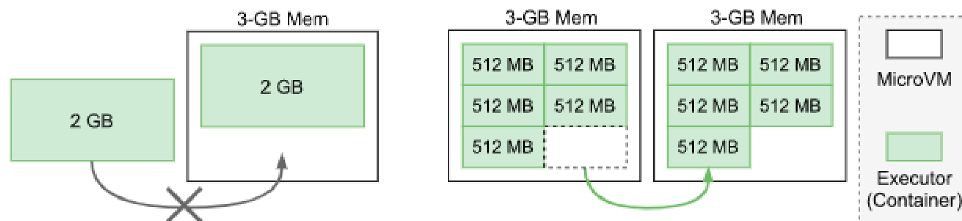


Figure 2.9: Memory fragmentation during container execution

2.3 CPU competition

Server-side computing is a $74B market which was dominated by Intel CPUs, with approximately 90% of servers containing Intel processors [21], however, AMD's latest EPYC processors are rapidly gaining traction and reported a huge increase in adoption, capturing approximately 20% of market share in 2022, thereby causing Intel's dominance fall to 70% [8]. The reason for this shift is AMD's Zen Architecture [2] and improvements in its multi-core CPU processing speeds at cheaper prices.

2.4 Serverless Benchmarking

Our study aims to evaluate the performance of serverless apps on two major competing CPUs - Intel and AMD and determine which type of workload is best suited for each. We find 9 serverless applications for performing our benchmarking. Additionally, we dig deeper by analyzing each application in terms of CPU performance metrics. We also profile the performance of official benchmarking tools.

We consider many benchmarks like Cinebench, GeekBench, PCMark10, and MemTestx86, however, we pick SPEC CPU 2017 and LFK loops for our study as they are industry standards and popular within the research community to stress test all aspects of a CPU. SPEC CPU comprises an exhaustive list of applications that perform tasks like - large code compilation, video compression, ethernet network

simulation, monte-carlo simulation, fluid dynamics, biomedical imaging, weather research forecasting, 3D rendering, nucleic acid builder, ocean modeling and computational electromagnetics just to name a few.

These benchmarking tools will allow us to make sure that our findings with serverless applications are consistent with a wide range of applications.

Chapter 3

Literature Review

Serverless computing is a relatively new paradigm in cloud computing that is exploding in popularity. It was projected to become a $7B market by 2021 [9]. It started with the introduction of AWS Lambda in 2014, since then all major cloud providers have released services supporting a similar style of deployment and operation, where developers can focus on writing code and do not worry about how their services are managed on cloud [1].

Companies have quickly adopted this new technology and reported significant reductions in operational costs which allowed them to move faster. For example, Coca-Cola reported having used serverless in their vending machines and loyalty program, saving 65% of costs at 30 million hits per month. Similarly, Abilisense reported integrating serverless into their platform which allowed them to handle the entire monthly load for only 15$ [10].

With many more companies increasingly adopting serverless technologies, researchers started coming up with many open-source alternatives like OpenWhisk, OpenFaas, and Kubeless to handle serverless functions on their cloud. This essentially allowed more transparency on how functions are managed thereby allowing developers more autonomy in choosing the right scaling strategies to fit their needs. These alternatives performed the same job but they were built with different tech stacks, different levels of flexibility, and runtime environment selections [11], [33], [48].

Researchers also started to look into the issues that came up with this new way of computing. For example, some companies expressed reluctance to break their monolithic applications into smaller functions as they fear that their growing app requirements may evolve to come in conflict with the capabilities of the platform. Famously, in a recent article by a team at Amazon Prime Video, they reported having saved 90% costs by switching from serverless to monolithic architecture [47]. This is interesting considering that AWS Lambda is the pioneer for serverless, this also indicates that not every application is suited for microservice architecture.

Every cloud provider does not support running every single programming language out there, this leads to companies having to use multiple services to fit their needs. Applications running over hybrid cloud environments tend to involve a high overhead learning curve and low flexibility for developers to improve performance. Since serverless platforms act as a black box, many researchers started to measure

different layers involved in the technological stack of serverless applications to determine any room for improvements.

FaasDom [29] is a benchmarking suite for all major FaaS cloud providers (AWS, Google, IBM, Azure) over all available regions in the world and it tries to automatically deploy different test suites and measure their performance and costs on various platforms. Although this paper provides meaningful insights into several cloud providers, it doesn't give an in-depth analysis of the performance of different applications and doesn't help paint a full picture of what is going on behind the scenes with the cloud providers' services.

Another similar research was done by ServerlessBench [51], where they analyzed different serverless platforms like AWS Lambda, OpenWhisk by IBM, and Fn. They demonstrate that breaking an application into different functions and having them interact with each other can be highly cost-saving and performant, however, the stateless behavior of functions can severely hurt the performance in many use cases. They emulate the behavior of real-world applications like Alexa skill, Image processing, and data analysis to analyze different scenarios of function chaining – namely Nested, sequential, and combined respectively. The approach helps in figuring out the best implementation of function chaining to optimize performance. However, this paper lacks insight into the internal workings of the service providers' systems.

FaasProfiler [41] tries to study the architectural implications of FaaS services. They use OpenWhisk to set up a serverless environment where users can use a CLI to deploy functions. The authors use an Intel Xeon CPU to set up a local server with enough memory capacity and network bandwidth to conduct their experiments. This allows authors to closely observe different applications at the hardware level (while having control over the available resources) and provide a much better analysis of the underlying performance.

The only fallback is that the authors only focus on one CPU architecture to benchmark performance. With the rise of the popularity of CPU manufacturers like AMD and their rapid adoption by cloud providers, it becomes pertinent to measure the performance of FaaS applications across different CPU architectures. [3] [5].

Serverless computing is still in its infancy and it exposes many opportunities for researchers to shape the future of it. Secondly, the boundaries of serverless computing dictate no control over the computing resources to developers. Many are starting to also question if these boundaries should be changed. For example, [7] questions if different service models can be mixed to create advanced solutions.

Lastly, FaaS services only allow users to provision a timeslice of CPU thread with some amount of memory, it doesn't allow users any access to specialized hardware [17]. In this thesis, we will benchmark serverless functions implemented from previous papers and report our findings in both Intel and AMD CPUs. This will help us to understand how virtualization of functions is handled by different architectures. This will also allow us to probe into possible ways for cloud service providers to automatically provision hardware based on app performance.

Chapter 4

Methodology

4.1 Research design and strategy

We set up a serverless platform like Apache OpenWhisk on two machines with Intel and AMD CPUs. We profile the execution of nine serverless functions, each of them stressing one or multiple aspects of a system (CPU, memory, cache, instruction type), and report their performance under each workload. We also perform a detailed benchmarking of the system using SPEC-CPU 2017. We use these data points to understand the performance differences between CPUs. Lastly, we discuss key findings and limitations.

4.1.1 CPU

Similarities	
ISA	x86
Cores	6
Memory	16GB
TDP	65W
Threads	12

Differences	Intel	AMD
Clock Speed	2.5GHz	**3.7GHz**
Turbo Clock	4.4GHz	**4.7GHz**
L2	**7.5MB**	3 MB
L3	18 MB	**32 MB**

Figure 4.1: Comparison of our setup with Intel i5 and AMD Ryzen 5

We use Intel® Core™ i5-12400 processor [23] and AMD Ryzen™ 5 5600X processor [4] for our experiments. Both CPUs have 6 Cores and 16 gigabytes of RAM. They are comparable in performance as per benchmarking tests published online [46] [44] [31]. Fig 4.1 highlights their similarities and differences. They both support 12 threads and x86 instruction set architecture. We configured our machine with 16GB of memory and 1TB of storage, using Ubuntu as the operating system and Docker for creating and running containers. Below is a list of all applications used and their respective versions -

Application	Version
Ubuntu	22.0 (Desktop)
Apache OpenWhisk	Latest Commit Dec 21, 2022
Docker	v20.10
Node.js	v12.22
Python	v3.10
CLANG	v14.0
GCC	v11.3.0
perf	v5.15.78

4.1.2 OpenWhisk

We move to open-source alternatives like OpenWhisk [35] and Fn [36]. They both are open-source distributed serverless platforms that can scale events to bursty workloads. OpenWhisk is developed by Apache and also used by IBM for their cloud offering. It is popular among the research community as it is built using other major open-source applications like Docker, Apache Kafka, Ansible, CouchDB, and NGINX (fig 4.2).

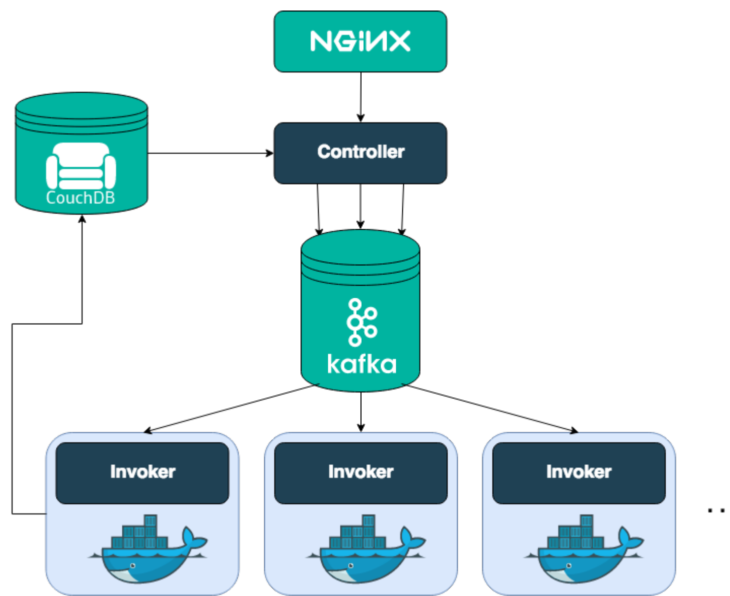


Figure 4.2: OpenWhisk Architecture

Installing OpenWhisk was not straightforward due to bugs in their installation scripts. I tried multiple methods mentioned on their website [34] and after some trial and error, these commands worked for our requirements [25].

4.1.3 "perf" Linux tool

"perf" is a popular Linux command line tool to analyze the performance data of a system, it interfaces with Performance Counters for Linux (PCL) to record and analyze a variety of hardware and software events. [39]

```
sudo perf stat
        -a                                  // Profile all CPUs
        -e <task-clock,cpu-cycles....>      // Names of events to record
        -r 100                              // # of times to run the code
        -x ","                              // CSV spec
        wsk action invoke markdown2html -i input.json
                                            // Command that we want to profile
```

Figure 4.3: Example command to run "perf"

```
Performance counter stats for 'ls':

             0.97 msec task-clock                #    0.668 CPUs utilized
                0      context-switches          #    0.000 /sec
                0      cpu-migrations            #    0.000 /sec
               92      page-faults               #   95.331 K/sec
        1,314,510      cpu_core/cycles/          #    1.362 G/sec
        1,599,678      cpu_core/instructions/    #    1.658 G/sec
          309,140      cpu_core/branches/        #  320.332 M/sec
           10,630      cpu_core/branch-misses/   #   11.015 M/sec

      0.001445243 seconds time elapsed

      0.001457000 seconds user
      0.000000000 seconds sys
```

Figure 4.4: Example output by "perf"

"perf" has many configuration options, we will use it to profile a given process running on all the CPUs, we provide a list of metrics we want to profile, and the number of times we want to run the command.

"perf" has access to many performance counters, the number of counters can depend on the system it is running on. However, for our study, we focus on the following counters that help us specifically understand the working of the underlying CPU.

List of performance counters recorded –

task-clock	instructions
cpu-cycles	iTLB-loads
branches	iTLB-load-misses
branch-misses	L1-dcache-loads
context-switches	L1-dcache-load-misses
cpu-migrations	L1-icache-load-misses
major-faults	dTLB-load-misses
page-faults	dTLB-loads

perf requires "sudo" access and major cloud providers use virtual machines on top of their hardware to isolate users from each other. This leads to restrictive access to hardware-level performance counters making it difficult for us to measure performance on the cloud. Running our experiments on a local machine allows us to closely monitor the performance and thus draw a better picture of the internal workings of serverless platforms.

4.1.4 Serverless Functions

We used nine functions that perform a wide variety of operations for our test set.

- **rsa** - an RSA key generator for a given length
- **monte-carlo** - runs Monte Carlo simulations for a given sample size and probability
- **matrix-multiply** - performs matrix multiplication on two arrays of huge sizes
- **fft** - performs Fast-Fourier Transform on a given frequency, size, and sampling rate
- **autocomplete** – a service that offers word suggestions for incomplete text inputs, with the assistance of a specified source file
- **img-resize** – resizes a given image into different sizes (hdpi,mdpi,xhdpi...) and compresses them in a single zip file
- **ocr-image** – extracts text from an image using the OCR algorithm
- **markdown-to-html** – converts a base64 uploaded markdown text into HTML
- **sentiment-analysis** – analyses sentiment of a given text

4.1.5 Serverless Test Configuration

Each application is allocated 128 megabytes of memory and 128 CPU shares. This is the minimum amount of resources required for functions to run. These are hard limits, if an app crosses these limits, it is automatically killed by the kernel by throwing an OOME, or Out Of Memory Exception [14]. So if an app is killed, developers need to increase the memory limits and restart their function. Our test apps are modified to make sure they run within these limits.

All the test functions are either written in Node.js or Python. Interpreted languages have low startup latencies as they don't need to be compiled.

We sequentially run each function 100 times and save the performance metrics during each run. We built a benchmarking script that automates this whole process [24]. Lastly, we analyze the results and present our findings.

We also notice that since applications used as serverless benchmarks are written in interpreted languages, they cannot be compiled using different optimization flags, which limits us from understanding the behaviour of same apps under different situations, for which, we then resort to industry standard benchmarks that allow more flexibility.

4.1.6 SPEC-CPU 2017 and LFK

SPEC is the Standard Performance Evaluation Corporation, a non-profit organization founded in 1988 to establish standardized performance benchmarks that are objective, meaningful, clearly defined, and readily available [43]. It consists of 43 benchmarks divided into 4 suites (fig 4.5).

Livermore loops (also known as the Livermore Fortran kernels or LFK) are a benchmark for parallel computers, it consists of 24 do-loops each of which carries out a different mathematical kernel [50]. Fig 4.6 lists all the available kernels.

We will use the benchmarks available in both suites to stress test the CPU. We will run these benchmarks stressing single and multiple cores of the CPU. And under each category, we will run the same applications compiled with two optimization flags **O0** and **O3**.

	Single Core	Multi-Core
O0	x	x
O3	x	x

As per the GCC documentation [28], applications compiled with **-O0** flags (that is the letter "O" followed by a zero) turn off optimization entirely and is the default if no -O level is specified. This reduces compilation time and can improve debugging info.

On the other hand, applications compiled with **-O3** enable optimizations that are expensive in terms of compile time and memory usage. It also enables "-ftree-vectorize" so that loops in the code get vectorized and will use AVX (Advanced Vector Instructions) registers. Along with the -O3 flag, we also append the **-march=native** flag, which tells the compiler that it may use the instructions from the native ISA

SPECrate®2017 Integer	SPECspeed®2017 Integer	Language[1]	KLOC[2]	Application Area
500.perlbench_r	600.perlbench_s	C	362	Perl interpreter
502.gcc_r	602.gcc_s	C	1,304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete Event simulation - computer network
523.xalancbmk_r	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

SPECrate®2017 Floating Point	SPECspeed®2017 Floating Point	Language[1]	KLOC[2]	Application Area
503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics: relativity
508.namd_r		C++	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging: optical tomography with finite elements
511.povray_r		C++, C	170	Ray tracing
519.lbm_r	619.lbm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++, C	1,577	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling (climate level)
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling

[1] For multi-language benchmarks, the first one listed determines library and link options (details)

[2] KLOC = line count (including comments/whitespace) for source files used in a build / 1000

Figure 4.5: List of SPEC CPU 2017 Benchmarks

(Instruction Set Architecture). On an Intel/AMD64 platform with -march=native the code will use AVX instructions to optimize performance.

We will run each benchmark under these 4 possible configurations on both the CPUs and report our findings.

Each loop carries out a different mathematical kernel . Those kernels[2] are:

- hydrodynamics fragment
- incomplete Cholesky conjugate gradient
- inner product
- banded linear systems solution
- tridiagonal linear systems solution
- general linear recurrence equations
- equation of state fragment
- alternating direction implicit integration
- integrate predictors
- difference predictors
- first sum
- first difference
- 2-D particle in a cell
- 1-D particle in a cell
- casual Fortran
- Monte Carlo search
- implicit conditional computation
- 2-D explicit hydrodynamics fragment
- general linear recurrence equations
- discrete ordinates transport
- matrix-matrix transport
- Planckian distribution
- 2-D implicit hydrodynamics fragment
- location of a first array minimum.

Figure 4.6: List of LFK Loops

Chapter 5

Results

5.1 Serverless Apps

AMD
INTEL

App	Performance Difference
sentiment-analysis	28%
autocomplete	16%
ocr-image	16%
image-resize	4%
fft	1%
monte-carlo	2%
matrix-multiply	6%
markdown-to-html	12%
rsa	82%

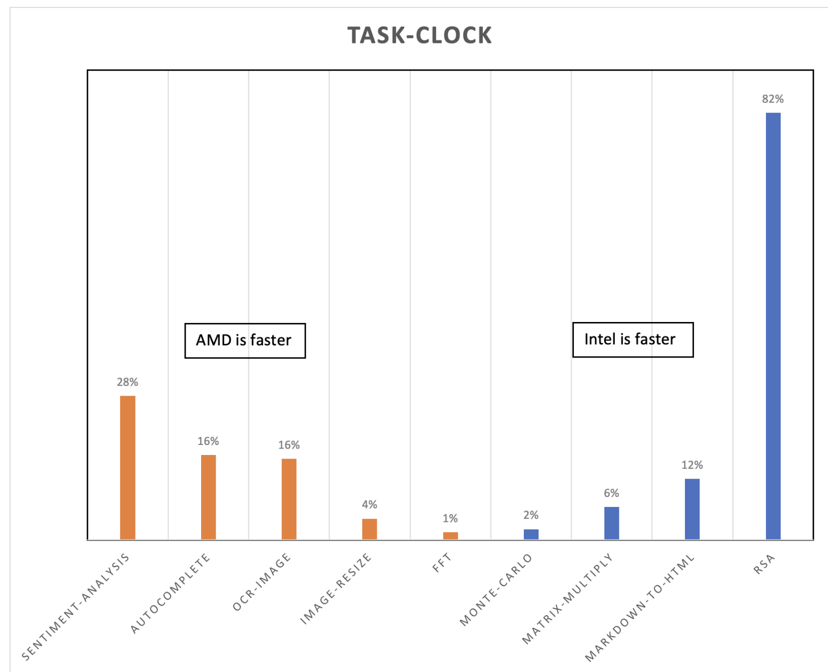


Figure 5.1: Performance of serverless functions

We test 9 serverless functions on two machines by running each function 100 times and profiling them using "perf". We calculate the average "task-clock" for each app, which represents the total duration time. We then compare the values between the two CPUs.

Fig 5.1 shows that "sentiment-analysis", "autocomplete", and "ocr-image" perform better on AMD while "markdown-to-html" and "rsa" perform better on Intel. "rsa" performs 82% faster on Intel, that is a

huge difference compared to other applications. This prompted us to dive deeper into the code and the performance metrics to understand why that happens.

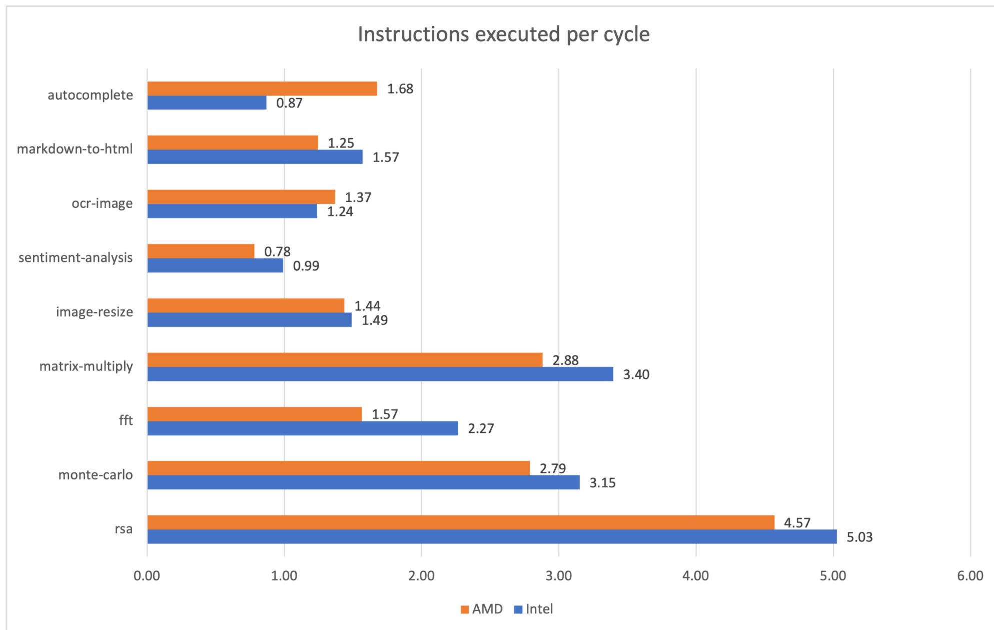


Figure 5.2: Instructions Executed per Cycle

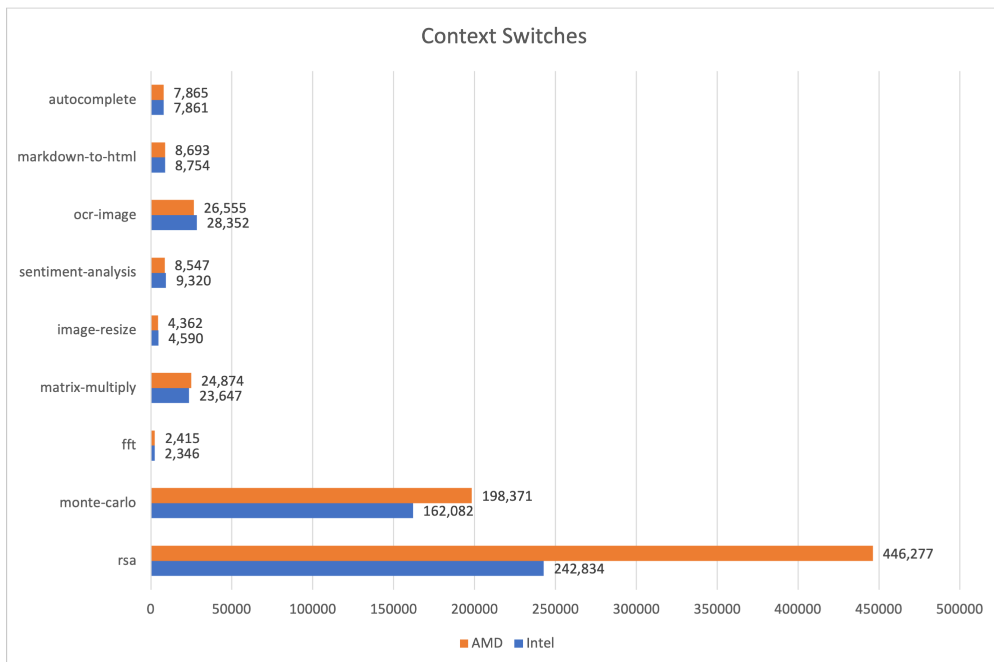


Figure 5.3: Context Switches

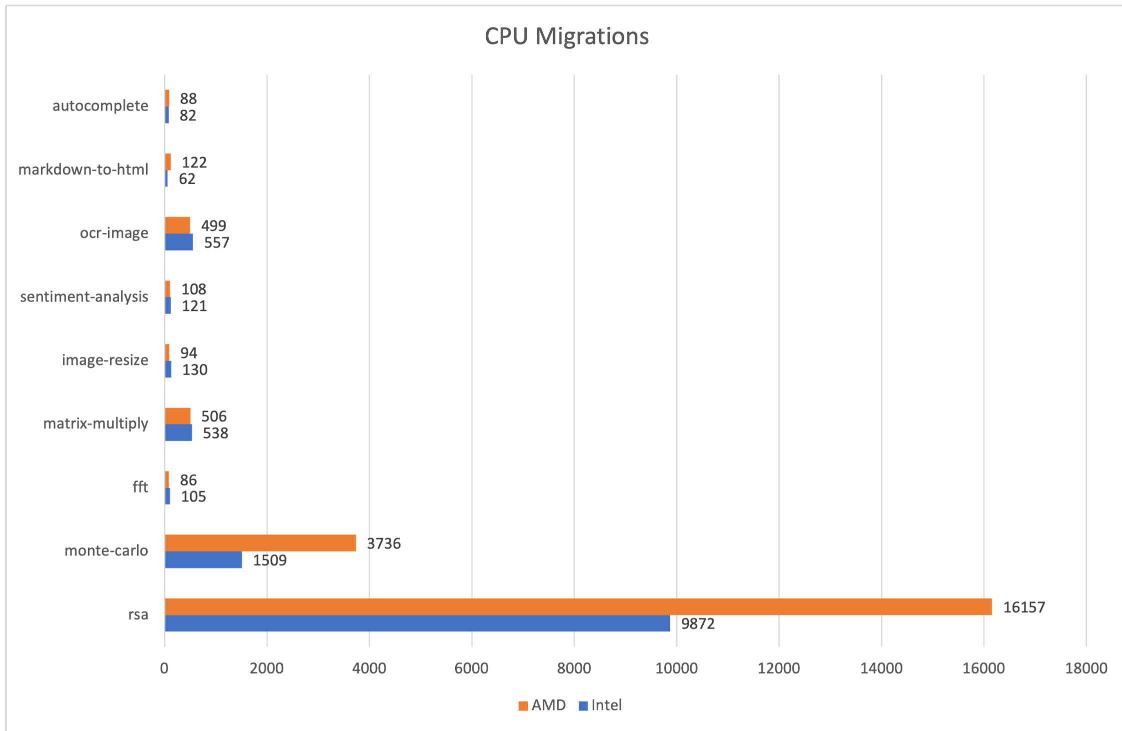


Figure 5.4: CPU Migrations

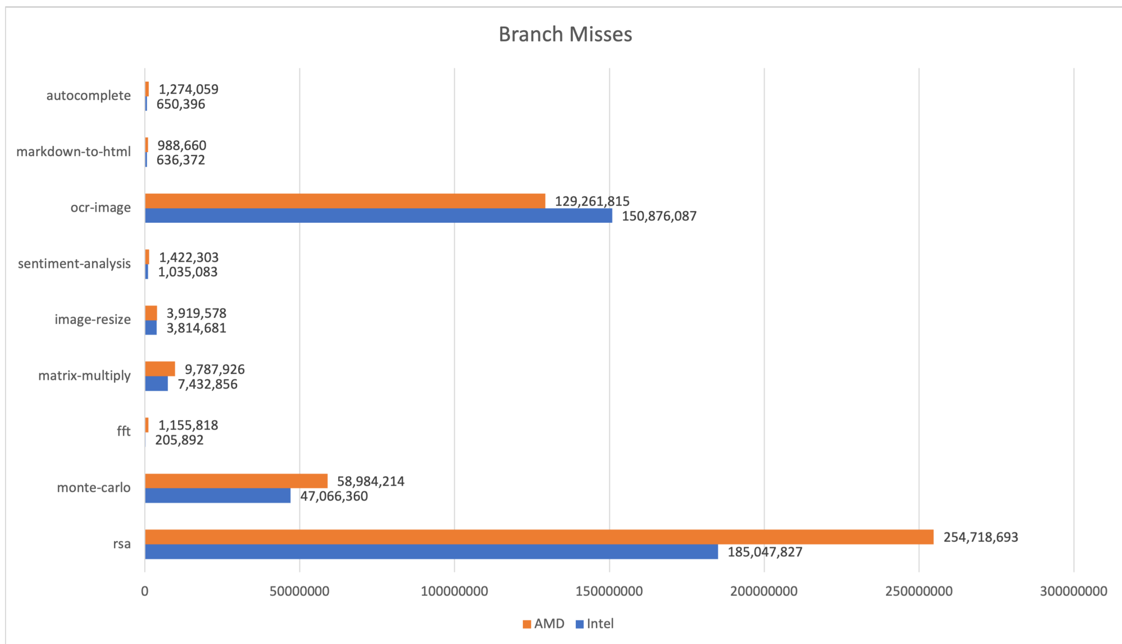


Figure 5.5: Branch Misses

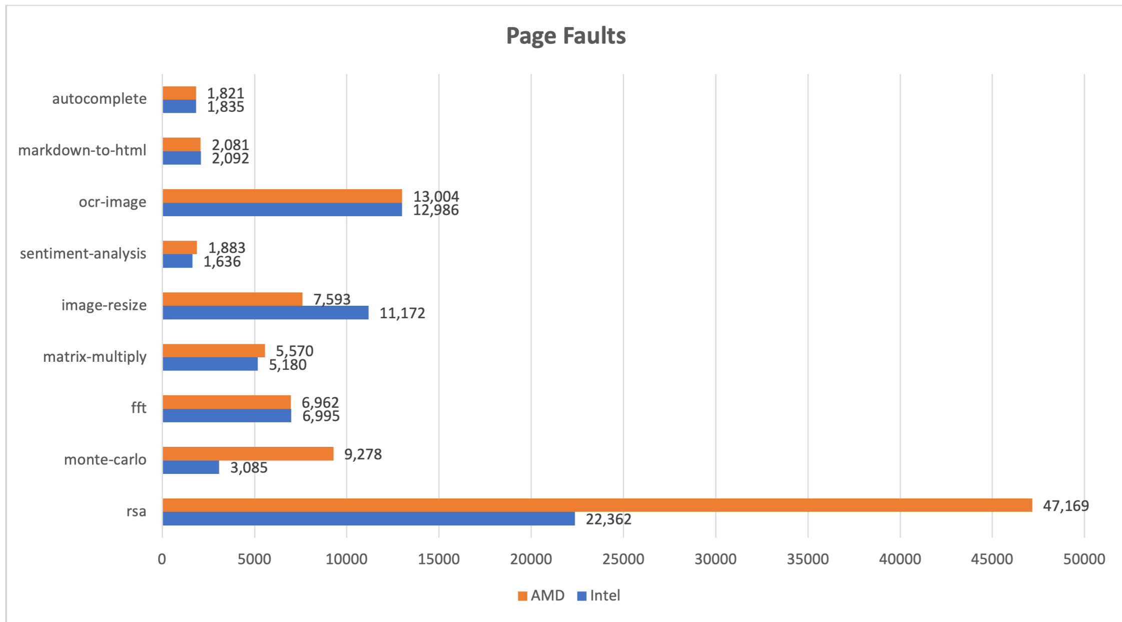


Figure 5.6: Page Faults

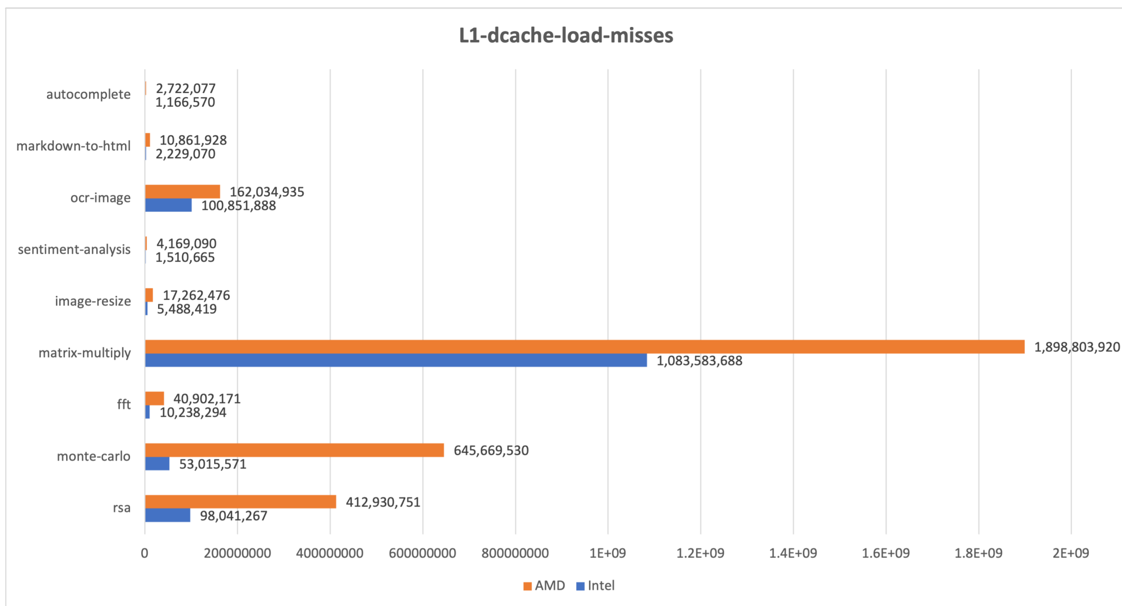


Figure 5.7: L1-dcache-load-misses

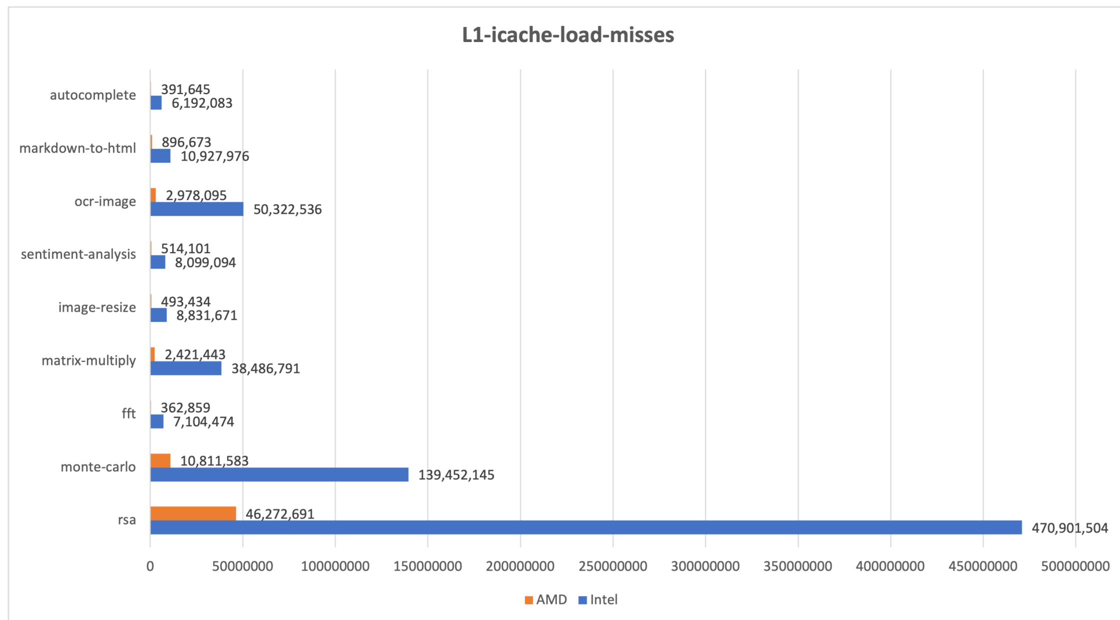


Figure 5.8: L1-icache-load-misses

On analyzing the source code for the apps, we found that "rsa" [40] utilizes only a single core of the CPU, and "ocr-image" [45] is optimized to utilize multiple cores, they were also optimized to make use of vector instructions. We also confirm these findings by monitoring the applications using "top", "docker stats" & "perf".

Fig 5.2 to Fig 5.8 displays different performance metrics. They help us understand the differences better. We will break down what these metrics mean for a few applications –

- **RSA Key Generator** (rsa) – runs 82% faster in Intel, we can see that "context-switches", "page-faults" & "cpu-migrations" are 2x higher in AMD. RSA Key generation is an arithmetic-heavy operation that relies on accessing the same data many times, we notice that "L1-dcache-load-misses" are significantly higher in AMD, which helps explain the huge performance difference.

- **OCR Image** – an application that is instruction heavy and also performs many operations on a given image, "branch-misses" are 20M more in count in Intel. We also notice that "L1-icache-load-misses" are also significantly higher in Intel, an application that contains many instructions accessed frequently, Intel will perform poorly due to heavy icache misses.

- **Matrix Multiply** – an app that frequently accesses same set of data in an array, we notice that "L1-dcache-load-misses" is 2x higher in AMD. This also explains why AMD is 6% slower for this function. We see similar patterns of high "L1-dcache-load-misses" in AMD, suggesting that AMD may perform poorly in apps that require heavy dcache access.

To further analyze performance, we will use industry-standard benchmarks with a larger set of functions.

5.2 Livermore Loops

We use the LFK benchmark available publicly [49]. There are 24 kernels that all perform floating point operations that are run on single and multiple cores in optimized and non-optimized modes built using the O3 and O0 flags.

	Performance Difference %			
	Non-Optimized (-O0 flag)		Optimized (-O3 flag)	
	Singlecore	Multicore	Singlecore	Multicore
hydrodynamics fragment	17%	2%	5%	1%
incomplete Cholesky conjugate gradient	35%	18%	19%	18%
inner product	26%	3%	21%	10%
banded linear systems solution	25%	2%	18%	4%
tridiagonal linear systems solution	20%	1%	8%	22%
general linear recurrence equations	32%	8%	22%	4%
equation of state fragment	16%	5%	6%	8%
alternating direction implicit integration	9%	0%	6%	9%
integrate predictors	13%	3%	12%	8%
difference predictors	31%	5%	10%	7%
first sum	27%	5%	12%	8%
first difference	9%	0%	1%	17%
2-D particle in a cell	45%	15%	41%	24%
1-D particle in a cell	32%	11%	10%	20%
casual Fortran	17%	9%	19%	12%
Monte Carlo search	19%	13%	11%	6%
implicit conditional computation	34%	13%	13%	5%
2-D explicit hydrodynamics fragment	11%	2%	20%	2%
general linear recurrence equations	20%	13%	10%	15%
discrete ordinates transport	43%	20%	21%	28%
matrix-matrix transport	7%	1%	15%	4%
Planckian distribution	19%	14%	17%	3%
2-D implicit hydrodynamics fragment	9%	1%	11%	5%
location of a first array minimum.	6%	5%	19%	4%

Legend: Intel (blue), AMD (orange)

Figure 5.9: Performance of LFK benchmark

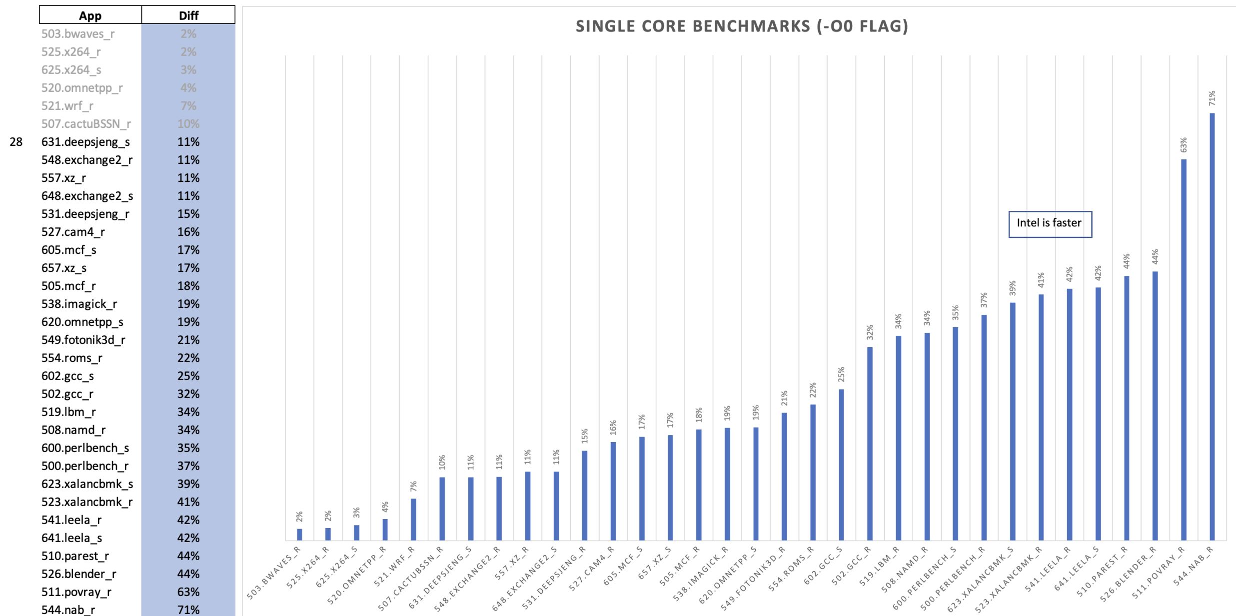
Fig 5.9 shows the performance difference for all the kernels. Results indicate that Intel performs better in all the kernels with the single-core unoptimized build, however, AMD starts performing better in 50% of the kernels as we run the multi-core optimized build. It can be seen that in the "tridiagonal linear systems solution" kernel, Intel is 20% faster, but it shifts to being 22% faster in AMD as we move to the right side of the table. We find similar performance shifts in kernels like "1-D particle in a cell", "discrete ordinates transport" & "implicit conditional computation".

This suggests Intel is better for single-core tasks and AMD for multi-core and vector instruction tasks. This is an interesting observation that falls in line with our original findings with serverless functions, so

we perform another benchmarking with SPEC-CPU 2017 to help better understand the performance differences.

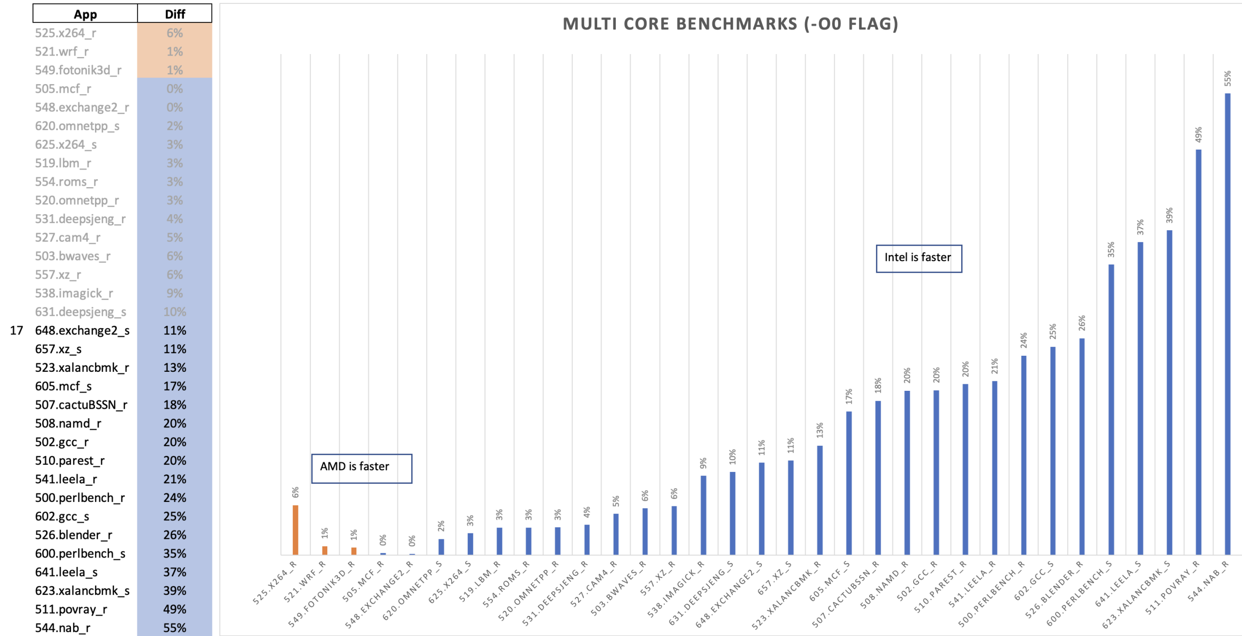
5.3 SPEC-CPU 2017

We use 34 applications in SPEC-CPU 2017 that are divided between 3 test suites - intrate, intspeed, fprate. Two suites contain applications that involve integer operations and one suite contains floating-point operations. We compile these applications to run in single and multiple cores of the CPUs with unoptimized (-O0) and optimized builds (-O3).



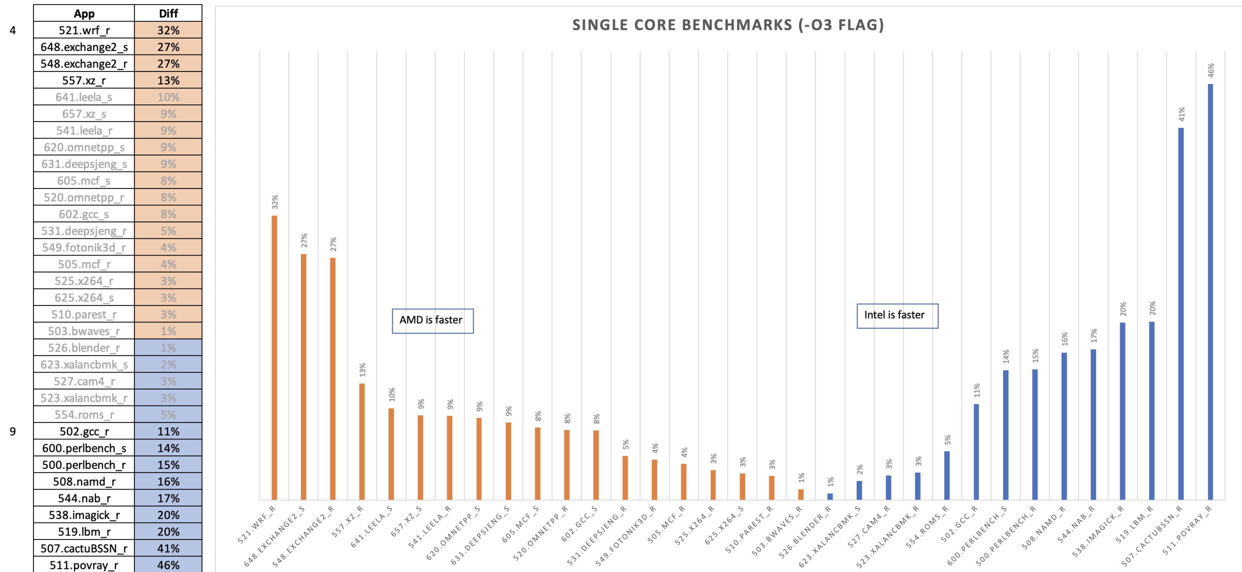
App	Diff
503.bwaves_r	2%
525.x264_r	2%
625.x264_s	3%
520.omnetpp_r	4%
521.wrf_r	7%
507.cactuBSSN_r	10%
631.deepsjeng_s	11%
548.exchange2_r	11%
557.xz_r	11%
648.exchange2_s	11%
531.deepsjeng_r	15%
527.cam4_r	16%
605.mcf_s	17%
657.xz_s	17%
505.mcf_r	18%
538.imagick_r	19%
620.omnetpp_s	19%
549.fotonik3d_r	21%
554.roms_r	22%
602.gcc_s	25%
502.gcc_r	32%
519.lbm_r	34%
508.namd_r	34%
600.perlbench_s	35%
500.perlbench_r	37%
623.xalancbmk_s	39%
523.xalancbmk_r	41%
541.leela_r	42%
641.leela_s	42%
510.parest_r	44%
526.blender_r	44%
511.povray_r	63%
544.nab_r	71%

Figure 5.10: SPEC-CPU Single Core O0 Optimized Benchmarks



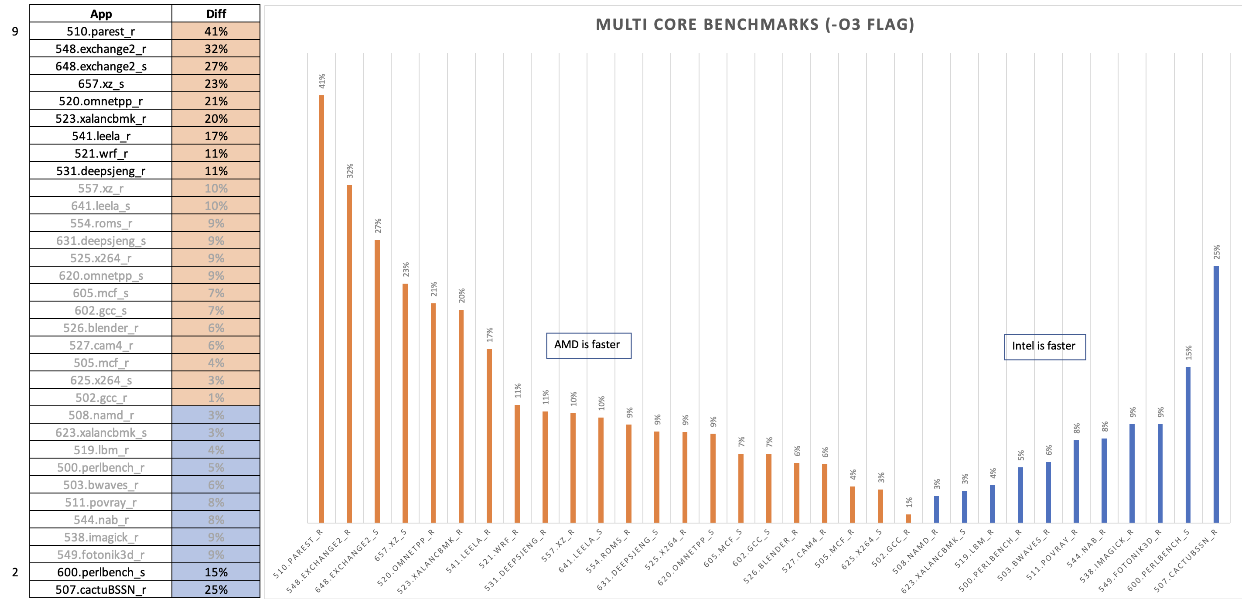
App	Diff
525.x264_r	6%
521.wrf_r	1%
549.fotonik3d_r	1%
505.mcf_r	0%
548.exchange2_r	0%
620.omnetpp_s	2%
625.x264_s	3%
519.lbm_r	3%
554.roms_r	3%
520.omnetpp_r	3%
531.deepsjeng_r	4%
527.cam4_r	5%
503.bwaves_r	6%
557.xz_r	6%
538.imagick_r	9%
631.deepsjeng_s	10%
648.exchange2_s	11%
657.xz_s	11%
523.xalancbmk_r	13%
605.mcf_s	17%
507.cactuBSSN_r	18%
508.namd_r	20%
502.gcc_r	20%
510.parest_r	20%
541.leela_r	21%
500.perlbench_r	24%
602.gcc_s	25%
526.blender_r	26%
600.perlbench_s	35%
641.leela_s	37%
623.xalancbmk_s	39%
511.povray_r	49%
544.nab_r	55%

Figure 5.11: SPEC-CPU Multiple Core O0 Optimized Benchmarks



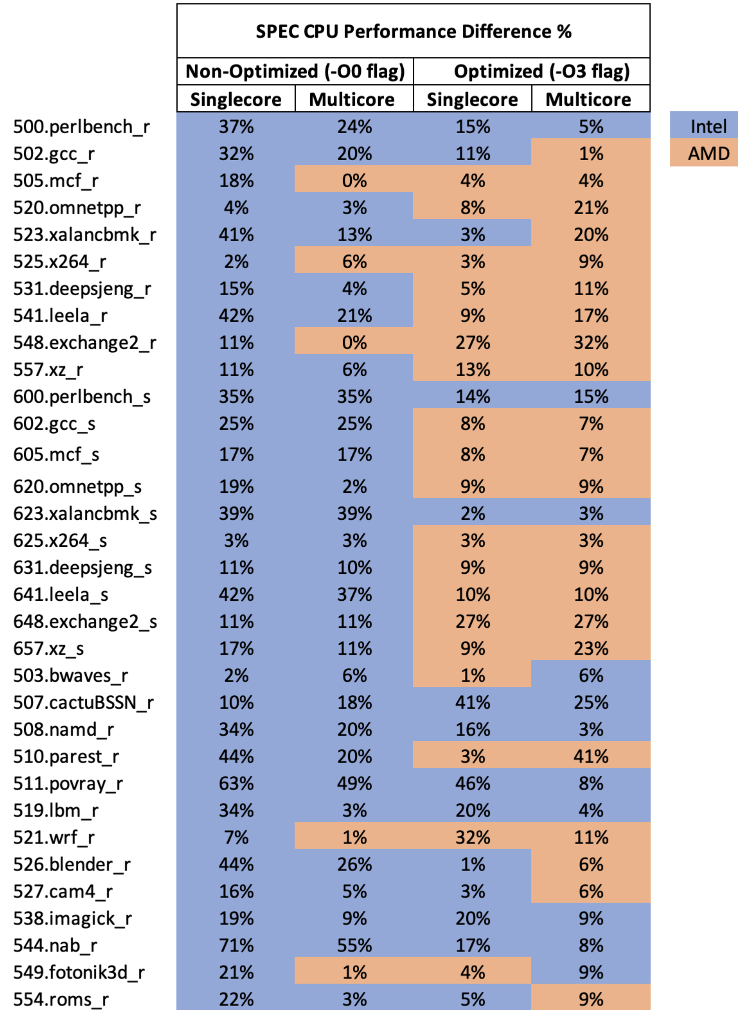
App	Diff
521.wrf_r	32%
648.exchange2_s	27%
548.exchange2_r	27%
557.xz_r	13%
641.leela_s	10%
657.xz_s	9%
541.leela_r	9%
620.omnetpp_s	9%
631.deepsjeng_s	9%
605.mcf_s	8%
520.omnetpp_r	8%
602.gcc_s	8%
531.deepsjeng_r	5%
549.fotonik3d_r	4%
505.mcf_r	4%
525.x264_r	3%
625.x264_s	3%
510.parest_r	3%
503.bwaves_r	1%
526.blender_r	1%
623.xalancbmk_s	2%
527.cam4_r	3%
523.xalancbmk_r	3%
554.roms_r	5%
502.gcc_r	11%
600.perlbench_s	14%
500.perlbench_r	15%
508.namd_r	16%
544.nab_r	17%
538.imagick_r	20%
519.lbm_r	20%
507.cactuBSSN_r	41%
511.povray_r	46%

Figure 5.12: SPEC-CPU Single Core O3 Optimized Benchmarks



	App	Diff
9	510.parest_r	41%
	548.exchange2_r	32%
	648.exchange2_s	27%
	657.xz_s	23%
	520.omnetpp_r	21%
	523.xalancbmk_r	20%
	541.leela_r	17%
	521.wrf_r	11%
	531.deepsjeng_r	11%
	557.xz_r	10%
	641.leela_s	10%
	554.roms_r	9%
	631.deepsjeng_s	9%
	525.x264_r	9%
	620.omnetpp_s	9%
	605.mcf_s	7%
	602.gcc_s	7%
	526.blender_r	6%
	527.cam4_r	6%
	505.mcf_r	4%
	625.x264_s	3%
	502.gcc_r	1%
	508.namd_r	3%
	623.xalancbmk_s	3%
	519.lbm_r	4%
	500.perlbench_r	5%
	503.bwaves_r	6%
	511.povray_r	8%
	544.nab_r	8%
	538.imagick_r	9%
	549.fotonik3d_r	9%
2	600.perlbench_s	15%
	507.cactuBSSN_r	25%

Figure 5.13: SPEC-CPU Multiple Core O3 Optimized Benchmarks



	SPEC CPU Performance Difference %			
	Non-Optimized (-O0 flag)		Optimized (-O3 flag)	
	Singlecore	Multicore	Singlecore	Multicore
500.perlbench_r	37%	24%	15%	5%
502.gcc_r	32%	20%	11%	1%
505.mcf_r	18%	0%	4%	4%
520.omnetpp_r	4%	3%	8%	21%
523.xalancbmk_r	41%	13%	3%	20%
525.x264_r	2%	6%	3%	9%
531.deepsjeng_r	15%	4%	5%	11%
541.leela_r	42%	21%	9%	17%
548.exchange2_r	11%	0%	27%	32%
557.xz_r	11%	6%	13%	10%
600.perlbench_s	35%	35%	14%	15%
602.gcc_s	25%	25%	8%	7%
605.mcf_s	17%	17%	8%	7%
620.omnetpp_s	19%	2%	9%	9%
623.xalancbmk_s	39%	39%	2%	3%
625.x264_s	3%	3%	3%	3%
631.deepsjeng_s	11%	10%	9%	9%
641.leela_s	42%	37%	10%	10%
648.exchange2_s	11%	11%	27%	27%
657.xz_s	17%	11%	9%	23%
503.bwaves_r	2%	6%	1%	6%
507.cactuBSSN_r	10%	18%	41%	25%
508.namd_r	34%	20%	16%	3%
510.parest_r	44%	20%	3%	41%
511.povray_r	63%	49%	46%	8%
519.lbm_r	34%	3%	20%	4%
521.wrf_r	7%	1%	32%	11%
526.blender_r	44%	26%	1%	6%
527.cam4_r	16%	5%	3%	6%
538.imagick_r	19%	9%	20%	9%
544.nab_r	71%	55%	17%	8%
549.fotonik3d_r	21%	1%	4%	9%
554.roms_r	22%	3%	5%	9%

Figure 5.14: SPEC-CPU 2017 Performance Overview

In the case of unoptimized builds (-O0) running on single-core (Fig 5.10), we notice that all 34 apps run faster in Intel, and in the case of multiple-cores (Fig 5.11), the performance starts to shift towards AMD.

In the case of O3 optimized benchmarks that run on single-core (fig 5.12), we observe that 9 applications have a performance difference of 11% or more in Intel, and only 4 runs faster in AMD. However, in the case of O3 optimized benchmarks that run on multiple cores (fig 5.13), we observe that only 2 applications run faster on Intel and AMD starts performing better with 9 applications.

We summarize the differences in this Fig 5.14. This indicates that Intel can better handle applications that run on single-core 25% faster and AMD performs 13% faster when running applications that utilize multiple cores with the SIMD instruction optimizations enabled.

This means that for most day-to-day applications developers can resort to using Intel machines with good performance, however, if they need to execute complex applications like image/video encoding, data analysis, or data compression, they can parallelize their code and use AMD machines for the same.

Serverless applications built using Node.js and Python utilize single core by default and they can be modified to leverage multiple cores by using the internal "cluster" and "multiprocessing" modules respectively. It allows forking processes that can run in parallel, each utilizing a separate core of the CPU [15] [37].

For parallelizing instructions, it is found that Node.js and Python don't automatically vectorize loops to improve performance [26]. For Node.js apps, developers can vectorize the code by using the official JavaScript SIMD API [30], and Python developers can leverage libraries like NumPy which have recently started to add native support for SIMD instructions in their code [32].

From the perspective of Cloud Service Providers (CSP), given that there is currently no way to automatically choose the most performant hardware for general workloads, they can either - (a) provide an option to developers to choose the type of hardware to run, or, (b) develop a hardware selection mechanism which is a machine learning model on top of profiling data to accurately predict which system would execute the code in the least amount of time. Since serverless functions are time-bound and shortlived, CSPs would have to determine the correct system only once and run subsequent invocations of the same function there. We leave this as an open area of research for future work.

Chapter 6

Discussion

As the competition of AMD and Intel continues, heterogeneity in cloud servers will increase, this makes it very important for cloud servers to ensure effective utilization of resources. As many more factors affect the performance of a serverless function, we plan to study other performance metrics collected by "perf" to find the ones that have the strongest correlation with the total execution time of a function. We will build regression models that allow us to accurately predict the execution time of a new function that will be executed on the cloud. In the future, we will also look into ways to build a profile for a given system so that we can build cross-system prediction models. This will be helpful as cloud providers wouldn't have to re-run every single function across all the machines to find the best one for a given function. We also intend to use tools like PerfKit [16] that allow us to benchmark systems across various cloud providers. The scores are consistent that allow us to draw insights from multi-cloud environments.

Secondly, we also found that "perf" allows users to inspect the performance of a code at the assembly instruction level to find bottlenecks. Although this is mostly used by developers to improve the performance of an application, we can use it to find the most time-consuming instructions for a system and create a unique profile. That way, cloud providers can build systems that quickly scan a given assembly code and group recurring instructions and then decide which system is best suited to execute those instructions faster.

To summarize, we hope to open up this area of research further to consider the following directions -

- Find other prominent factors that affect the performance of an application.
- Profile the performance of serverless function across a wide variety of CPU systems that are available on various cloud providers.
- Build regression models to allow cross-system prediction of execution time for a given application.
- Group hottest instructions for a given system to build a mechanism for cloud providers to analyze assembly code and decide which hardware to run the function on.

Chapter 7

Conclusion

We analysed the performance differences of various serverless functions on Intel and AMD CPUs. We used OpenWhisk, an open-source tool to run our functions. We benchmarked applications targeting different aspects affecting performance such as CPU cores (rsa, fft, monte-carlo), memory (img-resize), cache (matrix-multiply), vector instructions (ocr-img). We also profiled industry standard benchmarks like SPEC CPU and LFK loops across four different configuration builds, applications compiled with two different optimization flags (O0 and O3) each utilizing single and multiple-cores of the CPU. We found that Intel performed 25% faster in applications that were configured to utilize single-core of the CPU compiled with O0 flag, although AMD performed 13% faster in applications that were configured to utilized multiple-cores compiled with O3 flag. AMD showed performance improvements for applications that utilize vector instructions, these involve optimizing for-loops and complex arithmetic operations to be broken down into tasks that can be executed parallely.

With the rise in CPU heterogeneity, cloud service providers can smartly provision workloads based on this data for improved performance. If developers are given a choice to pick the underlying execution hardware, this knowledge can help them build their apps specific to the hardware allowing more predictable and efficient performance at reduced costs.

Bibliography

[1] Gojko Adzic and Robert Chatley. "Serverless Computing: Economic and Architectural Impact". In: ESEC/FSE 2017 (2017), pp. 884–889. DOI: 10.1145/3106237.3117767. URL: https://doi.org/10.1145/3106237.3117767.

[2] AMD. "AMD "Zen" Core Architecture". In: (2017). URL: https://www.amd.com/en/technologies/zen-core.

[3] AMD. "AMD EPYC™ and Google Cloud Platform Instances". In: (2023). URL: https://www.amd.com/en/processors/epyc-google-cloud.

[4] AMD. "AMD Ryzen™ 5 5600X". In: (2023). URL: https://www.amd.com/en/product/10471.

[5] AWS. "AWS and AMD - Powerful processors to right size your workload". In: (2023). URL: https://aws.amazon.com/ec2/amd/.

[6] AWS. "AWS Lambda". In: (2023). URL: https://aws.amazon.com/lambda/.

[7] Ioana Baldini, Paul Castro, Kerry Chang, et al. "Serverless Computing: Current Trends and Open Problems". In: (2017). Ed. by Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, pp. 1–20. DOI: 10.1007/978-981-10-5026-8_1. URL: https://doi.org/10.1007/978-981-10-5026-8_1.

[8] Akshara Bassi. "Data Center CPU Market: AMD Surpasses Intel in Share Growth". In: (2023). URL: https://www.counterpointresearch.com/data-center-cpu-market-amd-surpasses-intel-share-growth/.

[9] businesswire. "7.72 Billion Function-as-a-Service Market 2017". In: (2017). URL: https://www.businesswire.com/news/home/20170227006262/en/7.72-Billion-Funct%20ion-as-a-Service-Market-2017---Global.

[10] Paul Castro, Vatche Ishakian, Vinod Muthusamy, et al. "The Rise of Serverless Computing". In: *Commun. ACM* 62.12 (Nov. 2019), pp. 44–54. ISSN: 0001-0782. DOI: 10.1145/3368454. URL: https://doi.org/10.1145/3368454.

[11] Paul Castro, Vatche Ishakian, Vinod Muthusamy, et al. "The Rise of Serverless Computing". In: *Commun. ACM* 62.12 (Nov. 2019), pp. 44–54. ISSN: 0001-0782. DOI: 10.1145/3368454. URL: https://doi.org/10.1145/3368454.

[12] Colin Chartier. “Serverless is more expensive than you’d expect”. In: (2021). URL: https://webapp.io/blog/the-hidden-costs-of-serverless/.

[13] Denis. “PaaS vs FaaS”. In: (2021). URL: https://dbaltor.medium.com/paas-vs-faas-229919694163.

[14] Docker. “Runtime options with Memory, CPUs, and GPUs”. In: (2023). URL: https://docs.docker.com/config/containers/resource_constraints/#memory.

[15] Dave Dopson. “Node.js on multi-core machines”. In: (2011). URL: https://stackoverflow.com/questions/2387724/node-js-on-multi-core-machines.

[16] Google. “PerfKitBenchmark”. In: (2023). URL: https://github.com/GoogleCloudPlatform/PerfKitBenchmarker.

[17] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, et al. “Serverless computing: One step forward, two steps back”. In: *arXiv preprint arXiv:1812.03651* (2018).

[18] IBM. “A Brief History of Cloud Computing”. In: (2017). URL: https://www.ibm.com/cloud/blog/cloud-computing-history.

[19] IBM. “Cloud Computing History”. In: (2017). URL: https://www.ibm.com/cloud/blog/cloud-computing-history.

[20] IBM. “What is FaaS (Function-as-a-Service)?” In: (2023). URL: https://www.ibm.com/topics/faas.

[21] Intel. “Looking Beyond the CPU”. In: (2021). URL: https://www.intel.com/content/www/us/en/products/performance/amd-cloud-facts.html.

[22] Intel. “Intel® Instruction Set Extensions Technology”. In: (2022). URL: https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html.

[23] Intel. “Intel® Core™ i5-12400 Processor”. In: (2023). URL: https://ark.intel.com/content/www/us/en/ark/products/134586/intel-core-i512400-processor-18m-cache-up-to-4-40-ghz.html.

[24] Kevin Jain. “Get Serverless Performance Metrics”. In: (2023). URL: https://gist.github.com/thatsKevinJain/654fc8e047e715d6f61f553f27e0ecb6.

[25] Kevin Jain. “Openwhisk Installation Script”. In: (2023). URL: https://gist.github.com/thatsKevinJain/09920470930f2acbd4d3282d68d027c6.

[26] jmrk. “Is there any way to get Node.JS and V8 to automatically vectorize simple loops?” In: (2020). URL: https://stackoverflow.com/questions/63864497/is-there-any-way-to-get-node-js-and-v8-to-automatically-vectorize-simple-loops.

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, et al. “Cloud programming simplified: A berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383* (2019).

[28] Gentoo Linux. “GCC Documentation”. In: (2023). URL: https://wiki.gentoo.org/wiki/GCC_optimization#-O.

[29] Pascal Maissen, Pascal Felber, Peter Kropf, et al. “FaaSdom: A benchmark suite for serverless computing”. In: (2020), pp. 73–84.

[30] MDN. “SIMD types”. In: (2017). URL: http://www.devdoc.net/web/developer.mozilla.org/en-US/docs/Web/JavaScript/SIMD_types.html.

[31] CPU Monkey. “CPU comparison with benchmarks”. In: (2023). URL: https://www.cpu-monkey.com/en/compare_cpu-intel_core_i5_12400-vs-amd_ryzen_5_5600x.

[32] NumPy. “CPU/SIMD Optimizations”. In: (2022). URL: https://numpy.org/doc/stable/reference/simd/index.html.

[33] OpenFaas. “OpenFaas”. In: (2023). URL: https://www.openfaas.com/.

[34] Apache OpenWhisk. “openwhisk-devtools”. In: (2023). URL: https://github.com/apache/openwhisk-devtools/tree/master/docker-compose.

[35] Apache OpenWhisk. “What is Apache OpenWhisk?” In: (2023). URL: https://openwhisk.apache.org/.

[36] Fn Project. “Open Source. Container-native. Serverless platform.” In: (2023). URL: https://fnproject.io/.

[37] Python. “multiprocessing — Process-based parallelism¶”. In: (2023). URL: https://docs.python.org/3/library/multiprocessing.html.

[38] RedHat. “IaaS vs. PaaS vs. SaaS”. In: (2022). URL: https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas.

[39] RedHat. “Chapter 18. Getting started with perf”. In: (2023). URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/getting-started-with-perf_monitoring-and-managing-system-status-and-performance.

[40] rzcoder. “Node-RSA”. In: (2021). URL: https://github.com/rzcoder/node-rsa.

[41] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. “Architectural implications of function-as-a-service computing”. In: (2019), pp. 1063–1075.

[42] SigOps. “The Increasing Heterogeneity of Cloud Hardware and What It Means for Systems”. In: (2020). URL: https://www.sigops.org/2020/the-increasing-heterogeneity-of-cloud-hardware-and-what-it-means-for-systems/.

[43] SPEC. “SPEC CPU 2017”. In: (2022). URL: https://www.spec.org/cpu2017/Docs/overview.html#benchmarks.

[44] Jarrod’s Tech. “Intel i5-12400F vs AMD Ryzen 5 5600X - Best 6 Core CPU?” In: (2022). URL: https://www.youtube.com/watch?v=1GXb-doKETg&ab_channel=Jarrod%27sTech.

[45] tesseract-ocr. "Tesseract OCR". In: (2023). URL: https://github.com/tesseract-ocr/tesseract.

[46] Userbenchmark. "Intel Core i5 12400 vs AMD Ryzen 5 5600X". In: (2023). URL: https://cpu.userbenchmark.com/Compare/Intel-Core-i5-12400-vs-AMD-Ryzen-5-5600X/4122vs4084.

[47] Amazon Prime Video. "Scaling up the Prime Video audio/video monitoring service and reducing costs by 90". In: (2023). URL: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90.

[48] VMWare. "Kubeless". In: (2021). URL: https://github.com/vmware-archive/kubeless.

[49] waverian. "LFK-MP". In: (2022). URL: https://github.com/waverian/lfk-mp-benchmark.

[50] Wikipedia. "Livermore Loops". In: (2022). URL: https://en.wikipedia.org/wiki/Livermore_loops.

[51] Tianyi Yu, Qingyuan Liu, Dong Du, et al. "Characterizing serverless platforms with serverlessbench". In: (2020), pp. 30–44.