SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Design and Evaluation of a Benchmarking Service for AWS EC2 Hardware Metrics

Michał Jakub Szczepaniak

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Design and Evaluation of a Benchmarking Service for AWS EC2 Hardware Metrics

Entwurf und Evaluation eines Benchmarking-Dienstes für AWS EC2 Hardwaremetriken

Author:	Michał Jakub Szczepaniak
Supervisor:	Prof. Dr. Viktor Leis
Advisor:	Till Steinert
Submission Date:	07.11.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 07.11.2024 Michał Jakub Szczepaniak

Acknowledgments

First and foremost, I want to thank **my parents - Anna and Krzysztof**. Without their guidance and support over nearly twenty years of my education, I could not graduate with a Master's degree at the Technical University of Munich.

Additionally, I want to express my gratitude to my advisor, **Till Steinert**, whose feedback significantly influenced this thesis and made it more valuable. I am also deeply grateful to my supervisor, **Prof. Dr. Viktor Leis**, for the opportunity to work on this topic and whose lectures taught me a lot about cloud and data processing. Moreover, I want to thank **Prof. Dr. Thomas Neumann**, whose lecture about databases and data engineering considerably deepened my knowledge and broadened my interest in these topics.

Finally, I offer sincere thanks to all those people who were an integral part of my journey through academia and professional life. I convey my sincere thanks to my friends from the Poznań University of Technology - **Piotr, Wiktor, and Filip**, whom I could always rely on during my studies. I also want to express my gratitude to my friends from **TUMuchData** and **AKAI** for creating communities where I could develop my passion for computer science, both in Munich and Poznań. Furthermore, I would like to thank my friends from **MINGA** and **PolSoc** for enriching my time in Munich. Last but not least, I am appreciative to my colleagues from **JetBrains, Dobly, and Consdata** for the valuable professional experience that helped me during my studies.

Abstract

Cloud processing has become one of the most commonly used technology in recent years. The most basic service cloud providers offer is virtual machines, e.g., Elastic Compute Cloud (EC2) instances from Amazon Web Services (AWS). In order to achieve the best performance relative to the cost, it is crucial to choose the proper type of instance.

However, the cloud providers are not transparent about the performance of the offered instances because they do not provide hardware metrics. For example, they describe the performance of the instances using vCPUs. However, a number of vCPUs do not express all the differences between processors, such as different architectures, manufacturers, and generations, and whether they support hyperthreading or not. For this reason, benchmarking is the only way to obtain valuable information about hardware metrics that can be compared. Nevertheless, people encounter many obstacles during benchmarking, such as uncertainty of the results or the high cost and time consumption of benchmarking a large set of instances.

This thesis presents TUManyBenchmarks - the extensible benchmarking service for comparing EC2 instances based on hardware metrics. We demonstrate its design and implementation, allowing automatic benchmark execution after contributing the benchmark files to the repository. Additionally, we show how it can support users in benchmarking and cost-effective instance selection. We demonstrate examples of the benchmarks executed with the service and observations on their results that highlight the differences between EC2 instances. Moreover, we give examples of the models for cost-effective instance selection based on the data collected by the service and compare the service with the existing solutions.

Contents

1. Introduction

1.1. Transition to the cloud

In recent years, one of the most crucial trends in the IT industry was the transition from on-premise to cloud infrastructure. Companies decide to stop maintaining their own servers because adapting to the changing workload is challenging, especially if the workload contains frequently occurring peaks. Due to its lack of elasticity, on-premise architecture can lead to under- or over-provisioning problems, which can be highly harmful to cost-effectiveness. For this reason, companies decide to move their infrastructure to a more elastic environment - the cloud. While using cloud services, they can easily add new resources on demand while workload peaks occur and remove them later when they are not needed, ensuring optimal resource utilization.

The recent reports forecast the further intensive growth of funds spent on cloud technologies in the following years[26]. The size of the cloud computing market in 2023 was estimated at above 600 billion USD. However, Grand View Research specialists predict that by 2030, it will be worth above 2 trillion USD.

1.2. Transparent cost-effective selection of virtual machines

Despite many advantages, the cloud can be expensive because it does not automatically guarantee cost efficiency. Instead, it is the cloud user's responsibility to consider and compare different available alternatives, which can be challenging even for cloud providers[33]. The most fundamental service in each cloud infrastructure is virtual machines. They are the primary element of each cloud architecture because they are the natural replacement for on-premise servers and are crucial elements to achieve cost-efficiency. However, at the same time, virtual machines are one of the most complex in achieving cost-efficiency because they characterize high heterogeneity. AWS currently offers over 800 EC2 instance types[48] and constantly adds new types. For this reason, selecting the most cost-efficient one from so many alternatives is challenging and can cause an overchoice effect.

The fundamental factor during selection is the machine's parameters. Cloud providers characterize each instance using multiple parameters, such as the number of vCPUs, memory size, and network throughput. This information may be sufficient for some customers, yet they do not provide the full image of the machine because it lacks hardware metrics. Consequently, many essential characteristics are missing, which makes precise comparisons impossible.

For example, the cloud provider presents the number of vCPUs available to use by the virtual machine. However, the different EC2 instances use different processors. Consequently, their performance differs even if they have the same number of vCPUs. Also, the fact that the instance has an X times higher number of vCPUs does not have to imply an X times higher performance. This is caused by the fact that some processors use hyperthreading and some do not. Therefore, the two vCPUs can run on single physical core if the CPU uses hyperthreading. However, the performance of the two threads running on separate physical cores is better than the performance of two threads running on one physical core due to hyperthreading. This difference can significantly influence the performance of the virtual machine.

The customer can find analogical shortcomings in the data provided about a network bandwidth. In the past, AWS described network throughput using expressions such as "Low" or "Moderate," which have no specific value. Currently, AWS describes most instances using terms such as "Up to X Gigabit." However, this is the value of burst, which is available for a limited, not clearly stated time, after which the network throughput returns to a baseline bandwidth[4].

1.3. Difficulty of benchmarking

These examples show that cloud providers do not provide enough information to allow customers to make a conscious choice. Consequently, customers must rely on benchmarks, enabling them to discover and compare missing metrics. However, there are a few problems with this approach.

The first problem is that creating a proper benchmark is difficult. The correct benchmark should focus only on the tested characteristic and minimize the influence of other factors on the result. The user can also use existing benchmarks, but this does not guarantee correctness, as, for example, the benchmark can be executed with incorrect parameters. Nevertheless, correctness is crucial because faulty results can lead to false conclusions.

Furthermore, benchmarking many instances can be time-consuming and expensive. The user has to test multiple instances to collect reliable data for comparison. Some benchmarks, like SPEC CPU 2017 or fio, require a few hours to complete. Together, these two factors can generate a high cost. Benchmarking is also an ongoing effort as cloud providers continuously release new instances. Clients must also test these new instances to have a detailed view of all the alternatives again.

Moreover, the benchmark results are relevant for many people. For this reason, they execute benchmarks on their own. However, clients are interested in the same properties and repeat each other work, which generates unnecessary costs. For this reason, there exist services that collect information about benchmark results. Nonetheless, such services usually only contain data about a single benchmark.

1.4. Motivation for a new service

Both the lack of hardware metrics delivered by cloud providers and the difficulty of benchmarking lead us to the conclusion that there is a need for a new benchmarking service. We believe such a service should characterize two attributes: extensibility and transparency. Extensibility means that every user should be able to upload their benchmark. This feature would allow for simultaneously collecting multiple metrics from many instances without unnecessary work and, as a result, allow using multiple metrics for comparison. The second characteristic should be transparency. Transparency means everyone can check what and how benchmarks are executed to provide trustworthiness in the presented results. Moreover, the second aspect of transparency is achieving transparency of instances' performance within and between vendors by publishing their hardware metrics.

In this thesis, we present a design of TUManyBenchmarks - the extensible benchmarking service for comparing EC2 instances based on hardware metrics. We demonstrate a design that allows us to implement the mentioned characteristics and evaluate the service's features and potential practical applications. We created the service meeting these requirements for AWS as the most popular cloud provider, but its functionality can be easily extended to others.

1.5. Outline

The rest of this thesis is structured as follows. Chapter 2 is a brief overview of technologies used during the service design and benchmark used during its evaluation.

Chapter 3 focuses on the design and implementation of the benchmark service. Firstly, in section 3.1, we specify the functional and non-functional requirements for the system. Based on them, we describe the design and implementation details in the following sections. In Section 3.2, we focus on uploading the benchmark to the service. Section 3.3 presents the process of benchmark execution. Section 3.4 concentrates on benchmark result presentation and querying.

Section 4 evaluates the practical application of the system. Section 4.1 presents example observations about the EC2 instance's hardware metrics that our service helps to highlight. Section 4.2 presents example models that can be helpful for users while searching for the most proper instance. Section 4.3 focuses on the advantages and simplifications of the benchmarking process that are possible due to using TUMany-Benchmarks. Finally, we revisit the requirements and goals in Section 4.4.

Section 5 covers related work and compares other solutions with our benchmark service. Section 6 summarises the thesis and presents the further direction of the development.

1.6. Contribution

The main contributions of the thesis are:

- an implementation of a service to run benchmarks,
- an implementation of web application allowing for querying benchmark data,
- an evaluation of the benefits of the benchmark service.

2. Background

The implementation and evaluation described later will be based on solutions described in this chapter. Section 2.1 characterizes the frameworks, databases, and tools the service uses to run benchmarks and presents their results. Section 2.2 covers the information about the benchmarks used during the evaluation described in Section 4.

2.1. Technology

2.1.1. GitHub Actions

GitHub Actions[24] is a Continuous Integration (CI) and Continuous Delivery (CD) tool for executing jobs in GitHub repositories. It allows users to define and manage workflows executed on every new commit or Pull Request (PR). These workflows allow for automating tasks such as building projects, testing, and deployment. To use GitHub Actions in their repository, a user must provide a YAML file defining the workflow steps in the directory `.github/workflows`. The user can use one of many predefined actions or write a custom script. The custom script functionality allows running arbitrary code, such as Bash or Python scripts, so it can be used to perform any actions on the repository. For this reason, GitHub Actions can be used to automate and support a wide range of developers' tasks.

2.1.2. Spring Framework

Spring Framework[51] is the widely used framework for web application development with Java and Kotlin. It simplifies the implementation of the applications by providing features, such as dependency injection and integration with many third-party libraries. Dependency injection allows an object's dependencies to be provided externally instead of hard-coding them within the object, which results in better flexibility and testability. Spring consists of multiple modules that can be combined according to the developer's needs. The primary module is Spring Boot, which aims to enable the fast creation of stand-alone applications with an embedded server. One of its main features is automatic

Spring's libraries configuration. Another valuable module is Spring Data, which allows communication with multiple databases using standardized data repository interfaces. One more helpful Spring module is Spring WebFlux, which helps create reactive, non-blocking web applications using Netty. The applications based on it characterize better performance and resource utilization than classical Spring applications due to relying on non-blocking operations. All these solutions reduce the amount of code the developer has to write and maintain by providing ready-to-use solutions.

2.1.3. MongoDB

MongoDB[37] is one of the leading NoSQL document databases. Its main characteristics are storing data as Binary JSON and schemaless. These features enable the storage of data like JSON, which has multiple nested objects and different field structures. Moreover, the database has many features that are typical of SQL databases, such as transactions and views. Additionally, MongoDB allows users to express complex queries using Aggregation Pipelines[36], which can consist of an unlimited number of operators like `$map`, `$lookup`, `$unwind`, and `$group`. Aggregation Pipelines can also be used as input queries for views, simplifying communication with the database in the case of complex queries. Furthermore, MongoDB also offers high availability by using multiple replicas. MongoDB's advantages cause it to be widely used in web development.

2.1.4. Ansible

Ansible[41] is a widely used open-source automation tool for configuring virtual machines. In order to use Ansible, the user has to create a YAML file called playbook in which they define tasks. Later, the user can call the Ansible command from the terminal and pass the path to the playbook as a parameter. Ansible reads the playbook and executes the specified tasks, such as downloading dependencies or compiling files. While specifying tasks, the user can use one of many predefined tasks, such as creating a virtual environment or cloning a git repository. However, the user can also use Ansible to create and edit files or execute arbitrary commands, including the ones requiring root privileges. Ansible's flexibility and variety of built-in functions make it a standard for automatically configuring virtual machines.

2.1.5. Angular

Angular[25] is one of the most popular front-end frameworks for creating Single Page Application (SPA) developed by Google. Angular uses TypeScipt[35], a language built upon JavaScript offering strong typing and early error detection. Angular has multiple features that help with webpage creation, such as dependency injection and two-way data binding. Two-way data binding allows for automatic synchronization between the model and the view, resulting in instant view changes based on model modifications done in the view. Angular follows a component-based architecture. The application is structured into reusable components, encapsulating logic, template, and styles and separating them from other components. Because of Angular's popularity, many JavaScript libraries provide ready-to-use components for fast and seamless integration. All these features result in a well-established position of Angular among front-end frameworks.

2.1.6. WebAssembly

WebAssembly[52] (Wasm) is a binary instruction format for a stack-based virtual machine. It was designed to provide high-performance and portable running code in browsers. It allows executing code written, for example, in C or C++, together with JavaScript in the browser. Browsers execute WebAssembly in a memory-safe, sandboxed environment, which ensures security. To use code as WebAssembly, the source code has to be compiled into the proper format - `.wasm`.

2.1.7. DuckDB

DuckDB[40] is an in-process, high-performance database for analytical queries. DuckDB can be run in multiple environments, like the command line or the most popular programming languages. It is also available as WebAssembly[30], enabling it to be used inside browsers on the users' side. DuckDB allows for all operations typical for SQL databases, such as creating tables, manipulating data, and querying them. Its syntax is compatible with PostgreSQL, so it does not require learning a new syntax to work with it. Additionally, it has many extensions, enabling additional features like full-text search indexes or reading and writing data from various sources and formats, such as JSON, S3, and PostgreSQL. For this reason, DuckDB is gaining popularity as a solution for efficient analytical data processing.

2.2. Benchmarks

2.2.1. iperf

iperf[23] is a benchmark for testing the network bandwidth over time. To use this benchmark, a user needs at least two machines. The first works as a server, which receives data. The second is a client, which sends data. The benchmark works with TCP, UDP, and SCTP protocols and supports IPv4 and IPv6. The client program is configurable. The user can set parameters such as reported intervals, the number of threads used to send data, and the benchmark duration time. As a result, the user receives a bandwidth achieved by each thread and their sum.

2.2.2. sockperf

sockperf[34] is a benchmark to analyze network latency. To use this benchmark, a user needs at least two machines. The first works as a server, which receives data. The second is a client, which initializes connection. The fundamental client mode is ping-pong, which measures round-trip latency by sending packets to a server and awaiting the response. As a result, a user receives a report with minimum latency, maximum latency, average latency, standard deviation, and tail latency for different values of percentiles.

2.2.3. SPEC CPU 2017

SPEC CPU 2017® [43, 44] is a benchmark for CPU testing. It measures performance using compute-intensive operations, whose results depend on three aspects: processor, memory, and compilers. The benchmark programs are based on actual end-user applications, such as compiling code with GCC (502.gcc_r) or compressing video (525.x264_r).

SPEC CPU groups its 43 benchmarks into four suites: intspeed, fpspeed, intrate, fprate. Each suite is characterized by two features: whether it contains speed or throughput benchmarks and whether it contains integer or floating-point benchmarks. SPECspeed® 2017 is a time-based metric, so the score is higher when processing time is smaller. SPECrate® 2017 is a throughput metric that indicates completed work per unit of time. The main difference between these two benchmark types is that speed benchmarks allow users to use OpenMP and specify the number of used threads. In the case of throughput benchmarks, OpenMP is disabled, so processing is done on one thread. The additional difference is that speed benchmarks run only one benchmark copy, whereas throughput benchmarks can run multiple concurrent copies and, in this way, employ the CPU's multithreading. The distinction between integer and floating-point benchmarks is more straightforward and indicates which type of operation dominates for a given benchmark. Integer benchmarks are more critical for users who run classical programs, whereas floating-point benchmarks are more significant for those who focus on modeling and predictions.

The results of SPEC CPU are based on the execution times of particular tests. To calculate SPECspeed, SPEC CPU measures time for each benchmark in the suite. Then, it calculates the performance ratio by dividing the time measured on a reference machine by the time achieved during the test. For example, if the result equals 2, the tested machine requires 2 times less time to complete the same tasks. For SPECrate, there is a slight difference as the result additionally depends on the number of copies. The reference machine runs only one copy. For this reason, if the system runs multiple concurrent copies, the throughput result must be multiplied by the number of copies to represent the higher throughput. Based on these ratios, SPEC CPU calculates the final result. The SPEC CPU runs multiple benchmark iterations to provide more trustworthy results. If the measurements are done three times, the systems select the median. If there are only two iterations, the system selects the worse one. The SPEC CPU uses these selected values to compute the final result representing the machine's overall performance using the geometric mean.

Additionally, each SPEC CPU benchmark result describes base and peak metrics. SPEC CPU distributes benchmarks as source code. Consequently, they must be compiled on a machine before execution. However, the performance and applied optimizations differ depending on the compilation option. The base metric is obtained when all the benchmark source codes from the given suite are compiled using the same flags in the same order for a given language. On the other hand, to obtain peak metrics, SPEC CPU builds each benchmark with a set of individually selected flags and allows feedback-directed optimization. This causes the peak metric benchmark executable to be more optimized and produce more efficient code. Both metrics can be valuable to users depending on their use cases.

3. Design and implementation

This chapter will first consider and collect requirements for a benchmarking service for AWS EC2 hardware metrics. Later, we will present the service's components developed based on these requirements. We will start the analysis from the moment a user uploads a benchmark. Later, we will focus on setting up the AWS infrastructure required to run benchmarks and on benchmark preparation, execution, and saving results. Lastly, we will describe the web application for querying benchmark results and comparing instances. Based on the implementation, we will evaluate the service in the Chapter 4.

3.1. Service's requirements

The first step in designing any system is specifying the functional and non-functional requirements to define a clear development goal. We derive these requirements from the two main features that should characterize our service - extensibility and transparency. Extensibility should allow users to add new benchmarks, whereas transparency should enable users to query the results of completed benchmarks to get details about tested machines. These are the two main functionalities that the service has to offer from the user's perspective.

The process of adding a benchmark should allow for the upload of multiple files of different types (e.g., `.cpp` files, Bash scripts). While uploading the benchmark, the user should also provide the configuration file specifying the steps required for benchmark execution and result presentation. Furthermore, adding benchmarks should be available for any user. However, before accepting a benchmark for execution, the maintainer must verify it. The maintainer's role is to prevent the addition of malicious or incorrect code (e.g., code containing infinity loops preventing the benchmarks from completion). The verification process should be supported by the automatic validation of the configuration file to detect missing fields or wrong values. Finally, users should be able to go into details of all accepted benchmarks to see executed commands and files.

The second crucial functionality is presenting the result of the benchmark executions. The data should be available to all users through the web application. This application should enable users to find the instance best suited for them based on hardware metrics. They should be able not only to filter instances and their results but also to write complex queries. Additionally, it should be possible to export the queried data in a CSV format, enabling users to process them on their own. Moreover, the web application should allow for plotting benchmark results for a given instance to enable analysis and present results' fluctuation. Finally, the application should enable comparing different EC2 instances. The comparison view should present the plots with benchmark results. Additionally, if one benchmark is executed for multiple compared instances, their results should be presented on one common plot for easier comparison.

Besides the requirements from the user perspective, the service must meet other requirements. The most fundamental one is the ability to regularly run uploaded benchmarks within a repeatable environment and save their results in a database. Additionally, it should download, process, and save information about EC2 instances in a database to show them later on the web page.

Collecting all these requirements led to designing a service consisting of 5 components presented in Figure 3.1. In the following section, we will focus on describing each component.

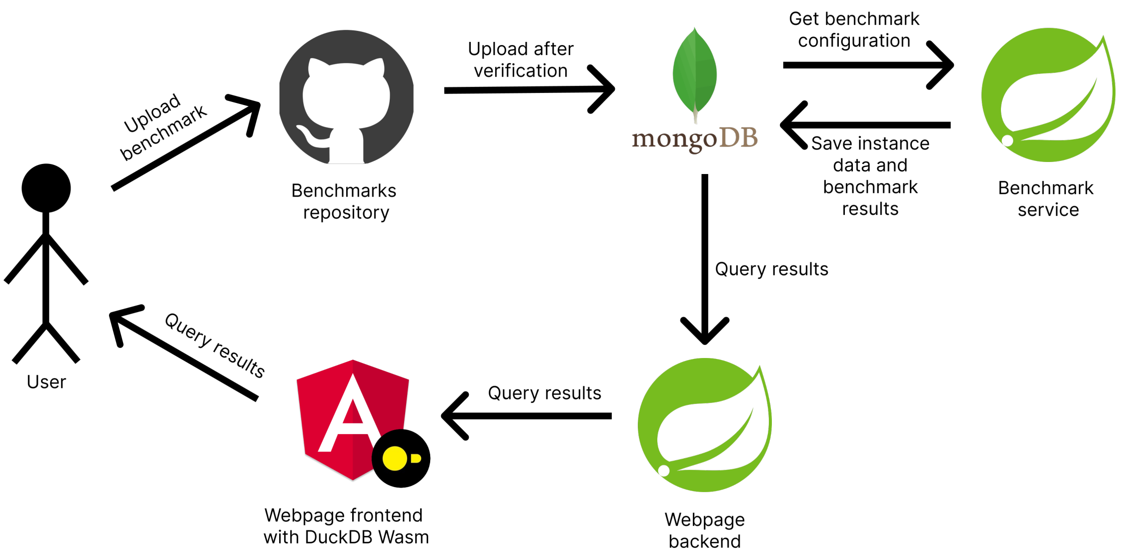

Upload benchmark
Benchmarks repository
Upload after verification
mongoDB
Get benchmark configuration
Save instance data and benchmark results
Benchmark service
Query results
User
Query results
Query results
Webpage frontend with DuckDB Wasm
Webpage backend

Figure 3.1.: High-level overview of the service architecture.

3.2. Uploading benchmark

The first step in executing any benchmark is to upload it to the service. We decided to use the GitHub repository for this purpose. A public GitHub repository enables the upload of new files of any type in a familiar way for everyone interested in adding new benchmarks - Computer Science specialists. Additionally, at the same time, it enables them to analyze details of each benchmark and even execute it on their own (with some exceptions described in Section 3.2.1). This approach is similar to the one used by Clickbench [1]. However, instead of uploading benchmark results, the user has to upload the benchmark itself.

To upload a new benchmark, the user has to create PR containing all files required to run a benchmark together with a configuration file `configuration.yml` (see Section 3.2.1) and optional Ansible files [41] (see Section 3.3.1) to the repository. After creating a PR, the GitHub Actions pipeline [24] validates the content of the YAML file. Using the GitHub Action is practical because it is run automatically after each commit to the PR and is free, reducing the overall service cost. Within the pipeline, the central role plays a Python script. It validates whether the file contains all required fields and their values meet the given constraints (e.g., specified EC2 instance types exist). Later, if the pipeline is completed successfully, the maintainer should additionally verify the PR to ensure the new benchmark is correct. The maintainer should confirm that the benchmark contains no code that could lead to benchmark failure, malicious behavior, or infinite execution due to infinite loops. The maintainer is also responsible for rejecting the benchmark if, for example, they decide that its execution may be too expensive or a similar benchmark already exists.

After verification and acceptance by the maintainer, the PR with the new benchmark is merged into the `main` branch. The merge activates the following GitHub Actions pipeline, which parses and uploads information from `configuration.yml` stored in GitHub to the MongoDB database. Consequently, the other services do not have to download files from GitHub and parse them during each benchmark execution or while presenting results on the web page. Instead, the other services can query data stored in the MongoDB instance anytime they want to use them.

3.2.1. Configuration file

The role of the configuration file is to pass all relevant information about the benchmark to the benchmark service. This enables the benchmark service to know when and how to execute the benchmark, and the web application knows how to present generated results. We decided to use the YAML format as it is easy to parse by machine and easy for users to read and write. The configuration file consists of three main sections:

- configuration - containing general information about benchmark,
- nodes - containing information about particular EC2 instances' configurations,
- plots - containing information on how to present benchmark output.

In the first section of `configuration.yml`, the user specifies general information about the benchmark, such as name, a short description, and a directory in the repository where the files are stored. Additionally, they must define how often the benchmark will be executed using CRON expression and how many EC2 instances will be needed for benchmark execution. Finally, the user specifies for which instances the benchmark should be performed. It can be defined in two ways: using a list of EC2 instances or lists of instance tags. These tags contain information about, for example, vCPUs, memory size, network bandwidth, and disks. Users can define multiple lists to execute the benchmark for instances with different properties. In order to run a test for a given instance, the instance must have all the tags specified in one of the lists in the configuration. The example of the configuration section is presented in Listing 3.1.

```
configuration:
    name: Example benchmark name
    description: Example benchmark description
    directory: example
    cron: 0 */4 * * *
    instance-number: 2
    instance-types:
      - t2.micro
      - t3.micro
    instance-tags:
      - - 1 GiB Memory
        - 1 vCPUs
      - - 0.5 GiB Memory
        - 1 vCPUs
```

Listing 3.1: configuration.yml: Example of configuration section.

The second section of `configuration.yml` is the nodes section presented in Listing 3.2. It is responsible for preparing nodes and defining commands run on particular instances during benchmark execution.

Firstly, the user must specify an ID for each node. The node with an ID equal to 0 is optional and is a default configuration. When a user does not specify a node's value (or an entire node), the service automatically uses the value from the default

```
nodes:
  - node-id: 0
    ansible-configuration: ansible-0.yaml
    output-command: python3 format_output.py benchmark_output.txt

  - node-id: 1
    ansible-configuration: ansible-1.yaml
    benchmark-command: ./benchmark-1 > benchmark_output.txt
    instance-type: c7gn.16xlarge
    image-x86: ubuntu-with-benchmark-iso-x86
    image-arm: ubuntu-with-benchmark-iso-arm

  - node-id: 2
    benchmark-command: ./benchmark > benchmark_output.txt
```

Listing 3.2: configuration.yml: Example of nodes section.

node configuration during benchmark execution. If a value is specified both in a node configuration and the default configuration, the node's value overrides the default value.

Later, the user can specify the file with Ansible configuration to run on a node. The detailed use of Ansible by the benchmark service is described in Section 3.3.1. Afterward, the user specifies a command that executes the proper benchmark. For example, this command can run a compiled C++ program or Bash script running multiple commands. The last step is defining an output command, which is run after benchmark execution. The output field is not required for each node, but at least one node must specify it. It should transform the benchmark output from a given node to the JSON object. The JSONs produced by different nodes are read and merged into a single JSON by the benchmark service. The combined JSON's fields are used in the plot section of `configuration.yml` to create data series.

Additionally, the user can specify the constant instance type of a particular node that is different from the tested instance. This feature can be used, for example, to set up the node, which will generate network traffic during benchmark execution. In order to create the same environment for comparing different instances, the program that generates load must always run on the same type of machine.

The configuration file also allows for specifying images (x86 and ARM) that should be used to run the benchmark. This functionality is added in order to, for example, run benchmarks protected with licenses like the SPEC CPU benchmark [43]. Sharing benchmark files in a public repository and enabling users to run this test locally would

```
plots:
  – type: scatter
    title: Example scatter plot
    yaxis: Time [ms]
    series:
      – y: minimum_output_scalar
        legend: Minimum value
      – y: maximum_output_scalar
        legend: Maximum value
  – type: line
    title: Example line plot
    xaxis: Execution number
    yaxis: Time [ms]
    yaxis–log: 2
    series:
      – x: increasing_values
        y: output_array
        legend: Some name of series
      – x: some_custom_x_input
        y: other_output_array
        legend: Some other name
series–other:
  – non_plot_series
```

Listing 3.3: configuration.yml: Example of plots section.

be illegal. For this reason, if the user wants to run a benchmark using the service, it has to contact the maintainer and provide an Amazon Machine Image (AMI) with benchmark files inside. Then, the benchmark service can use these images to start a machine and run the benchmark while its files are protected from public access.

The last part of `configuration.yml` is the plot section. It describes benchmark output and groups data series into plots. The user defines the list of plot objects and describes typical plot properties like a title and axis labels. Additionally, the user can set the logarithmic scale on the y-axis and specify the logarithm base as an integer or e. While defining a data series, the user can decide to group multiple values from the benchmark on the same plot (e.g., to highlight differences between them). The user must also choose the type of plot. Currently, the service supports two types of plots: scatter plots for the output JSON fields being a single value (e.g., the maximum value of network latency) and lines for the fields being a list of values (e.g., network bandwidth over a time). While defining scatter plots, the user can only define the y-series and y-axis labels because the x-series is automatically set to the execution timestamp. This enables users to observe fluctuation in value over time. For line plots, the user has to specify both the x-axis and y-axis series and both axis labels. Additionally, suppose the user does not want to provide any custom x-series for a line plot. In that case, they can set the value to `increasing_values` in order to automatically generate x-axis values from 1 to N, where N is the number of elements in the y-series.

Moreover, there is an optional section `series-other`, which helps to inform other users about series generated by benchmark but not presented in any plot. This field enables users to, for example, provide benchmark result series that cannot be presented using any currently supported plots. The example of the plot section with two plots, each with two data series, and one non-plot series is presented in Listing 3.3.

3.3. Benchmark execution

When new benchmark files are merged into the `main` branch and information from `configuration.yml` is uploaded to the MongoDB database, the benchmarking service can detect the benchmark at the next scheduled execution. The benchmark service examines whether there are any benchmarks with CRON matching the current time every hour. If there are matching benchmarks, the service starts the procedure to recognize for which instances the benchmark should be run.

In the start-up phase (and then every week), the benchmark service calls AWS Software Development Kit (SDK) [7] to get detailed information about each EC2 instance. The benchmark service parses and extracts the most valuable information (e.g., vCPUs number, memory size, and network bandwidth) and creates tags that can be used in the configuration section of the `configuration.yml` file. After parsing, the instances' data are saved to the database. The benchmark service queries this information about instances after detecting the benchmark to run. If the benchmark contains tag lists in its configuration, the service must identify all instances with the given tags.

The benchmark service prepares node configurations after identifying the benchmarks and instances on which they should be executed. In this step, the benchmark service merges the default and particular node configurations. Additionally, in this step, the service recognizes if there are nodes with specified constant instance type or AMIs. As a result, the benchmark service has a full view of AWS services and their configuration that need to be created.

As the first step of executing the benchmark, the service setup required AWS architecture using AWS SDK based on created node configurations. The following resources are created during the service start-up phase if their IDs are not specified as parameters in the Spring configuration file (`application.yml`):

- Virtual Private Cloud (VPC)[10] - to create AWS resources in an isolated virtual network,
- private IPv4 and public IPv6 addresses[6] - to enable communication between instances and with the benchmark service in VPC,
- Internet Gateway[11] - to enable network traffic to the Internet (e.g., GitHub to download benchmark files),
- Routing Table[12] - to set routing via Internet Gateway.

During executing any benchmark, the service has to create the following AWS resources:

- Subnet[14] - to enable assigning IPv4 and IPv6 addresses to the instances and separate different benchmark executions,
- Security Group[13] - to specify allowed ingress and egress network traffic,
- EC2[5] - to create machines that will be used during benchmark.

The service uses EC2 Spot Instances[9] to run benchmarks. Spot instances can be a few times cheaper than regular on-demand instances because they use a cloud provider's spare capacity. Nevertheless, when demand for on-demand instances is higher again, the customer may receive a signal that their instance will be interrupted in two minutes. However, most benchmarks do not require a long processing time, so the probability of termination by AWS is low, but the financial gain is significant.

Nevertheless, if there is no spare capacity while sending the spot instance request, AWS does not fulfill it. For this reason, we set a request "Valid Until" parameter for only 1 minute to limit waiting time. If, during that time, the request is not fulfilled, the service starts the on-demand instance instead. Using on-demand instances causes a higher cost, but it guarantees that the instances are started. It allows for a benchmark execution for all instances. Some instances are rarely available as spot instances, and gathering benchmark data for them is only possible with that mechanism.

An additional aspect while managing a large number of the EC2 instances is Service Quotas[8]. The AWS limits the number of vCPUs used by one user. When AWS detects that fulfilling the spot request would exceed the limit, AWS responds to the user with an error. The same limitation involves our service. While starting a new benchmark, the spot instance request fails if currently running benchmarks use nearly the maximal number of resources. For this reason, the service monitors the number of available vCPUs and sends the request only when there are enough resources to start all required machines for the given benchmark execution.

When the entire infrastructure is ready, the service connects to each node using SSH. After establishing a connection between the service and the nodes, the benchmark service executes commands, preparing the environment for running benchmarks. Firstly, it downloads benchmark files from its directory in the GitHub repository. Then, if the Ansible file is specified, it is launched. As a part of Ansible, the user can, for example, download required dependencies or compile files into the executable. Finally, `/etc/hosts` files are populated with the IP addresses of the nodes to enable communication between nodes. Additionally, the environmental variable indicating the ID of a node is set to enable the node's self-identification.

When all setup commands are finished successfully, the service can execute the proper benchmark. It runs the benchmark command specified for each node in `configuration.yml` and waits for its completion. After benchmark execution, the service runs the command responsible for formatting benchmark output to JSON with field names specified in the plots section of `configuration.yml`. This command should print the produced JSON to the console to enable the benchmark service to read it. Later, the service merges the JSONs produced by each node. This result JSON with added timestamp is saved to the MongoDB database in a document corresponding to the instance on which the benchmark was executed. Also, the service terminates the whole created infrastructure to eliminate further costs.

3.3.1. Ansible

Ansible[41] is a widely used open-source automation tool for configuring virtual machines. In order to use Ansible during benchmark execution, the user has to create a YAML file called playbook in which they define tasks. Additionally, the user has to assign the playbook to a node in `configuration.yml` using the node's field `ansible-configuration`. While setup phase, Ansible executes the defined tasks for nodes specified in `configuration.yml`. Listing 3.4 presents an example of such a playbook, which downloads g++ and then uses it to compile a benchmark file.

Using Ansible for benchmark configuration simplifies the process from both user and service perspectives. The user can describe all configuration steps required to run their benchmark with a widely use tool or, nowadays, even try to generate it with tools like ChatGPT. From the service perspective, it is also valuable because it simplifies implementation and maintenance. In order to run Ansible, the service has to call only one command, and Ansible handles the rest of the configuration process. Consequently,

```
---
- name: Compile C++ Benchmark Program
  hosts: localhost
  tasks:
    - name: Install g++
      become: true
      package:
        name: g++
        state: present
    - name: Compile benchmark.cpp
      shell: g++ -o benchmark benchmark.cpp
```

Listing 3.4: Example of Ansible Playbook.

there is no need to maintain its configuration system, but it can use an existing one with multiple implemented and advanced functions.

In the specific case of our benchmark service, the service runs Ansible locally on the instances to move processing from the service benchmark to the nodes. This solution has one disadvantage - all nodes need to have installed Ansible. For this reason, we use custom Ubuntu images with installed Ansible beforehand.

3.3.2. MongoDB

The critical element of the service is MongoDB[37], which is the database storing all data about instances, benchmarks, and benchmark results. The main reason why we decided to use MongoDB is that it is schemaless. The benchmarks have diverse output with different names and types, like double, integers, or lists of them. Nevertheless, MongoDB's schemaless property enables saving the results without knowing the exact schema. This enables the service to easily save and extract results that mix fields with single values and a list of values as key-value maps. However, despite MongoDB's schemaless, the web application creates schema based on data from the `configuration.yml` file. This allows us to work with benchmark results without knowing the exact schema, but we can use it when needed to present results as plots.

Moreover, many values stored by the service are nested lists of objects. Each document describing an EC2 instance contains a list of benchmarks. Each such benchmark object contains a list of objects representing the benchmark execution results, and each result object may contain multiple lists. An example of MongoDB's document storing instance data can be found in the appendix (Appendix A.3). The typical SQL database would require sending multiple queries or one query with many join operators, which makes extracting and saving data more complex. By contrast, because the service uses MongoDB, the benchmark service can get and save all required data with a straightforward request, making development and maintenance easier.

Furthermore, MongoDB also has typical database functionalities like views and indexes. The benchmark service creates an index on the instance name to improve query performances because most of the queries filter on this field. Additionally, the service creates three views using MongoDB's Aggregation Pipeline[36]. This enables MongoDB to perform operations on the input documents and expose output as views. These views expose data that are used for plot visualization and which users can query using DuckDB (Section 3.4.3).

3.3.3. Implementation details

The benchmark service is implemented using Kotlin and Spring Boot with Spring WebFlux and Spring Data modules. Kotlin[29] is a modern JVM language developed by JetBrains. Thanks to its full interoperability with Java, it can use various Java libraries. However, it also provides many additional features, making development more straightforward and safer, like null safety and extension functions. The main reason for choosing Kotlin for the benchmark service implementation is coroutines[28]. The executing benchmarks require many I/O operations, like database queries or communication over SSH. Coroutines enable the service to process multiple benchmark executions in a highly concurrent manner. Coroutines can be suspended during I/O operations and resumed later on any thread. While one coroutine is suspended and waits for data, the service can continue processing other concurrent coroutines and thus proceed with executing other benchmarks.

The benchmark service, besides running benchmarks, also exposes one internal endpoint. Its purpose is to simplify the deployment of the new service version by preventing the subsequent benchmark executions from being scheduled. When the maintainer wants to deploy the new version, they must connect to the instance that runs the service and send an HTTP request to localhost. The request changes the flag in the service, which causes the function that runs benchmarks to never be called again. Afterward, the maintainer can deploy the new service version to handle future benchmark executions. Then, when the maintainer observes that all benchmark finished their execution based on logs, they can terminate the old service instance. This enables the maintainer to update the service without terminating benchmarks during execution and allows the service to delete its AWS resources.

The benchmark service uses Apache MINA SSHD library [47] for communication over SSH with created EC2 instances. It is a subproject in Apache MINA that provides functionalities to build SSH clients and servers in Java applications. The service uses it to communicate with the EC2 instances and call commands that prepare the environment and run benchmarks.

3.4. Result presentation

The web application is the second part of the service with which users can interact. Its primary purpose is to allow users to filter and sort instances based on benchmark results. Users can do that with the aid of predefined filters or a query console. The filters are helpful for simple queries focused on one benchmark. However, the console allows users to write complex queries to get any data in which they are interested. Additionally, the web page can visualize the plots with results according to the definitions in `configuration.yml`. Moreover, the visualization is not limited to a single instance. Users can also choose multiple instances to compare and display their benchmark results on the same plot. The web application consists of a back-end Spring Boot application for communication with the database and a front-end Angular application for data presentation. In the following sections, we will focus on a detailed presentation of each part of the web application.

3.4.1. Back-end application

The back-end application is separated from the benchmark service for easier maintenance and separation of functionalities. Similarly to the benchmark service, this application is written with Kotlin and Spring Boot with Spring WebFlux and Spring Data modules. The back-end application is responsible for responding to front-end application requests and returning data stored in MongoDB. The service accesses three views created in MongoDB while starting the benchmark service to answer the front-end requests. Thanks to using the views, the complex preprocessing is hidden from the service and done in the database.

Additionally, this service caches data using the Caffeine Cache library[17]. The back-end service uses the cache to improve response times by eliminating the need to query the database for each request. This is possible and useful because the data are changing infrequently, and responding with their newest version is not critical.

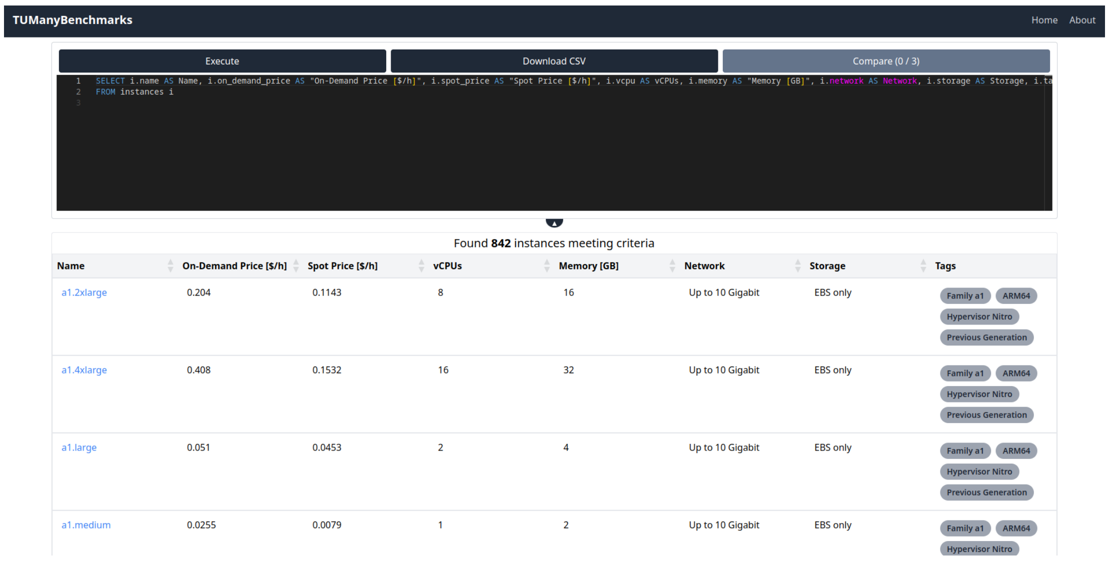
The last functionality of the back-end application is to supply instance information from the database with data about their prices. Currently, the service only supports on-demand and spot instance prices from the `us-east-1` region. At the start-up phase, the application calls AWS APIs and caches received prices. Afterward, the service repeated calls every hour to store up-to-date prices. The service has to frequently update prices because spot prices change over time. Additionally, while calling the spot instance price API, AWS responds with prices in all availability zones in the given region. However, our back-end service returns only the lowest price from them for simplicity. The data are stored in the cache, not the MongoDB database, because they change frequently and can be quickly restored.

3.4.2. List view

The list view is the most crucial part of the web application. The homepage view allows users to look for instances, sort them, and see information about their benchmark results. It works in two modes: a simple set of filters or a query console.

The first mode allows for a fast lookup of basic information about instances and their benchmark results' statistics. While using this mode, the user can filter instances by the basic instance properties like name, on-demand price, spot price, vCPU, memory, network bandwidth, and tags. Additionally, they can choose one of the data series of benchmarks from a dropdown list. After selecting the series, the user sees only instances with the results of the selected benchmark. Moreover, the statistics of the picked benchmark series are presented to the user in addition to the previously mentioned basic properties. This enables the user to quickly check and sort statistics like minimum, maximum, average, and median values.

The second mode, the query console, allows for extracting more detailed information. It allows the user to interact with the DuckDB Wasm[30], which is initialized with a copy of data from MongoDB's views. After opening the console, the current filter is automatically translated into a SQL query. Later, the user can change the query according to their will. Upon preparing the query and clicking the "Execute" button, the query is passed to the DuckDB. DuckDB executes it locally, and then the web application shows results in place of the previous data filtering results. The user also has the option to download the displayed data as CSV. Additionally, if the inserted query is incorrect, the application will display the error returned by the database to help the user fix the query.



```
SELECT i.name AS Name, i.on_demand_price AS "On-Demand Price [$/h]", i.spot_price AS "Spot Price [$/h]", i.vcpu AS vCPUs, i.memory AS "Memory [GB]", i.network AS Network, i.storage AS Storage, i.ta
FROM instances i
```

Found **842** instances meeting criteria

Name	On-Demand Price [$/h]	Spot Price [$/h]	vCPUs	Memory [GB]	Network	Storage	Tags
a1.2xlarge	0.204	0.1143	8	16	Up to 10 Gigabit	EBS only	Family a1, ARM64, Hypervisor Nitro, Previous Generation
a1.4xlarge	0.408	0.1532	16	32	Up to 10 Gigabit	EBS only	Family a1, ARM64, Hypervisor Nitro, Previous Generation
a1.large	0.051	0.0453	2	4	Up to 10 Gigabit	EBS only	Family a1, ARM64, Hypervisor Nitro, Previous Generation
a1.medium	0.0255	0.0079	1	2	Up to 10 Gigabit	EBS only	Family a1, ARM64, Hypervisor Nitro

Figure 3.2.: Web application: Instance list with query console.

3.4.3. Querying data with DuckDB

DuckDB [40] is a high-performance database for analytical queries. Due to the fact that it is available as Web Assembly[30], it can be used in the browser. DuckDB Wasm enables the user to run queries locally, which does not affect the performance of the central database and allows for greater freedom. The user works on a local copy of the database, allowing them to create and modify tables and views at their own will.

Additionally, DuckDB has an extension for JSON files [22]. This extension enables the front-end application to import data from MongoDB without additional operation on the user side. Furthermore, the extension also adds support for JSON column type, which allows nested objects to be stored as JSONs and simplifies retrieving individual values from them. The JSON column type enables users to access JSON fields without splitting them into separate columns, which would create a much more complex schema. Instead, the user can access data within JSONs using -> or ->> operators, which returns data as JSON and VARCHAR, respectively, or lists of those types.

Additionally, DuckDB SQL is compatible with PostgreSQL syntax. PostgreSQL is an industry and academic standard, so users interested in hardware metrics should know how to use it to query data. SQL provides users with simple access to data without additional knowledge, unlike other much less commonly used query languages, such as MongoDB Aggregation Pipeline.

In MongoDB, all information is stored in two collections: `instances` and `benchmarks`. However, to make access to basic information about benchmark results more straightforward, we defined three views in MongoDB. These views hide complex data operations from the user and transform data to JSONs, which can be loaded into simple SQL tables in DuckDB. The data from these views are copied to three tables in DuckDB:

- `instances` - a table with basic information about instances and with detailed benchmark results as JSONs,
- `benchmarks` - a table extracting the most important information about benchmarks,
- `statistics` - a table with precomputed values of statistics (minimum, maximum, average, and median) for each data series generated by benchmarks for each instance.

Detailed information about the SQL tables structure and their relations is presented in Figure 3.3.

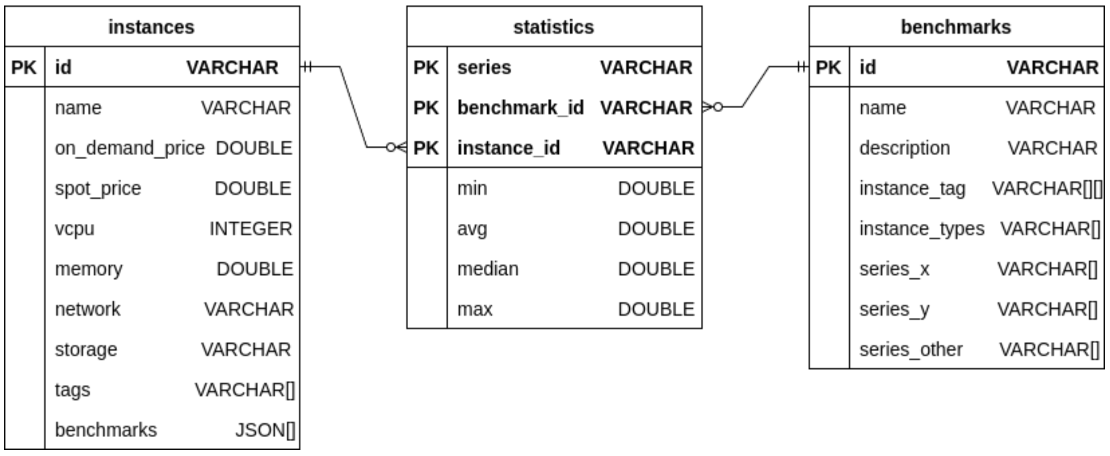
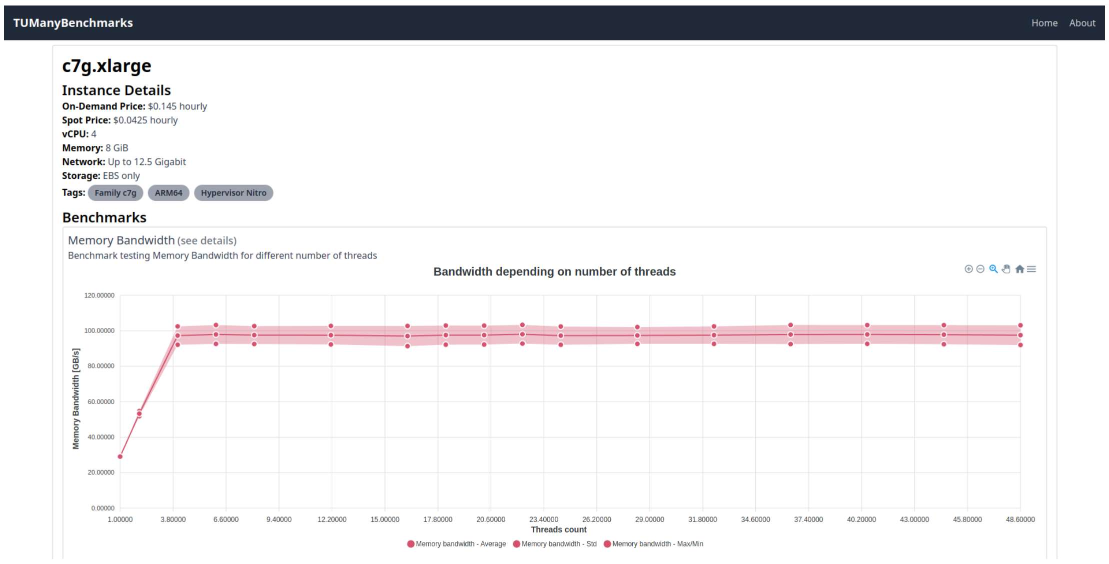


Figure 3.3.: DuckDB database schema.

3.4.4. Details view

In the details view, the user can see basic information about the selected instance together with its benchmark results. The benchmark results are presented on the plots based on the specifications from the `configuration.yml` files. The user can go to this view by clicking on the instance's name in the instance list view. This feature also works for query results if the result contains a column called `name`. As mentioned in Section 3.2.1, the service supports the presentation of two plot types: scatter and line. The


TUManyBenchmarks

Home About

c7g.xlarge

Instance Details

On-Demand Price: $0.145 hourly
Spot Price: $0.0425 hourly
vCPU: 4
Memory: 8 GiB
Network: Up to 12.5 Gigabit
Storage: EBS only
Tags: Family c7g ARM64 Hypervisor Nitro

Benchmarks

Memory Bandwidth (see details)
Benchmark testing Memory Bandwidth for different number of threads

Figure 3.4.: Web application: Instance details view.

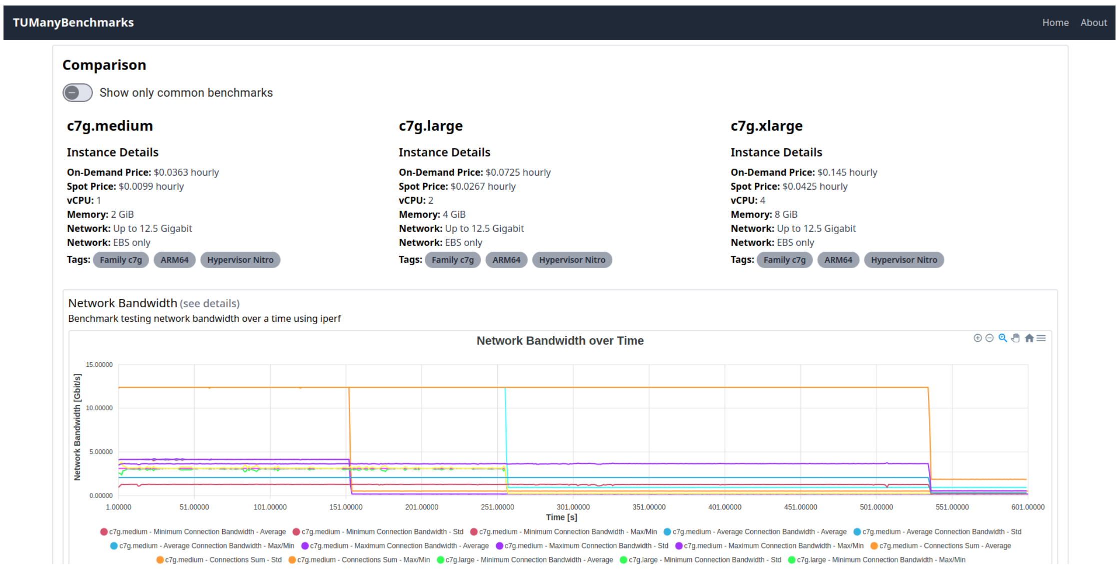
web application uses ApexChart.js [16], an open-source library for creating interactive plots. The library displays plots that automatically have implemented functions such as zooming data, hiding chosen series, and exporting data, which makes the data analysis more comfortable for a user.

When a benchmark result has a series that contains a single value per benchmark execution (e.g., the maximal value of network latency), a user should use a scatter plot for data presentation. In such a case, the user can specify only the y-axis label. The x-axis value is always a timestamp indicating when the benchmark was executed. This enables users to analyze the fluctuation of the benchmark value over time and observe, for example, daily patterns (e.g., higher values on nights) and weekly patterns (higher values on weekends) caused by noisy neighbors.

The second type of plot is a line plot, which is for a series containing multiple values per benchmark execution (e.g., the network bandwidth value over time). In the case of this plot type, a user can specify the x-axis and y-axis values and both labels. However, it would be impossible to present value fluctuation clearly if the plot contained all benchmark results. For this reason, the line plots show average values with marked areas indicating a standard deviation and minimal and maximum values, providing the user with a good overview of values and their fluctuations.

3.4.5. Comparison view

The last view that users can use is the comparison view. From the instance list view, the user can select up to three types of instances for comparison by clicking on them. This feature also works for query results if the result contains a column called `name`. After choosing the instances and redirecting to the comparison view, the user sees a view similar to the details view with presented benchmark results. However, in this view, if the same benchmark is executed on multiple instances, their results will be shown together on one plot. This makes it easy to compare metrics between different instances and conclude from them. Moreover, the view has a switch that limits the displayed benchmarks only to those common to all instances in the comparison. This feature allows the user to focus on comparable metrics.



Comparison

Show only common benchmarks

c7g.medium

Instance Details

On-Demand Price: $0.0363 hourly
Spot Price: $0.0099 hourly
vCPU: 1
Memory: 2 GiB
Network: Up to 12.5 Gigabit
Network: EBS only
Tags: Family c7g ARM64 Hypervisor Nitro

c7g.large

Instance Details

On-Demand Price: $0.0725 hourly
Spot Price: $0.0267 hourly
vCPU: 2
Memory: 4 GiB
Network: Up to 12.5 Gigabit
Network: EBS only
Tags: Family c7g ARM64 Hypervisor Nitro

c7g.xlarge

Instance Details

On-Demand Price: $0.145 hourly
Spot Price: $0.0425 hourly
vCPU: 4
Memory: 8 GiB
Network: Up to 12.5 Gigabit
Network: EBS only
Tags: Family c7g ARM64 Hypervisor Nitro

Network Bandwidth (see details)
Benchmark testing network bandwidth over a time using iperf

Network Bandwidth over Time
15.00000
10.00000
5.00000
0.00000
Network Bandwidth [Gbit/s]
1.00000
51.00000
101.00000
151.00000
201.00000
251.00000
301.00000
351.00000
401.00000
451.00000
501.00000
551.00000
601.00000
Time [s]
● c7g.medium - Minimum Connection Bandwidth - Average
● c7g.medium - Minimum Connection Bandwidth - Std
● c7g.medium - Minimum Connection Bandwidth - Max/Min
● c7g.medium - Average Connection Bandwidth - Average
● c7g.medium - Average Connection Bandwidth - Std
● c7g.medium - Average Connection Bandwidth - Max/Min
● c7g.medium - Maximum Connection Bandwidth - Average
● c7g.medium - Maximum Connection Bandwidth - Std
● c7g.medium - Maximum Connection Bandwidth - Max/Min
● c7g.medium - Connections Sum - Average
● c7g.medium - Connections Sum - Std
● c7g.medium - Connections Sum - Max/Min
● c7g.large - Minimum Connection Bandwidth - Average
● c7g.large - Minimum Connection Bandwidth - Std
● c7g.large - Minimum Connection Bandwidth - Max/Min

Figure 3.5.: Web application: Instance comparison view.

4. Evaluation

This chapter focuses on the evaluation of TUManyBenchmarks. Firstly, we will present some of the observations about the EC2 instance's performance that we were able to highlight due to benchmarking of hardware metrics. Then, we will present some example cost-efficiency models which can be created using our system. Afterward, we will examine how TUManyBenchmarks simplifies the benchmarking. Lastly, we will revisit the motivations and requirements stated in Chapters 1 and 3.

4.1. Example observations on hardware metrics

One of the main goals of developing TUManyBenchamrk was to allow the user to access hardware metrics. These metrics enable them to get a better view of the instance than information provided by cloud providers. This section will present some example observations that can be discovered by accessing the detailed benchmark result collected using TUManyBenchmarks.

4.1.1. Changes in network bandwidth over time

AWS describes network bandwidth for small instances using baseline and burst bandwidth. The baseline bandwidth is proportional to the bandwidth of the metal instance and the number of cores of the given instance ($\frac{metal\_network\_bandwidth}{\#metal\_instance\_vcpus} * \#instance\_vcpus$). However, this value is low for instances with only a few vCPUs, such as medium and large instances. For this reason, AWS provides burst bandwidth - a high network bandwidth available for a limited time, usually described as "Up to X Gbits/s." However, the possibility of using burst bandwidth leads to oversubscription of the network bandwidth. Therefore, the burst bandwidth can be used only for a limited amount of time. The AWS uses a bucket token algorithm to manage available burst bandwidth. The instance starts with the assigned maximal number of tokens and uses them when utilizing the network over the baseline bandwidth. Then, the bandwidth is limited to the baseline value when the instance spends all its tokens. However, when the instance does not utilize the baseline bandwidth fully, it earns additional tokens, which can be used later for the burst.

Nevertheless, AWS does not specify the details of this algorithm for each instance, such as the maximal number of tokens. For this reason, it is not transparent to users how much of the burst is available while using the given instance and, consequently, how much data they can download with it. However, this information may be crucial for achieving cost efficiency for people whose processing downloads a large amount of data.

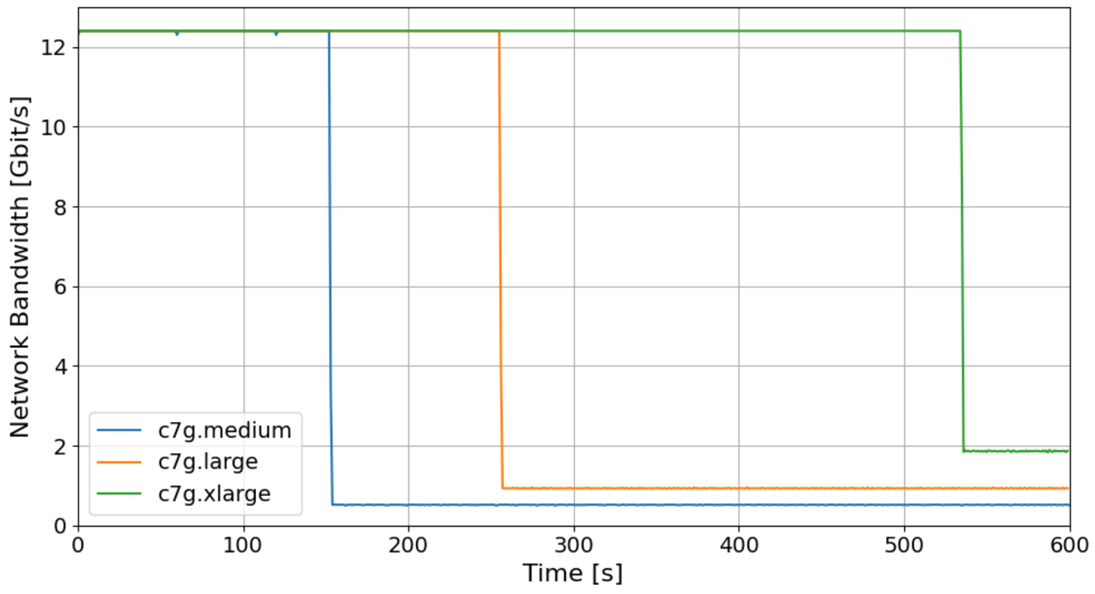


Figure 4.1.: Network bandwidth over time for small instances from c7g family.

Figure 4.1 presents bandwidth measurements made by TUManyBenchmarks using iperf[23] for three instances from the c7g family: c7g.medium, c7g.large, and c7g.xlarge. According to AWS information, all of them offer the same burst network bandwidth equal to 12.5 Gbit/s, which we can observe in the plot. After using the burst, the baseline bandwidth is, in fact, much smaller and proportional to the number of vCPUs of the given instance - around 0.52 Gbit/s for c7g.medium, 0.93 Gbit/s for c7g.large, and 1.85 Gbit/s for c7g.xlarge. The bandwidth per core is approximately equal to 0.47Gbit/s, which is consistent with the calculations - $\frac{30Gbit/s}{64} = 0.47GBit/s$

Nevertheless, we can also observe that the instances significantly differ in available burst bandwidth. The burst times equal about 150s for c7g.medium, 260s for c7g.large, and 550s for c7g.xlarge. This allows the user to download 234 GB, 406 GB, and 859 GB, respectively. However, considering only the data downloaded with bandwidth above the baseline, we utilize 224 GB, 376GB, and 730GB, respectively. We can observe that the burst increases with more vCPUs and cost, but they do not scale linearly. For example, c7g.xlarge has only 3.26 times greater download size with bandwidth above the baseline than c7g.medium, but c7g.xlarge is 4 times more expensive. For this reason, the available burst bandwidth is not transparent and benchmarking is the only way to discover the exact values.

This available burst can be crucial, for example, for users who download a large amount of data at the beginning of the processing using small instances. If they do not manage to download all the data in a time with burst bandwidth, they must download the rest using a much slower baseline bandwidth. This causes the processing to take longer and can generate high costs for small instances. For example, if the user has to download 300 GB, then the c7g.medium utilizes the whole burst bandwidth to download 234GB and then the baseline bandwidth to download the rest. The whole process will take 1166s (150s burst + 1016s baseline) and using on-demand prices from the us-east-1 region, it costs 0.012$. While using c7g.large, the instance can download all the data using the burst bandwidth in 192s, which costs only 0.0039$ despite two times higher price per hour. The difference in available bursts causes the difference in cost. If download time is the dominant time factor, it significantly reduces total processing time, which may result in lower costs. For this reason, transparency with available burst bandwidth can be important for some users.

4.1.2. Change of CPU performance over years

As described in Section 1.2, cloud providers use the number of vCPUs to illustrate the machine's performance. In the beginning, AWS used the Elastic Compute Unit to describe the performance of different instances. Nevertheless, they stopped publishing this information when the number of instances increased, and the number of vCPUs is the only remaining metric. However, this single characteristic does not provide complete transparency and does not enable users to compare different instances. The vCPUs of different machines can have different performances, which causes the instance with an X times higher number of vCPUs to not have to be X times faster.

The performance of the instance's vCPUs depends on multiple factors. First, the CPUs are produced by one of the three vendors - Intel, AMD, or AWS - which additionally base their products on different architectures. Intel and AMD operate on x86 architecture, whereas AWS Graviton uses ARM architecture. Moreover, the vendors constantly improve their products, so even the instances from the same vendor but different generations perform differently. Furthermore, some processors use hyperthreading to increase the number of vCPUs, which also impacts the performance. If the processor supports hyperthreading, the two threads can run on a single physical core, which doubles the number of vCPUs available to the machine. However, the two threads perform significantly better when they run on separate physical cores than on a single physical core due to hyperthreading. All these factors cause the performance of vCPUs from different families to vary, and comparing the instance performance only on the number of vCPUs is not transparent. For this reason, we decided to use TUManyBenchmarks to run the SPEC CPU 2017 ®[43] benchmark on instances from different generations and manufacturers to discover how the single-core and multi-core performance has changed over the years.

Single-core performance

The most fundamental processing is single-core processing. For this reason, we have run the SPECrate® 2017 Integer suite[44] (a throughput, integer-based suite) on X.large instances to analyze single-core performance trends. The results are presented in Figures 4.2 and 4.3.

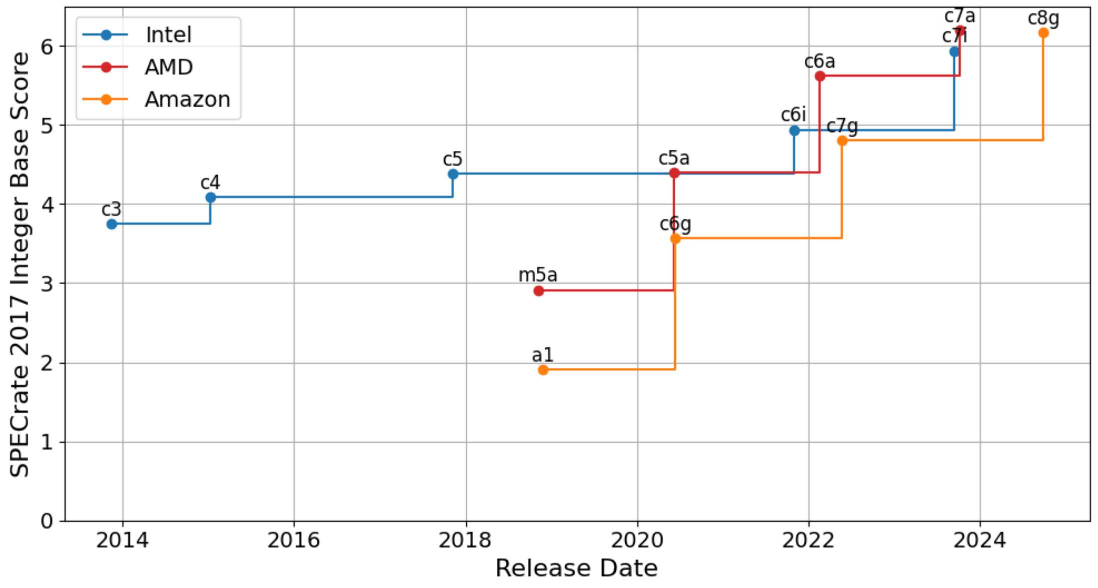


Figure 4.2.: SPECrate 2017 Integer Base values over time for single-core processing.

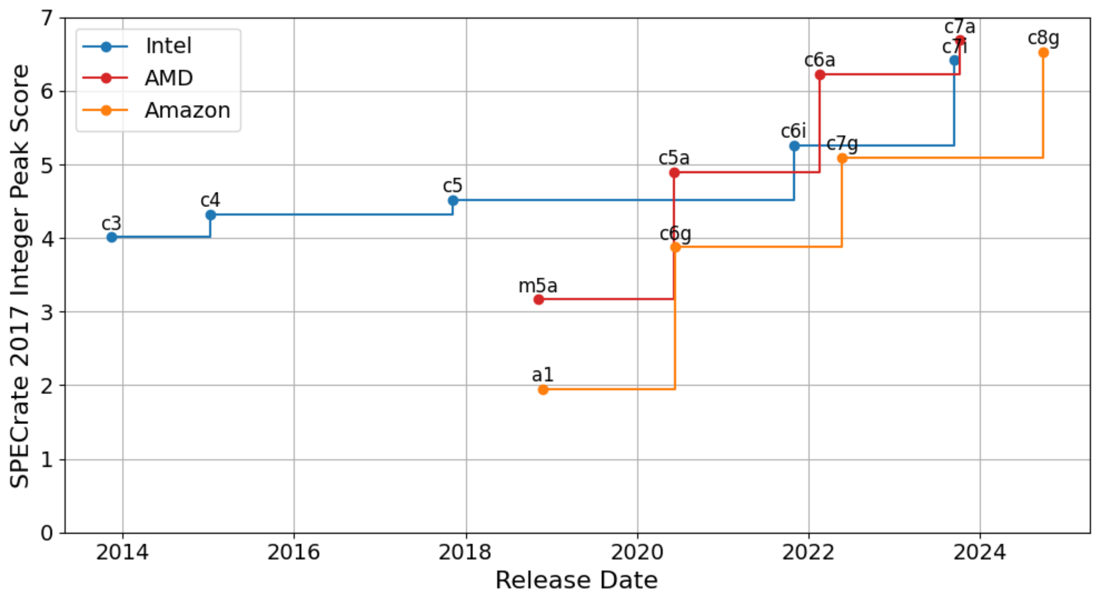


Figure 4.3.: SPECrate 2017 Integer Peak values over time for single-core processing.

In the beginning, the AWS offered only Intel instances. This changed at the end of 2018 when they introduced AMD instances and Graviton instances a few days later. However, their single-core performance was significantly worse than the performance of the available Intel instances at that point. Nevertheless, with the next generation and compute-optimized instances, the performance improved. The first AMD compute-optimized instances released in 2020 have the SPECRate Integer Base value nearly identical to Intel's but noticeably better the SPECRate Integer Peak value. The following AMD generation also characterizes excellent single-core performance, better than the other machines released at a similar time. The significant improvement with each generation is also visible for Graviton instances. The first released instances, family a1, were significantly slower than others, but the gap was becoming smaller with each new generation of compute-optimized instances. The newest c8g instances perform even better than Intel's and are only slightly worse than AMD instances. This shows that there is no significant difference between instance single-core performance from different manufacturers currently, but it was significant in the past.

The performance analysis showed that the new instances characterize better single-core performance with each generation, especially Graviton instances. However, the cost is a fundamental factor in the cloud environment to achieve cost-efficient processing. For this reason, it is essential to consider how performance per dollar changed over

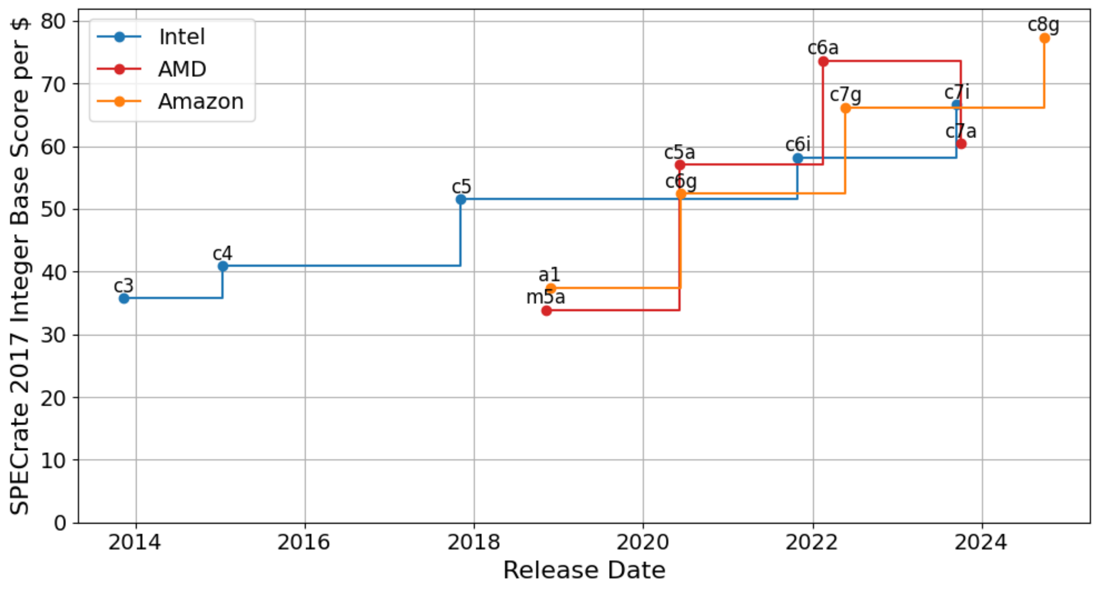


Figure 4.4.: SPECrate 2017 Integer Base per Dollar values over time for single-core processing.

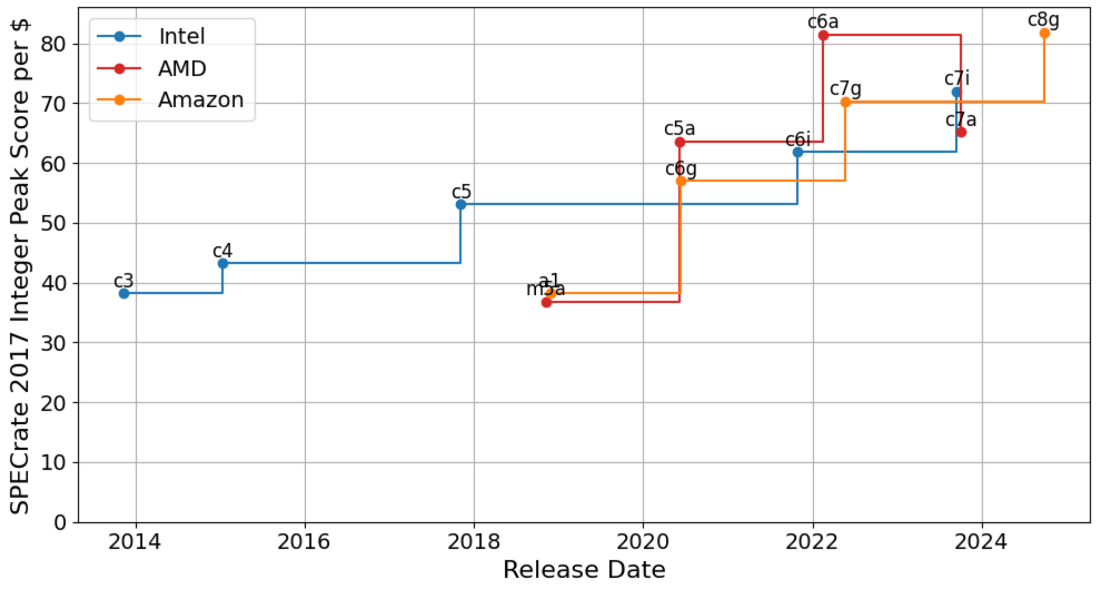


Figure 4.5.: SPECrate 2017 Integer Peak per Dollar values over time for single-core processing.

time for different instance families, as all the instances have different prices. The calculations are based on on-demand prices from us-east-1. The results are presented in Figures 4.4 and 4.5. First, we can observe that the cost-efficiency of single-core processing has improved over the years nearly all the time. The only exception is a significant drop in cost-efficiency for the c7a instance. It is caused by the fact that the performance improvement compared to the c6a instance is small, but it has a nearly 35% higher on-demand price. This makes the c7a instance much less cost-efficient regarding single-core processing cost-efficiency. Besides that, AMD compute-optimized instances were the most cost-efficient for most of the time. c6a is still the second most cost-efficient instance due to its low price and high performance. Additionally, the plots show that compute-optimized Graviton instances are better when considering cost efficiency than performance. This is caused by the fact that even though c6g and c7g are slower than Intel and AMD instances, they are much cheaper. Because of these low prices, although the c7g instance is slower than the c7i, it has the same cost-efficiency. The low price also enables c8g to be the most cost-efficient instance currently.

Multi-core performance

As we can observe in the previous section, the improvement for single-core processing over 10 years is small. For example, the Intel instance c7i is only about 50% better than the c3 - 10 years older Intel instance. Consequently, multi-core processing is becoming more and more essential nowadays. However, the vCPUs offered by AWS often are hyperthreads, which are significantly slower. For this reason, we have run the SPECrate 2017 Integer suite[44] on X.2xlarge instances to analyze multi-core performance trends and the impact of hyperthreading. The SPEC CPU throughput benchmark can be used to analyze multi-core processing by specifying the number of concurrent copies of the benchmark to run. For each instance, we executed the benchmark using half of the vCPUs (4 copies) and all vCPUs (8 copies), which allowed us to notice the difference in performance due to hyperthreading. In this section, we concentrate only on the results for the base metric. The results of the peak metric are available in the Appendix A.4 and lead to identical conclusions.

Figure 4.6 presents SPEC CPU 2017 IntRate results for multi-core processing with four copies (half of the available vCPUs). All the achieved scores are approximately four times better than the single-core processing results. Such improvement is caused by the fact that the benchmark copies are independent and executed only by independent physical cores. This allows the copies to be run in parallel, nearly without interference. For this reason, the throughput can increase approximately proportionally to the number of copies. However, the scaling is not perfectly linear because cores share the

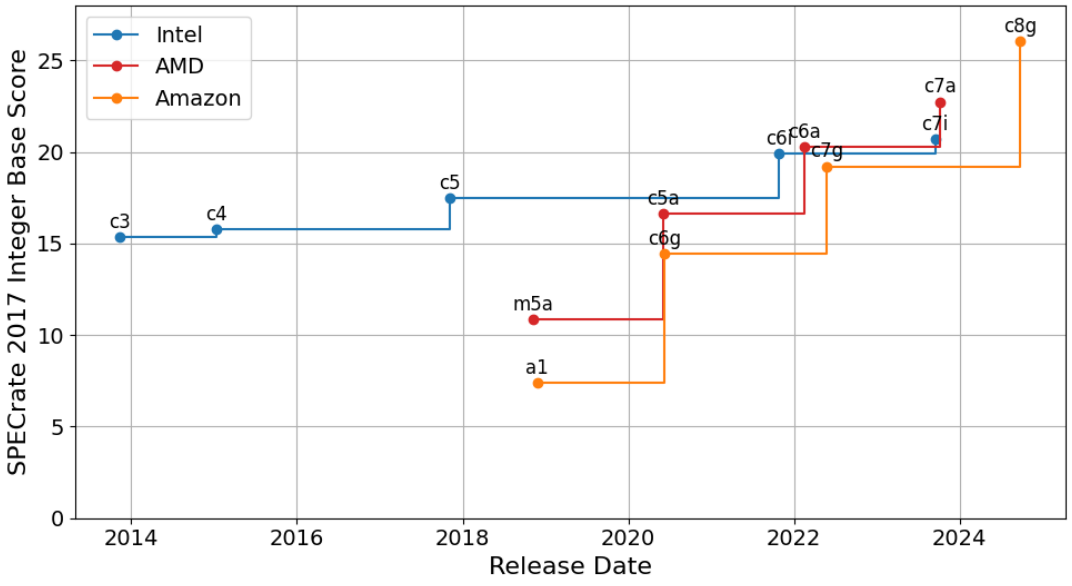


Figure 4.6.: SPECrate 2017 Integer Base values over time for multi-core processing using half of the vCPUs.

L3 cache in the multi-core benchmark, contrary to single-core when the single copy owns the whole L3 cache. This can cause a slight performance decrease, such as the one visible in the results of the c7a instance. In the case of linear scaling with the number of cores, the instance should achieve a score equal to around 25, but instead, it achieved only 22.7.

If we focus on cost efficiency, we can observe that the results presented in Figure 4.7 are nearly identical to those for single-core processing. This is caused by the fact that the benchmark results are approximately four times better. On the other hand, to calculate the score per dollar, we are using X.2xlarge instances instead of X.large, which are four times more expensive instances. For this reason, there is no significant difference between the results concerning cost efficiency for four copies.

The changes between the single-core processing and the processing using only half of the available vCPUs are the same for all instances and easy to foresee - the more cores, the higher the throughput. However, we observe noticeable differences compared to the previous cases when the SPEC CPU benchmark utilizes all vCPUs, as presented in Figure 4.8. We can divide the instances into two groups. The first group consists of all Intel and AMD instances except c7a.2xlarge. These instances characterize

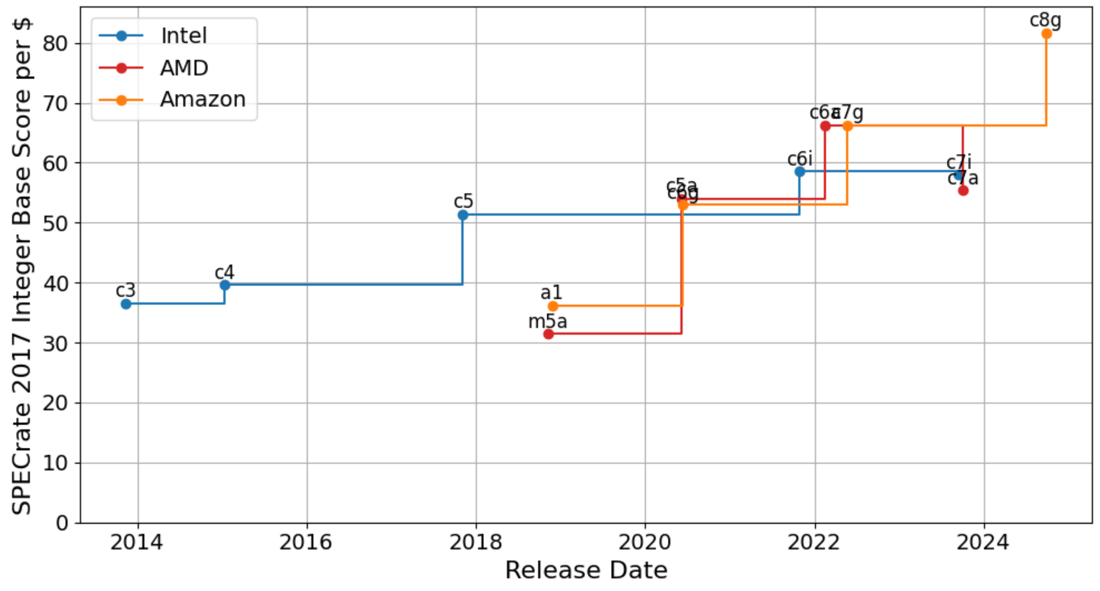


Figure 4.7.: SPECrate 2017 Integer Base per Dollar values over time for multi-core processing using half of the vCPUs.

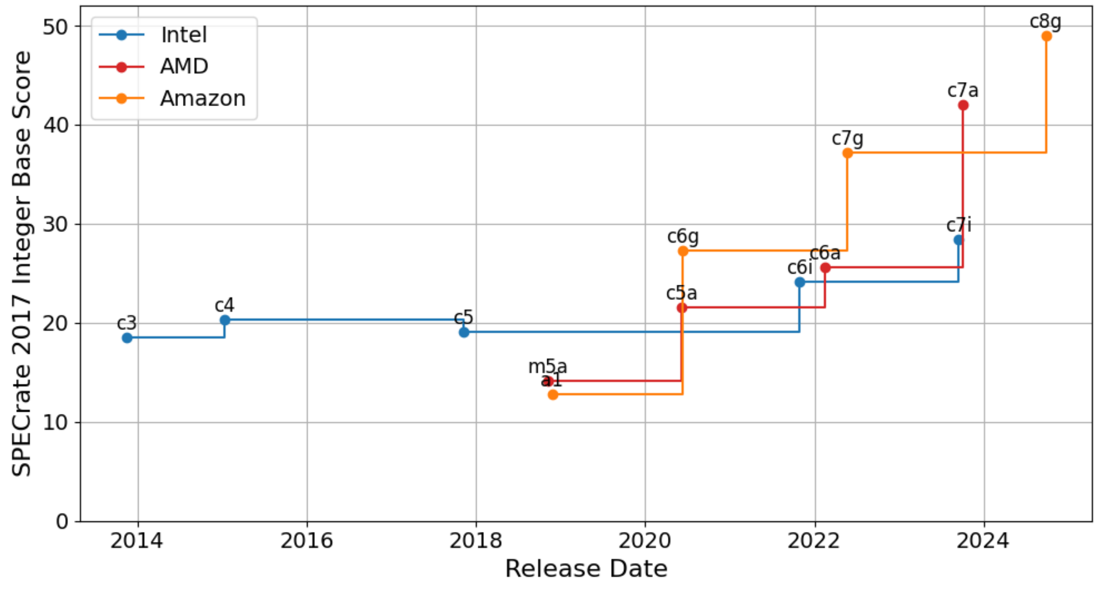


Figure 4.8.: SPECrate 2017 Integer Base values over time for multi-core processing using all vCPUs.

a slight improvement, equal to about 25% of the score for the processing using half of the vCPUs. Another group consists of Graviton and c7a.2xlarge instances, which characterize further approximately linear throughput increase with the increase of vCPUs. This difference between instances is caused by hyperthreading. The instances in the first group utilize hyperthreads, which account for half of their vCPUs. This means that the two benchmark copies are executed on a single physical core. Consequently, the additional hypthread is significantly slower, and its performance equals about 25% of the regular thread. The instances without hyperthreading offer regular cores, and processing does not differ from the case with four copies for them - it increases linearly with the number of vCPUs. This difference causes the instances without hyperthreading to offer a higher throughput while utilizing all cores, even if they have noticeably slower single-core processing like the c6g instance. However, simultaneously, the advancement of single-core processing of Graviton instances over the years translates to a significant improvement in throughput for multi-core processing. For this reason, with each new generation, the gap between AWS and other manufacturers becomes bigger.

The impact of vCPUs that are actually hyperthreads is the best visible for c6a.2xlarge and c7a.2xlarge and is illustrated in Figure 4.9. Both instances guarantee a similar single-core performance and scale approximately linearly with more vCPUs as long as they operate on physical cores. However, when the c6a.2xlarge starts using hyperthreads, adding four vCPUs equals adding one physical core before. c7a.2xlarge, which uses only physical cores, improves linearly further and achieves much higher throughput. This is the reason why the c7a instance is significantly more expensive and, consequently, offers worse cost-efficiency for single-core processing, as observed in Figures 4.4 and 4.5. However, when the c7a instance uses all available cores, its throughput is nearly two times better. Therefore, it can significantly improve its cost efficiency compared to the c6a instance as shown in Figure 4.10.

c7a.2xlarge is not the only instance that achieves much better cost efficiency using all the vCPUs. Graviton instances offer the best cost efficiency while using all vCPUs because they offer the highest throughput at the lowest prices. On the other hand, the instances that use hyperthreading do not significantly improve their cost efficiency compared to the previous cases due to only a slight increase in throughput. We can also observe that for Intel instances, the cost efficiency only doubled over 10 years. Contrary to Intel, the Graviton instances have developed a little greater progress in a much shorter time. The initial Graviton instances' cost efficiency was comparable to the Intel c5 instance. However, it enhanced with each generation due to improvements in single-core processing. For this reason, the Graviton cost efficiency is now 2.6 times greater than six years earlier.

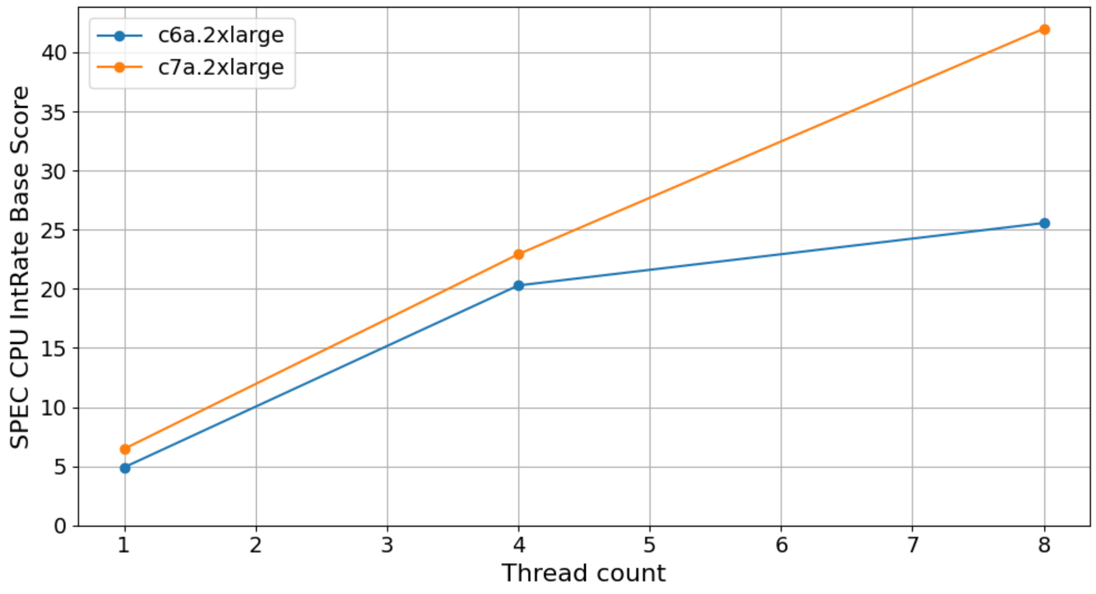


Figure 4.9.: SPECrate 2017 Integer Base values for different number threads for c6a.2xlarge (hyperthreading) and c7a.2xlarge (no hyperthreading).

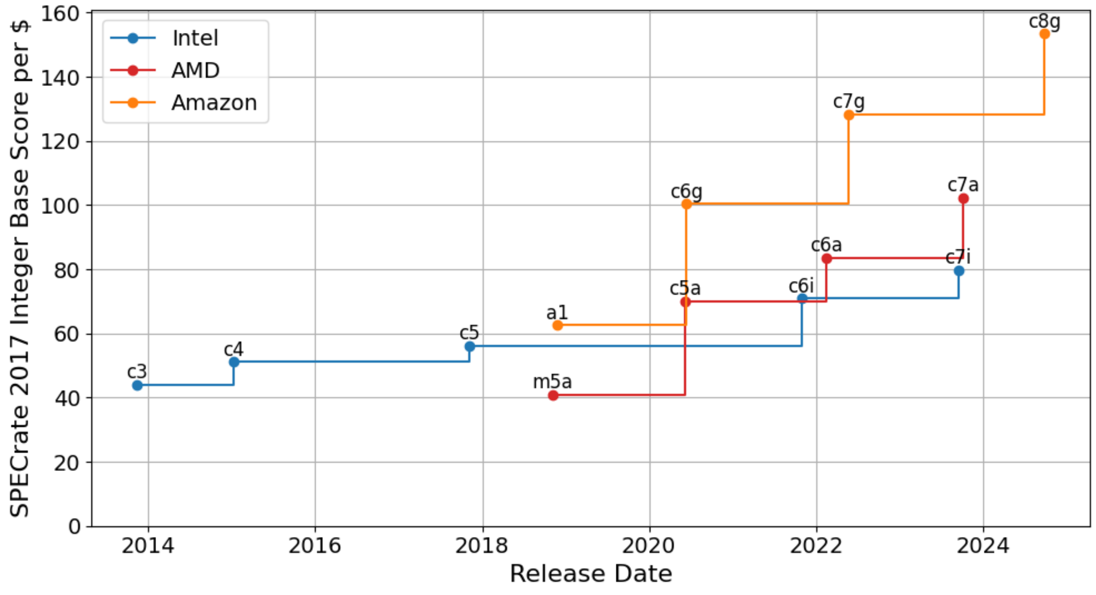


Figure 4.10.: SPECrate 2017 Integer Base per Dollar values over time for multi-core processing using all vCPUs.

All the presented observations prove how important proper benchmarking is. Based on the data provided by cloud providers, the differences between instances can seem tiny. However, factors such as various manufacturers, architectures, generations, and hyperthreading make it challenging to compare in practice. For this reason, the only way to transparently compare the instances is to depend on the benchmark results from different instances, like the ones collected by TUManyBenchmarks. Then, the user can analyze these data to transparently choose the proper instance for their use case to achieve cost efficiency.

4.1.3. Cache Allocation Technology in the cloud

The central resource offered by cloud providers is CPUs. This resource is bought by clients who need a machine with a dozen cores but also small clients who need machines with only a few cores. For this reason, the cloud providers have to offer instances of different sizes. However, the servers bought by them often have even more than a hundred cores. For this reason, cloud providers must split their machines into smaller virtual machines. This causes its resources to be shared between multiple clients. For this reason, the user can observe such phenomena as noisy neighbors in which the instance that intensively uses resources impacts the performance of others with which the hardware is shared.

To reduce the influence of instances on each other, the manufacturers developed technologies such as Cache Allocation Technology (CAT)[38]. In modern CPUs, the L1 and L2 caches are private for each physical core. However, the L3 cache is usually shared between multiple cores, which causes it to be shared between a few instances in the cloud environment. CAT allows control of the L3 cache size assigned to the virtual machines. Consequently, the cloud providers can assign less L3 cache to smaller instances to achieve a fairer share of resources and more predictable performance.

However, some details about this technology and its influence on performance are nontransparent for cloud customers. CAT is a new technology, so it is not available for all processors on which instances run. Additionally, even if CAT is available, the cloud providers are not obligated to use it. Moreover, CAT is an Intel technology, but AMD has a similar equivalent called Quality of Service (QoS)[15]. However, there is no detailed information about using similar technology in AWS Graviton processors. For this reason, the behavior can differ between CPUs of different manufacturers. Therefore, benchmarking different instances is crucial to provide transparency and to discover whether AWS uses such technologies and how they impact performance for different CPU manufacturers. We have used TUManyBenchmarks to examine it using the cache latency benchmark.

Intel instances

For the Intel instances, we concentrate on instances from the c5 and c6i families. Their results are illustrated in Figures 4.11 and 4.12 respectively. During tests, we observed that the small instances of the c5 family can use the Intel Xeon Platinum 8124M or Intel Xeon Platinum 8275CL processors. For this analysis, we selected only the results of the c5.large and c5.2xlarge instances executed on the second processor, as the metal instance uses only Xeon Platinum 8275CL. The processors can be differentiated by comparing the L3 cache sizes. The results for the c7i family are similar to those of the c6i family and lead to the same conclusions. They are available in the Appendix A.5.

All instances from the c5 family achieve the same results if the benchmark input size fits the L1 and L2 caches. The latency equals about 1ns for the L1 cache and below 10ns for the L2 cache. The identical observations are valid for the c6i family, so there is no difference except for the cache sizes between all instances.

However, there are significant differences in behavior between families and the instances within the same family for the L3 cache. For the c5 family, the large and 2xlarge instances behave similarly. They are able to utilize the L3 cache until the input size is below 5MB. For a greater value, the latency starts to increase quickly, which does not happen for the c5.metal instance. For this reason, it is noticeable that the c5.metal instance can use a larger portion of the L3 cache.

On the other hand, all the c6i instances behave differently. The metal instance behaves similarly to the c5.metal instance and fully utilizes the L3 cache. However, the values of the c6i.large and c6i.2xlarge instances start increasing to 100ns much earlier. Nevertheless, this increase happens for a significantly larger input size for the c6i.2xlarge instance. After exceeding the L3 cache size, all instances again achieve values comparable to each other due to accessing data from the main memory.

These differences are caused by CAT, enabled in the c6i family servers. CAT forces each instance to access only part of the L3 cache proportionally to its number of vCPUs. For this reason, the c6i.large instance can use only about 1.7MB of the L3 cache, and the c6i.2xlarge instance utilizes only 6.75 MB, while the whole cache has 54 MB. Consequently, if data exceed the assigned part of the L3 cache, they have to be stored in the main memory, which causes a higher latency visible in Figure 4.12. On the other hand, the metal instance does not share the L3 cache with other instances, and the CAT does not influence the benchmark results.

At the same time, we can observe that the c5 family does not use CAT. The c5.large and c5.2xlarge instances achieve nearly identical results, which should be impossible with the active CAT. The c5.2xlarge instance has four times more vCPUs and should be

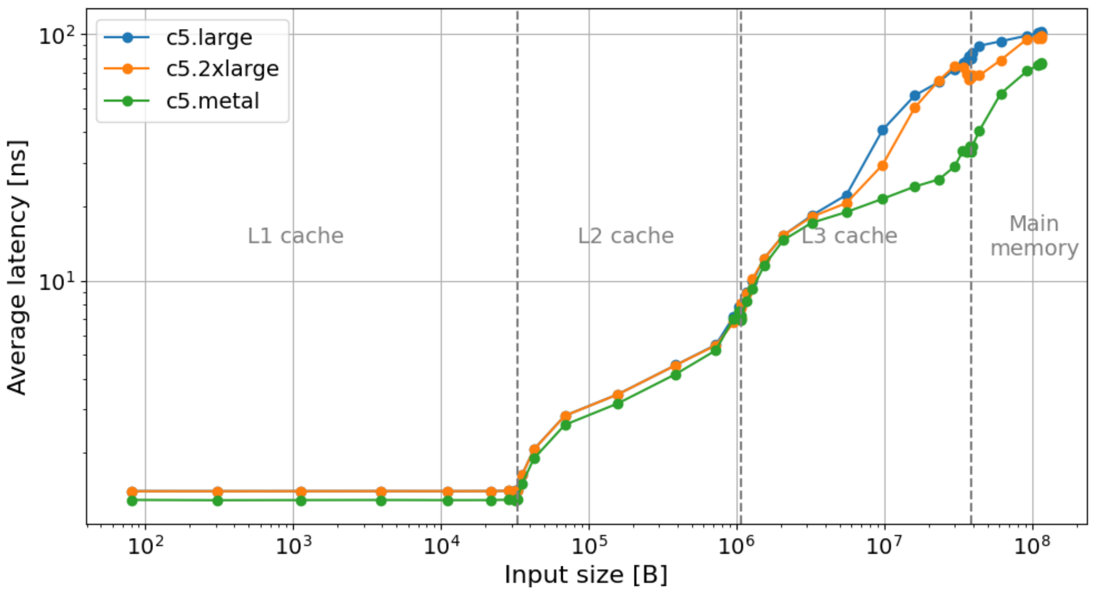


Figure 4.11.: Cache latency benchmark results for instance family c5.

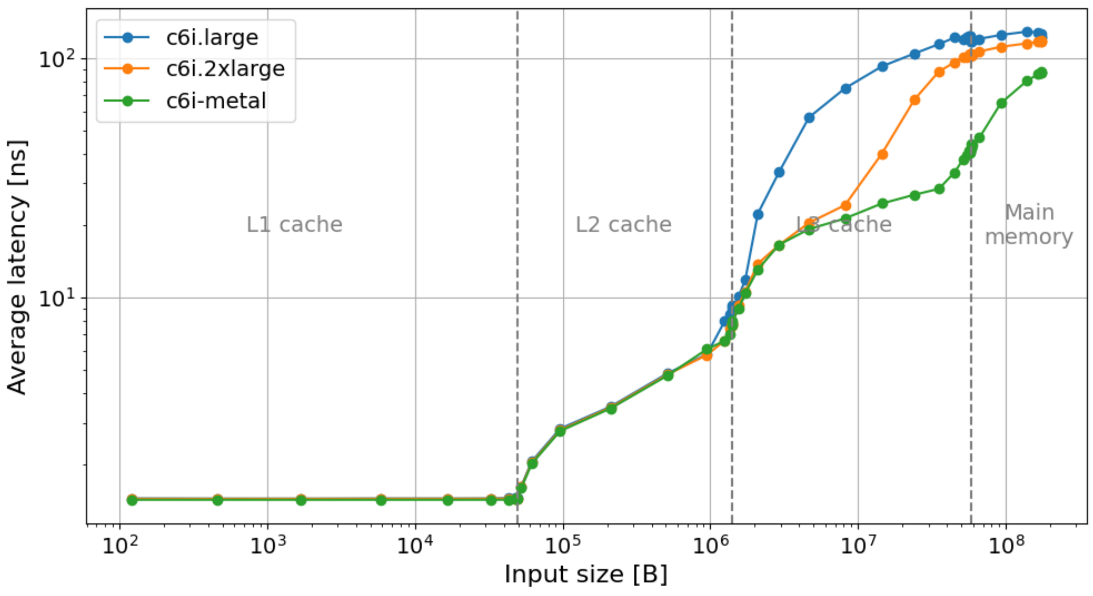


Figure 4.12.: Cache Latency benchmark results for instance family c6i.

able to operate on the four times bigger part of the L3 cache than c5.large. Nevertheless, the c5.large instance utilized around 5 MB of the L3 cache on average in the benchmarks. If the CAT were active, such high L3 cache utilization by the cheapest machine from the c5 family would not be possible because, according to a fair share, it should only have access to around 1.5 MB. However, the c5.large and c5.2xlarge instances could not use the whole L3 cache despite the lack of a limit. It is caused by the fact that the same processors run multiple instances utilizing the common L3 cache. Consequently, utilizing the whole cache by a single virtual machine is difficult. However, it is possible for the metal instance because it had no neighbors who would share the same L3 cache. For this reason, it could operate on the entire L3 cache and achieve the best results.

This comparison presents that introducing CAT in the c6i family leads to better resource separation and a more fair share of the L3 cache. CAT allows the customers who utilize the more expensive instances to operate on the larger part of the L3 cache without the interference of other noisy neighbors.

AMD instances

AMD is another manufacturer whose processors are available in AWS. Its processors offer a similar technology to Intel's CAT called QoS. An essential aspect of comparing AMD and Intel processors is the fact that there is a significant difference in processor design regarding the L3 cache between the manufacturers. In Intel's processors, the L3 cache is shared between dozens of cores. For example, in the processor of the c6i family, 64 threads share a common L3 cache. On the other hand, AMD has multiple L3 cache instances. Single L3 cache instance is shared between 8 or 16 (only for c6a family) vCPUs. This design reduces the impact of noisy neighbors by reducing the number of threads using the same L3 cache. Consequently, the X.2xlarge (c6a.4xlarge for the c6a family) instances have sole access to the entire L3 cache, like metal instances. To analyze the impact of these differences, we performed the analogous tests for AMD instances to check how they impact latency.

In Figure 4.13, we can observe that the instances from the m5a family characterize unpredictable performance. The characteristics of the L1 and L2 caches are identical to those of Intel processors - all instances achieve the same results. The differences between instances are visible only for the L3 cache. The m5a.2xlarge instance behaves precisely as the m5a.24xlarge because they both have access to the entire cache due to AMD processors' design. Moreover, we can observe that m5a.xlarge can also use the whole L3 cache and achieve the same results as m5a.2xlarge. The instance m5a.large achieves only slightly worse results than other instances and uses more cache than

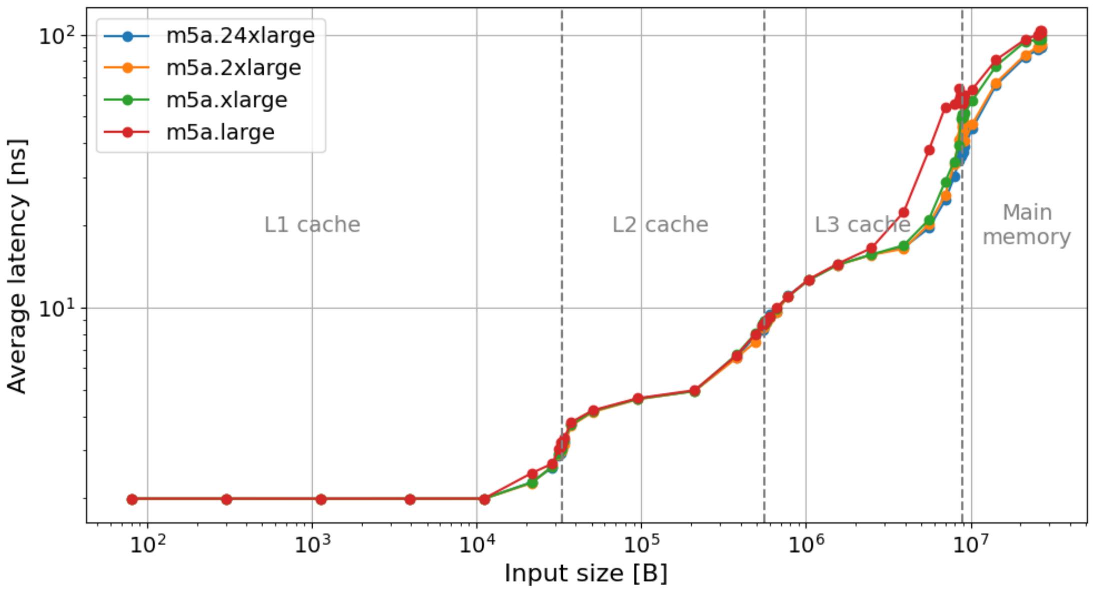


Figure 4.13.: Cache latency benchmark results for instance family m5a.

should if the cache was shared fairly. For this reason, it is visible that the m5a family does not use any technology to fairly share the L3 cache.

While running the benchmark for the c5a, c6a, and c7a families, we observed that the input sizes of the benchmark differ between instances. The medium, large, and xlarge instances operated on smaller datasets than 2xlarge and metal instances. The benchmark calculates input sizes based on the information returned by the `lscpu` command. For this reason, we ran this command manually for these instances and observed that the `lscpu` command returns values proportional to the number of vCPUs for the L3 cache. For example, `lscpu` returns 16MB for the c5a.2xlarge instance and only 4MB for the c5a.large instance. This is the size of the L3 cache fairly assigned to the instance based on the number of vCPUs. This behavior indicates that AMD's QoS is enabled for AMD instances starting from the c5a family and it influences the `lscpu` command contrary to Intel's CAT.

The effects of QoS are also visible on the benchmark latency result. The differences for the L3 cache between c5a.large, c5a.xlarge, and c5a.2xlarge instances seem minimal in Figure 4.14 due to the logarithmic scale. Nevertheless, we can observe that c5a.large uses the smallest part of the L3 cache because its latency increases earlier than the others. If we zoom in and focus only on the L3 cache and the main memory (Figure

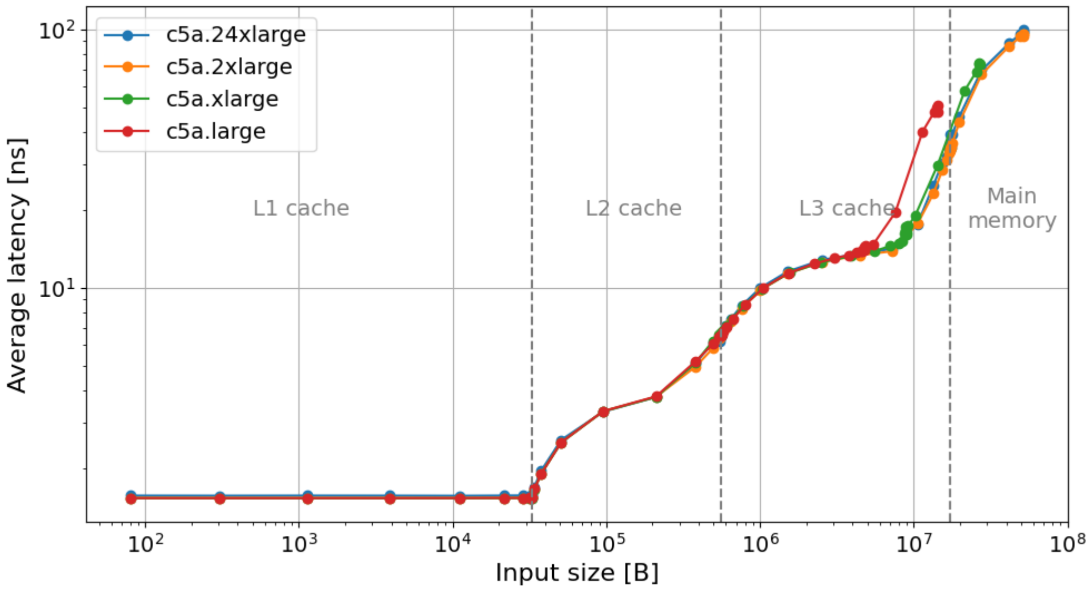


Figure 4.14.: Cache latency benchmark results for instance family c5a.

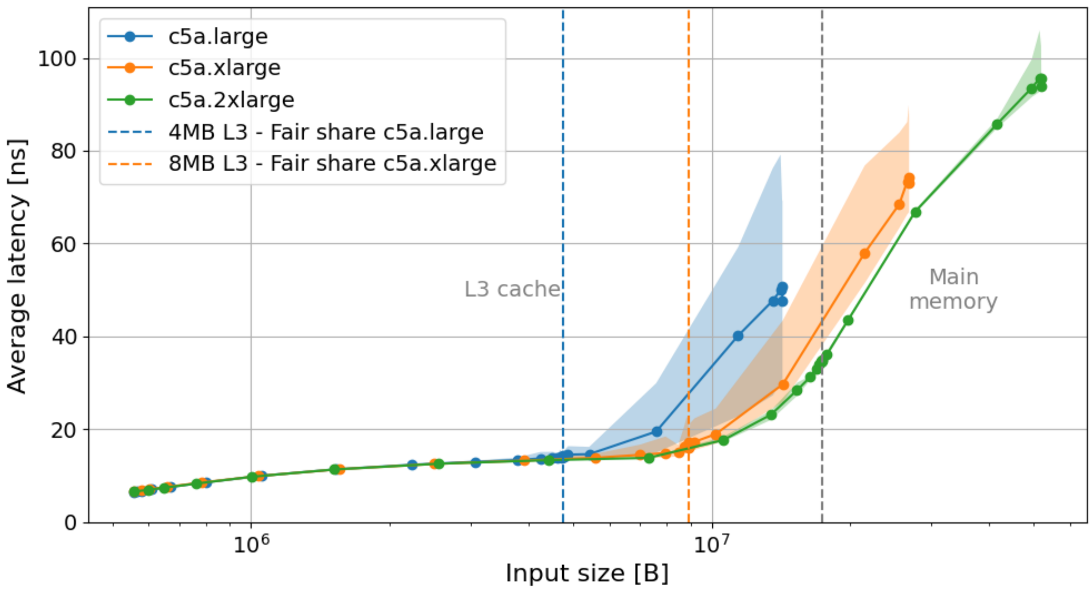


Figure 4.15.: Cache latency benchmark results for L3 cache of instance family c5a.

4.15), we can observe that the latency starts to increase exactly after exceeding the amount of the L3 cache assigned by QoS for the c5a.large and c5a.xlarge instances. Figure 4.15 also marks minimal and maximal values obtained during benchmarks for the given inputs. Until the benchmark inputs are below the fair share of the L3 cache, there are no deviations. However, the deviations become observable for inputs that exceed the amount of the assigned L3 cache to the given instance. We cannot state the exact factors that impact the deviations, but it is noticeable that they begin to occur after exceeding the amount of the assigned L3 cache. For this reason, we can state that AMD's QoS provides resource isolation for the L3 cache for the c5a family. Similar conclusions are valid for the c6a and c7a families, whose results are available in the Appendix A.5.

Graviton instances

The last kind of processor available in AWS is Graviton CPUs. The technologies that provide a resource separation for the L3 cache are officially known for Intel and AMD processors, but there are no such details for Graviton processors. For this reason, it is even more crucial to benchmark Graviton instances. In this section, we concentrate on the c6g and c7g families. The behavior of the instances from the c8g family is similar to the c7g family, and the results are presented in Appendix A.5.

For instances from the c6g family, there is a noticeable lack of resource separation in Figure 4.16. There is no difference between the instances for the L1 and L2 caches, as in the case of Intel and AMD processors. However, the performance is unpredictable when the benchmark starts accessing data from the L3 cache for the non-metal instances. In our tests, the c6g.medium instance achieves even lower latency than the c6g.2xlarge for some input sizes that require access to the L3 cache. While using a technology analogous to CAT, such a situation should be impossible.

For family c7g, there is a significant improvement compared to the previous generation regarding the L3 cache. The c7g.medium is the smallest instance, and according to the fair share, it should only have access to 0.5MB of the L3 cache. For this reason, it has to access data from the main memory much earlier than other instances. The c7g.2xlarge has a bigger part of the L3 cache available. This indicates that the Graviton instances also use a technology similar to CAT to better isolate the L3 cache starting from the c7g family.

These examples show a trend toward providing better resource isolation by AWS. All manufacturers offer technologies to fairly share the L3 cache between multiple users, and the cloud provider uses them in their instances. Additionally, resource isolation

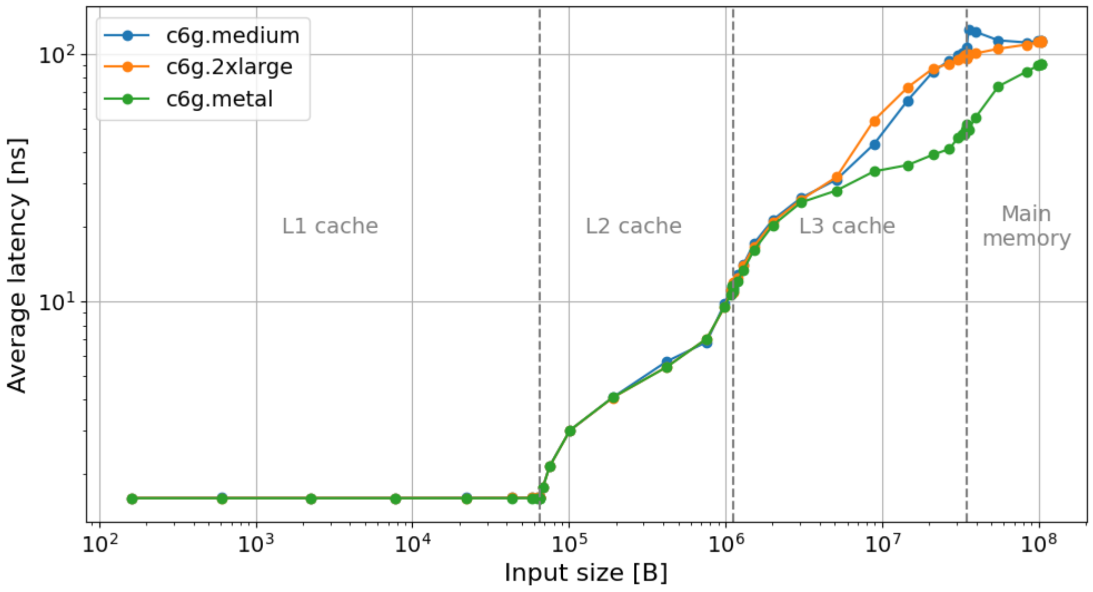


Figure 4.16.: Cache Latency benchmark results for instance family c6g.

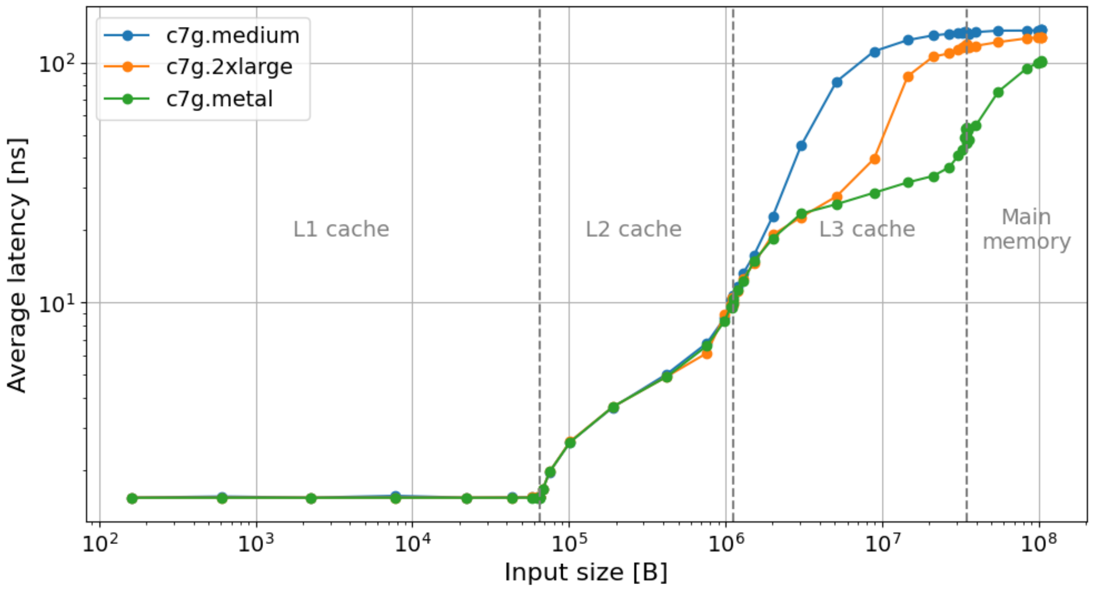


Figure 4.17.: Cache Latency benchmark results for instance family c7g.

differs between instances with processors from different manufacturers. We can observe that Intel and Graviton instances behave similarly in that matter and CAT is easily observable for both of them. On the other hand, AMD changes the value of `lscpu` output, and its effect on latency is more difficult to observe due to the smaller number of threads sharing the L3 cache and large deviations after exceeding the amount of the assigned L3 cache. Furthermore, we notice that while choosing the proper instance for the cache-intensive operations, the user has to be careful. The small instances with CAT have significantly smaller available L3 caches. For this reason, they access the main memory for smaller datasets than the larger instances, such as metal instances.

4.1.4. Memory Bandwidth Allocation technology in the cloud

The L3 cache is not the only resource shared by the multiple instances. The instances also share memory bandwidth. While each instance has an assigned independent area in the main memory, the memory bandwidth is common for all instances running on the same server. Consequently, cloud users can observe the effects of noisy neighbors similar to those observed for L3 caches when some EC2 instances use it disproportionately to their size. As a result, it can negatively impact other instances, as memory bandwidth is not unlimited. For this reason, the manufacturers developed a technology similar to CAT that operates on memory bandwidth called Memory Bandwidth Allocation (MBA)[27]. Instead of dividing the L3 cache, it limits the memory bandwidth available to the virtual machine, preventing the utilization of the whole memory bandwidth by a single instance.

However, similar to the CAT case, some details about this technology are nontransparent. MBA is Intel's technology, but AMD offers its equivalent as a part of QoS [15]. However, again, there is no detailed information about resource isolation of memory bandwidth for Graviton instances. For this reason, benchmarking different instances is critical to discovering whether cloud providers use resource isolation technologies on memory bandwidth, whether they differ between manufacturers, and how they impact performance. We tried answer these questions by executing memory bandwidth benchmarks using TUManyBenchmarks.

Intel instances

The most known technology to limit memory bandwidth is Intel's MBA. For this reason, we have analyzed how the memory bandwidth behaves for Intel instances. Figure 4.18 shows that the memory bandwidth of the different instances from the c5 family grows proportionally with the number of used physical cores. When the number of

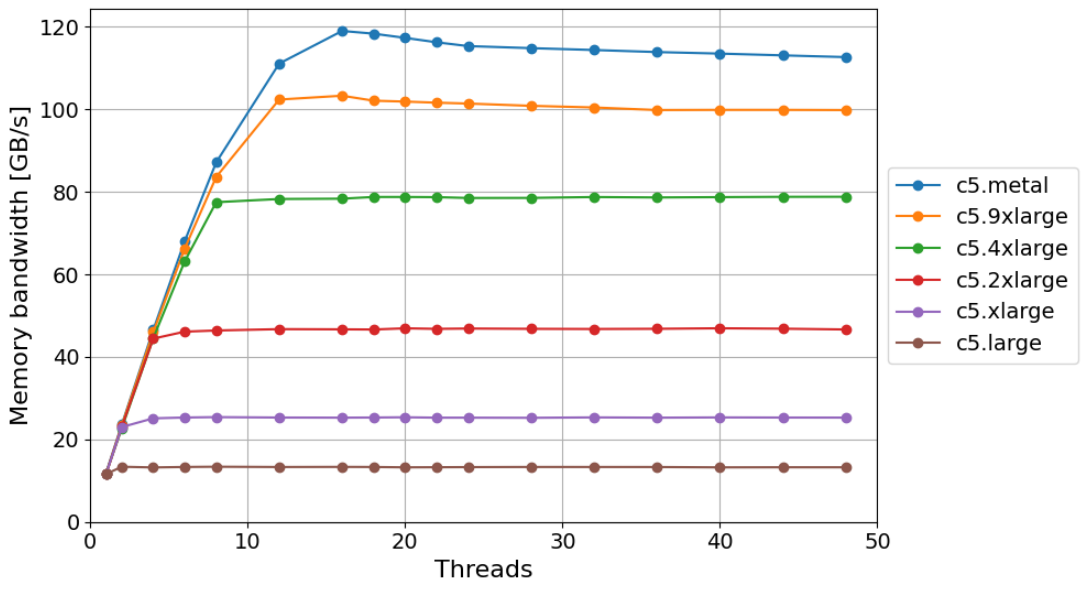


Figure 4.18.: Memory Bandwidth benchmark results for instance family c5.

threads exceeds the number of cores, the bandwidth becomes constant. However, there is a difference for c5.9xlarge and c5.metal instances for tests with more than 8 threads because the increase in the memory bandwidth slows down. The c5.9xlarge instance requires 12 threads to achieve maximal bandwidth despite having more processing capabilities as it has 18 physical cores. On the other hand, the c5.metal instance requires 16 threads to achieve the maximal bandwidth. This difference can be caused by the limit set by MBA or the fact that the c5.9xlarge instance runs together with other users' instances on the same hardware. The metal instance has the whole hardware for private use, so MBA and the other AWS clients do not affect it. Moreover, the perfect constant results for instances between c5.large and c5.4xlarge can be caused by MBA, but the limited processing capability can also cause them. Nevertheless, we can observe that the memory bandwidth is not shared fairly. For example, the c5.large instance utilizes 5% (13GB/s) of the memory bandwidth available for the c5.metal (2 NUMA nodes * 120 GB/s), which is too much for such a small instance if there would be a fair share. All of these aspects result in the fact that there is no clear answer to the question of whether the c5 family uses MBA. The results for newer Intel families (c6i and c7i) also do not provide a clear answer about MBA because the memory bandwidth scales with a number of vCPUs, but the instances also use a disproportionally large part of bandwidth compared to the metal instances. The plots for these families are available in Appendix A.6.

AMD instances

AMD's QoS provides features to not only manage the L3 cache but also memory bandwidth[15]. As we observed in Section 4.1.3, AMD technologies can differ in behavior from Intel despite also using x86 architecture. For this reason, we analyzed whether memory bandwidth's resource isolation differs between AMD and Intel instances.

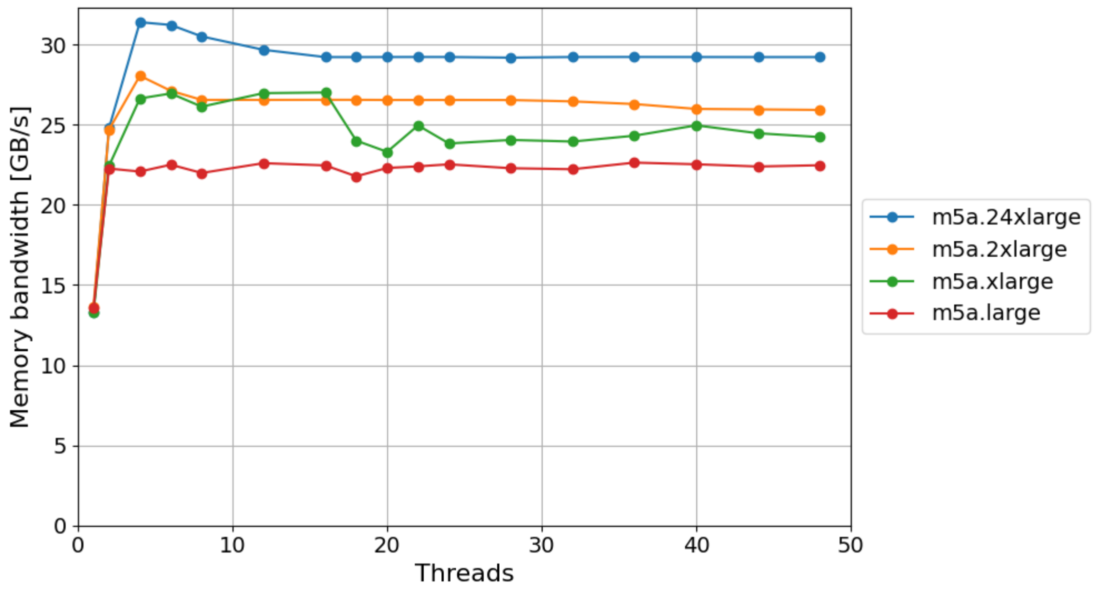


Figure 4.19.: Memory Bandwidth benchmark results for instance family m5a.

The family m5a is a perfect example of a lack of resource isolation for memory bandwidth. The results presented in Figure 4.19 are characterized by large fluctuations, which are the most visible for the m5a.xlarge instance. The m5a.xlarge instance's results significantly vary between executions for different numbers of threads, though the instance has only four vCPUs. Due to the fluctuations, this instance sometimes achieves a higher bandwidth than two times more expensive m5a.2xlarge. Additionally, the cheapest m5a.large instance can utilize about 80% of the memory bandwidth available for m5a.2xlarge instance, which is four times more expensive. These observations prove that the m5a family does not support resource isolation for memory bandwidth, as the bandwidth is not shared fairly between different instance types, which can negatively impact the user's processing.

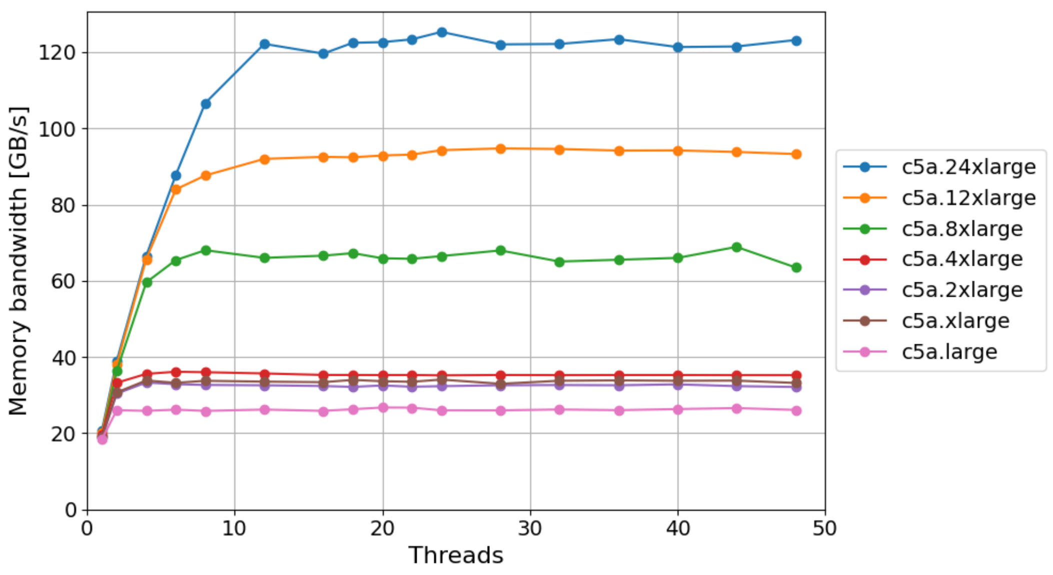


Figure 4.20.: Memory Bandwidth benchmark results for instance family c5a.

Figure 4.20 presents the results of the benchmark for the c5a family. The results significantly differ from the results for the m5a family, but also from the Intel instances' results. There are noticeable limits on the memory bandwidth, equal from 30 to 40 GB/s for all instances smaller than c5a.8xlarge. Moreover, we can observe that the limits are much higher for the larger instances like c5a.8xlarge and c5a.12xlarge. The limits are not caused by the lack of processing capabilities, as only two threads are required to reach the limit for the c5a.4xlarge instance, which has 16 vCPUs. The limits must be caused by the QoS to provide resource isolation. They prevent the single instance from utilizing the whole memory bandwidth and solve the noisy neighbor problem. However, they are not proportional to the number of vCPUs, so it is unfair to some customers. For example, the c5a.4xlarge instance can use nearly the same bandwidth as c5a.xlarge despite being four times more expensive. If the user's processing is memory-bounded, they must be concerned about it when selecting the proper EC2.

The effects of AMD's QoS are also visible in newer AMD families. The detailed results for c6a and c7a families are available in Appendix A.6. For family c6a, the instances between c6a.large and c6a.8xlarge can use only from 30 to 40 GB/s of the memory bandwidth, whereas, for c6a.12xlarge, the values skyrocket to values achieved by the metal instance. On the other hand, the c7a family provides much better granularity than previous generations because only c7a.large, c7a.xlarge, and c7a.2xlarge achieve the same results.

Graviton instances

The last type of instances available in the AWS are the ones with Graviton processors. Similar to the L3 cache, there is no detailed information on whether they use a technology equivalent to Intel's MBA, which allows for a fair share of memory bandwidth. For this reason, the only way to discover the memory bandwidth characteristic is to execute benchmarks.

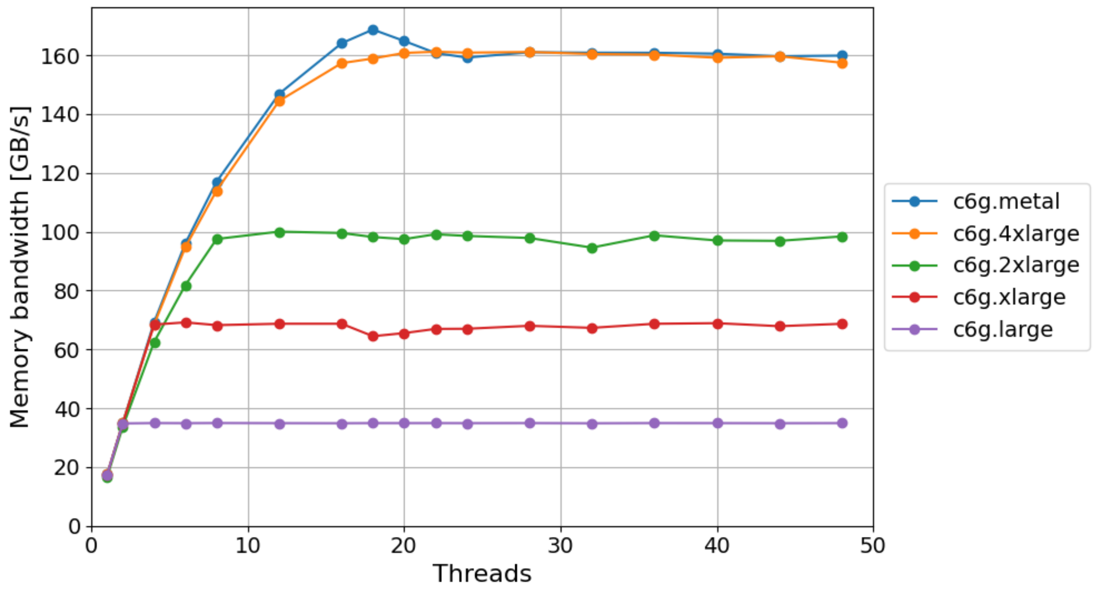


Figure 4.21.: Memory Bandwidth benchmark results for instance family c6g.

The analysis of the c6g family presented in Figure 4.16 shows that the memory bandwidth increases with the number of vCPUs for the instances from this family. Additionally, we can observe small fluctuations in the values for the c6g.xlarge and c6g.2xlarege instances. This suggests that there is no artificial limit for their memory bandwidth. Moreover, c6g.4xlarge can achieve the identical bandwidth as c6g.metal. The c6g.4xlarge instance, contrary to c6g.metal, runs with other virtual machines on the same hardware. For this reason, such high memory bandwidth can negatively impact other instances. These observations indicate that the c6g family does not support resource isolation for memory bandwidth, as it does not support the resource isolation for the L3 cache.

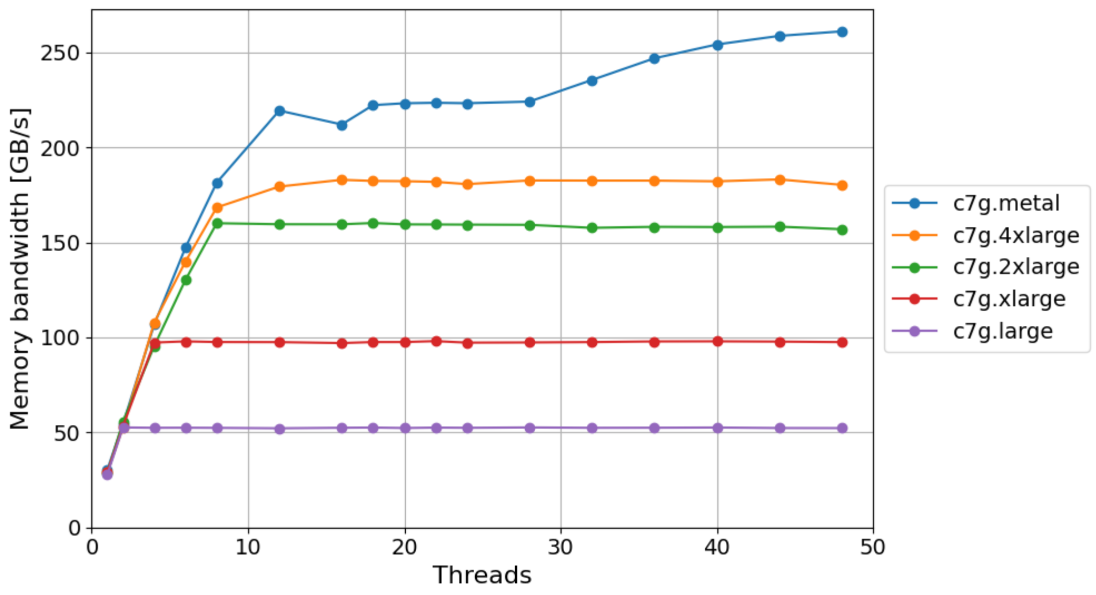


Figure 4.22.: Memory Bandwidth benchmark results for instance family c7g.

The results for the c7g family illustrated in Figure 3 show a difference compared to the previous generation. The lines representing the values for small instances like c7g.xlarge and c7g.2xlarge are straight after the stop of rapid increase, whereas the analog instances from the c6g family have visible fluctuations. Additionally, the memory bandwidth of the c7g.4xlarge instance increases similarly to c7g.metal until it utilizes only eight cores. Later, the memory bandwidth increase is slight, and then the memory bandwidth becomes constant. This unexpected behavior cannot be explained by the lack of processing capabilities because the c7g.4xlarge instance has 16 physical cores. The metal instance achieves much higher bandwidth using the same number of threads. The stop indicates a limit in memory bandwidth available for the c7g.4xlarge instance, which can be caused by MBA or lack of the available memory bandwidth due to the neighbors. The constant maximal values and the sudden stop of increase may suggest that the c7g family supports some equivalent to MBA, which limits the amount of available memory bandwidth for different types of instances. However, if there is MBA, it does not offer a fair share as the c7g.large instance can use 50GB/s of the bandwidth which is about 20% of the memory bandwidth available for the metal instance. We can draw similar conclusions from the analysis of the c8g family, whose results are available in Appendix A.6.

The analysis of the memory bandwidth of various instance families confirms a trend toward providing better resource isolation by AWS, which we observed in the L3 cache. We have observed that limiting memory bandwidth differs between instances with processors from different manufacturers, just as it differs for the L3 cache. We can observe that Intel and Graviton scale the maximal memory bandwidth with the number of vCPUs, but often, it is hard to determine whether the limit is caused by the lack of processing capabilities or MBA. On the other hand, MBA is clearly visible for AMD because it often sets a constant maximal value common for multiple small instances with different numbers of vCPUs. These examples show that memory bandwidth behavior can differ between the processors and show the importance of benchmarking multiple instances.

4.2. Example models

Another goal of TUManyBenchmarks is to support the user while choosing the most proper EC2 instance for their use cases based on hardware metrics. The service offers a query console that enables users to create various models using the results collected by the benchmark service. This section will present some examples of queries that can create models for different purposes and help compare instances. The queries are based on the tables `instances`, `benchmarks`, and `statistics` presented and detailed described in Section 3.4.3

4.2.1. Model finding the cheapest instance meeting criteria

The first example is a simple model that selects instances meeting the minimum criteria and sorts them. Listing 4.1 presents an example of such a query. In this model, the user must specify the instance's requirements. The requirements can involve the instance properties, such as vCPU and memory size, or benchmark result statistics. The user can access one of the four precomputed statistics: minimum, maximum, average, and median. These four statistics are calculated for each benchmark result series, which allows the user to create queries with various conditions. The final step is sorting the instances meeting criteria by the price (on-demand or spot) ascending. The presented instances meet requirements, so using one of the cheapest is the most cost-efficient choice.

```
SELECT i.name, i.vcpu, i.spot_price,
       s1.min as min_bandwidth, s2.avg as network_latency
FROM instances i, benchmarks b1, benchmarks b2,
     statistics s1, statistics s2
WHERE i.id = s1.instance_id AND i.id = s2.instance_id
AND b1.id = s1.benchmark_id AND b2.id = s2.benchmark_id
AND i.vcpu >= 4
AND b1.name = 'Network␣Bandwidth'
AND s1.series = 'sum_bandwidth'
AND s1.min > 20
AND b2.name = 'Network␣Latency'
AND s2.series = 'percentile_99_900'
AND s2.max < 50
ORDER BY i.spot_price ASC
```

Listing 4.1: Example of query finding the cheapest instance meeting criteria.

4.2.2. Model finding the best performance-price ratio

The next model focuses on achieving the best ratio between resource amount or quality and price. For example, the user can use the type of query to identify the instance with the highest value of the memory GB per dollar. However, it can also be applied to benchmark results, as shown in the example in Listing 4.2. As we observed in Section 4.1.2, the vCPUs from different generations and manufacturers can achieve different performances. This example query enables the user to choose the instance with the most cost-efficient single-core processing based on the SPEC CPU2017 Integer Rate benchmark. For example, the more expensive AMD instance may be more beneficial than the Graviton instance because it completes processing faster. As a result, the AMD instance may generate a lower cost despite a higher price per hour. This model can be smoothly integrated with the model presented in Section 4.2.1 to filter out some instances based on other properties.

4.2.3. Model finding the most cost-efficient instance in OLAP processing

The query console allows the users to write arbitrary queries. For this reason, it is also possible to create complex models consisting of a sequence of SQL subqueries. For example, the user can create a query to find the most cost-efficient instance in the Online Analytical Processing (OLAP) processing as proposed by Leis and Kuschewski[31].

```
SELECT i.name, i.vcpu, i.spot_price, s.avg,
       (s.avg / i.spot_price) AS value_to_price_ratio
FROM instances i, benchmarks b, statistics s
WHERE i.id = s.instance_id AND b.id = s.benchmark_id
AND b.name = 'SPEC␣CPU2017␣Integer␣Rate␣-␣Single␣Core'
AND s.series = 'SPECrate2017_int_base'
ORDER BY value_to_price_ratio DESC
```

Listing 4.2: Example of query finding the best performance-price ratio.

Additionally, the user can improve the model by using benchmark results instead of some assumptions, which makes the model more reliable. For simplicity, we focus on the query for the "Basic Model (M1)", which depends only on the number of cores and network bandwidth. However, even this simple example is enough to show the possibilities of the service, and extending it to other models requires just adding more parameters and connecting them with proper benchmark results analogically.

The M1 model takes two parameters as input: `CPU hours` and `scanned data`. Then, it divides `CPU hours` by the number of vCPUs to obtain processing time and divides `scanned data` by 80% of network bandwidth to obtain download time. These two results are summed and multiplied by the price per second to obtain the total processing cost. However, the model assumes that all vCPUs from different machines have the same performance, which is untrue (Section 4.1.2). Additionally, it assumes the constant network bandwidth over time, but for many instances, the maximal bandwidth is achievable only temporarily (Section 4.1.1).

For this reason, we can include hardware metrics from the benchmarks to make the model more accurate. Suppose that 1 CPU hour equals an hour of processing on a single SPEC CPU Reference Machine core. In that case, we can use SPEC CPU Rate results from the `statistics` table to compute processing time concerning differences between vCPUs. Instead of dividing CPU hours only by the number of vCPUs, the new model divides it by SPEC CPU Rate for the processing using all cores, indicating how much higher throughput the tested machine's core has than the reference machine [44].

Similarly, we can use network bandwidth results to calculate how many seconds the given instance needs to download `scanned data`. However, we need more detailed information than stored in the `statistics` table for this case. For this reason, first, we create a view called `cumulative_bandwidth_view`, which accesses detailed benchmark results stored in the `benchmarks` column in the `instances` table. Based on values from JSON, the average bandwidth for each processing second is calculated for each instance.

```
WITH download_time AS (
    SELECT instance_id, time
    FROM (
        SELECT
            instance_id,
            time,
            row_number()
                OVER (PARTITION BY instance_id ORDER BY time) AS row
        FROM cumulative_bandwidth_view
        WHERE cumulative_bandwidth >= getvariable('scanned_data_gb')
        ORDER BY time
    )
    WHERE row = 1
),
compute_time AS (
    SELECT
        i.id AS instance_id,
        getvariable('cpu_hours') / s.avg * 3600 AS time
    FROM instances i, benchmarks b, statistics s
    WHERE i.id = s.instance_id AND b.id = s.benchmark_id
    AND b.name = 'SPEC␣CPU2017␣Integer␣Rate␣-␣All␣Cores'
    AND s.series = 'SPECrate2017_int_base'
)
SELECT
    i.name,
    d.time + c.time AS processing_time,
    (d.time + c.time) * (i.spot_price/3600) AS cost
FROM instances i, download_time d, compute_time c
WHERE i.id = d.instance_id AND i.id=c.instance_id
ORDER BY cost ASC
```

Listing 4.3: Example of query finding the most cost-efficient instance in OLAP processing.

As the last step, the query uses window operation to compute the sums of averages, which indicate how many gigabytes were downloaded until the given second. The complete code is presented in Listing A.7 For simplicity, we assume that the downloads take up to 600s.

Finally, we can use both benchmarks to calculate processing and download times using the `WITH` clause. The first subquery obtains the minimal time required to download scanned data using the prepared view and window operator. The second subquery calculates the time required to process data using the given instance based on its SPECrate 2017 Integer Base value for processing using all vCPUs. Then, we calculate the cost using computed times and price per hour and sort the results by the price to get the cheapest instance on the top. The complete query is presented in Listing 4.3. This query explicitly exemplifies how TUManyBenchmarks can create complex models based on hardware metrics by accessing detailed results and precomputed statistics using only SQL. The main task while creating such models is to identify and replace the assumptions and basic instances' properties with the proper benchmark results.

4.3. Benchmarking simplification

In Chapter 1, we observed that correct benchmarking is difficult. One of the reasons why we have decided to develop TUManyBenchmarks is to simplify this process. The service helps not only in benchmark executions but also in resource management and simplifying benchmarks themselves.

4.3.1. Simplification of benchmark setup

The main advantage of TUManyBenchmarks is the simplification and automatization of resource management. The service removes the user's responsibility for creating and managing the resources, which brings multiple advantages. The first is that the user does not have to spend time creating resources using AWS Console or AWS CLI or writing Terraform files. Automating this process also guarantees its correctness, as it is highly probable that the user will accidentally omit some configuration options while creating it manually each time. Additionally, the service can terminate all resources right after the benchmark is completed and the output is extracted. While running benchmarks manually, it is the user's responsibility to delete resources. This causes the resources to stay idle even after finishing the benchmark and generates unnecessary costs. It is especially crucial for long-running benchmarks such as SPEC CPU, whose execution takes a few hours. In such a case, predicting precisely when the benchmark is completed is hard. Consequently, the resources can remain unused and generate costs as nobody checks their state constantly to terminate them at the correct moment.

Additionally, TUManyBenchmarks is especially beneficial when the benchmark requires multiple nodes to communicate with each other. The typical case of such a situation is measuring network bandwidth and latency. The service takes responsibility for communication setup and synchronization, allowing users to focus only on their benchmark. First, the service automatically updates the nodes' `/etc/hosts` files. This simplifies benchmarking because the users do not have to discover the correct IP addresses during execution by themselves. Instead, they can specify the domain name using the `name-$nodeId` pattern to connect to the correct node. Moreover, TUManyBenchmarks provides a synchronization mechanism across nodes between the setup and execution phases and between the execution and output extraction phases. The benchmark execution only starts if all nodes are ready, and the output-extracting process starts when all benchmark processes are completed. Without using our service, the user has to use some additional program that plays the role of distributed barrier and synchronizes the nodes. However, it causes unnecessary work for the user not connected directly to the benchmark execution.

4.3.2. Simplification of multiple measurements

The users usually run benchmarks to compare the performance of different machines, which results in running the same benchmark multiple times. Typically, the user writes a script that automates this process. However, it requires additional effort and time from the user. TUManyBenchmarks enables users to run the benchmark on multiple machines without additional effort. The user only specifies all instances (or tag combinations) for which the benchmark should be run, and the service takes care of the whole process.

Additionally, TUManyBenchmarks manages available AWS quotas. Sometimes, running multiple benchmarks at once is impossible due to a limited number of quotas. This problem often occurs while simultaneously running benchmarks for a few metal instances with dozens of vCPUs. While running the benchmarks manually, the user has to wait and start executing the following benchmarks on their own when the previous ones are completed. TUManyBenchmark simplifies this process and automatically starts the next benchmark when the previous benchmarks release enough resources.

TUManyBenchmark also helps with regularly running benchmarks. Repeating benchmarks multiple times is crucial to obtain valuable measurements, as results can differ between executions. It is especially essential for testing virtual machines offered by cloud providers, as noisy neighbors can significantly impact the results. While running the benchmark themselves, the user must manually start it multiple times, which limits the parts of the day for which the benchmark is executed to working hours. The user can also create a cron job to automate it and take measurements in other parts of the day, such as at night. However, it requires a machine with an active cron job to always be active and waste resources if the benchmark is executed rarely. TUManyBenchmarks can also solve this problem. While defining the new benchmark using our service, the user must specify a cron expression in the `configuration.yml` file. Then, the service schedules the benchmark executions based on it. This functionality simplifies gathering multiple measurements from different parts of the day for an extended period of time. It also minimizes wasting resources as the service simultaneously monitors multiple benchmarks.

4.3.3. Other benefits

Moreover, using our service increases the assurance that the benchmark results are correct. The common problem with writing benchmarks on their own is the correctness. TUManyBenchmarks helps with that problem because the critical part of adding a new benchmark is the review process from the maintainer. The maintainer is a person with deep knowledge and comprehensive experience about benchmarks. For this reason, such a person can spot eventual problems with the benchmark and propose fixes. Additionally, the accepted benchmarks are an excellent basis for creating new benchmarks, as some general rules are common for all benchmarks, such as printing to avoid dead code elimination [44].

TUManyBenchmarks can also eliminate the need for benchmarking for many users, saving them time and money. The service collects many different benchmark results in one place. Additionally, these results are trustworthy as the whole benchmarking process is transparent. For this reason, the user often can use the data provided by TUManyBenchamrks instead of running their own benchmarks, which enables them to quickly complete the tasks for which they need benchmark data.

4.4. Characteristic and requirements revisited

4.4.1. Characteristic

In Chapter 1, we observed a need for a service that enables users to compare cloud providers' virtual machines using hardware metrics. We have stated that such a service should be characterized by extensibility and transparency.

We have presented the extensibility of our service by running various benchmarks focused on different hardware metrics, such as CPU, network bandwidth, and memory bandwidth. Our service accepts all types of benchmarks, which can be run using a single or multiple EC2 instances. Compared to the standard benchmark procedure, the user has to provide one more file (`configuration.yml`), which is enough to describe setup, execution, and output extracting processes. This makes integrating existing benchmarks with our service a straightforward process. Additionally, we integrated our service with Ansible, which is highly configurable and simplifies the process of preparing the environment for new benchmarks.

As stated in Chapter 1, we aimed for transparency both from the perspective of the benchmarking process and cloud vendors. Our service guarantees the transparency of benchmarking by storing all benchmark files in the public GitHub repository. This way of storing files allows all the users to see the details of executed benchmarks and get a full view of what and how the given benchmark measures. This also allows users to download and execute the benchmarks on their computers or virtual machines to better understand them. With all publicly available benchmark results, users can compare their results with those presented by the service, reinforcing their trust in the service's data.

In order to provide transparency from the cloud vendors' perspective, the service allows access to detailed information about EC2 instances' hardware metrics. The provided information is much more specific and low-level than metrics such as the number of threads or maximal network bandwidth published by cloud providers. For this reason, it provides more transparent information about available instances than the specification provided by AWS. Consequently, TUManyBenchmark enables much more detailed and precise comparisons between different EC2 instances.

4.4.2. Requirements

In Chapter 3, based on the characteristics, we derived the requirements that we meet during the implementation of our service. The main requirement was allowing users to provide their own benchmarks and automate their execution. We have achieved this by creating a GitHub repository to which anyone can contribute. After the review and merge, the service can access and execute the benchmarks without needing the user's or the maintainer's interactions. Using the GitHub repository also simplifies the review process. GitHub is one of the most fundamental tools that software developers and researchers use, so they are already familiar with it. GitHub also enables the maintainer to use the PR review functionalities to point out the problems with the benchmark or its benchmark configuration. Furthermore, the part of the review is done automatically by GitHub Actions, which checks the correctness of the `configuration.yml` file. The GitHub Actions are also responsible for updating the database after accepting PR by the maintainer.

The second requirement was to enable users to access the benchmark results and help them choose the most suitable instance for their use case. The developed web application plays a central role in achieving this goal. It allows users to use filters to find instances based on their basic properties and benchmark results. Additionally, integrating DuckDB Wasm within the web application simplifies the querying data process and gives many ready-to-use functionalities. This enables users to create complex and well-suited models for their use cases based on the benchmark results. The web application also creates plots to visualize data and help compare different instances. Support for exporting query results to the CSV file also helps users process data locally and create plots using their favorite external tools.

4.4.3. Comparison to other services

TUManyBenchmarks offers multiple functionalities to support users with a cost-efficient selection of virtual machines. However, it is one of many solutions created with this aim. We have classified the other existing solutions into three groups: instance comparison services, benchmark services, and automatic selection tools. In this section, we will focus on their general specification and functionalities. The details of examples of each group are presented in Section 5.

TUManyBenchmark and all three types of services allow the users to access information about instances. However, only TUManyBenchmark and automatic selection tools provide information about multiple hardware metrics on which users can compare instances. The benchmark services also provide hardware metrics and enable comparisons, but usually, they focus only on one chosen metric, which does not guarantee the full view of the instances. The same case applies to instance comparisons. Simple comparison services operate only on data provided by cloud providers, which do not share detailed information. TUManyBenchmarks has one unique functionality regarding instance comparison. Besides simple filtering, it allows for creating SQL queries, which provide much more elasticity and functionalities, which is especially practical for advanced users such as researchers. Additionally, TUManyBenchmarks was created to help users with any use case, similar to comparison services. This is the problem of automatic selection tools, which often work for only one specific product, e.g., SQL Server databases. Benchmark services provide only one metric, for example, CPU performance, which is useless if the network bandwidth is the metric in which the user is interested. Another unique TUManyBenchmarks functionality is the possibility for the users to add new benchmarks. This feature allows the users to impact the service and make it more valuable for themselves and others. These examples show that TUManyBenchmarks combines functionalities of instance comparison services, benchmark services, and automatic selection tools. Moreover, it extends them by adding unique functionalities, such as a query console and the possibility to add their own benchmarks, which makes it even more practical.

	TUMany-Benchmarks	Instance comparison service	Benchmark service	Automatic selection tools
Information about instance properties	+	+	+	+
Information about hardware metrics	+	-	+/-	+
Instance comparison	+	+	+	+
Instance comparison based on hardware metric	+	-	+/-	+
Instance comparison using SQL queries	+	-	-	-
Work with any use case	+	+	-	-
Allows for adding new hardware metrics	+	-	-	-

Table 4.1.: Comparison of functionalities of TUManyBenchmarks with other types of services.

5. Related work

Due to its complexity, choosing the most cost-efficient instance configuration for different purposes and its automation led to the development of many solutions. In this chapter, we will compare existing services to ours.

5.1. Services to compare virtual machine instances

One of the most commonly used web services for comparing different virtual machine instances are services like `https://instances.vantage.sh/`[48, 49] and `https://gcloud-compute.com/`[39]. These services provide a simple web application for searching and comparing instances. They base their functionalities on publicly available information that cloud providers share via APIs. However, they merge data from multiple sources and present them in a well-structured and user-friendly way. Using these applications, users can filter and sort instances by their fundamental properties like name, vCPUs number, memory size, and network throughput. Moreover, simultaneously, they can see current instant prices in different purchase models (on-demand, spot, reserved) and compare prices in different regions. This information allows users to find the most appropriate instance by comparing alternatives. However, contrary to our service, these services do not provide information about hardware metrics. For this reason, our service can also be helpful for more advanced users interested in the hidden details of the machines.

Leis and Kuschewski[31] have analyzed different AWS EC2 instances in the context of query processing. To solve the selection problem in this scenario, they have proposed a model to determine the optimal instance configuration for OLAP workloads and published it as a service (`https://maxi-k.shinyapps.io/costoptimal/`). The model selects the optimal instance based on:

- workload - input size, materialization fraction, caching and materialization data distribution,
- hardware - number of vCPUs, memory size, network throughput, disk size,

- cost - on-demand or spot instance prices,
- scaling - number of nodes and how efficiently distributed workload scale across nodes.

Their model is beneficial but may be hard to use for general use cases, as workload parameters are strictly connected to query processing. Our service does not have such limitations because it allows users to write arbitrary models using SQL queries on hardware metrics. For this reason, users can use it to create models with any resources as bottlenecks, depending on their needs. An additional observation is that such models as the one created by Leis and Kuschewski could provide more accurate predictions if they base their computation on benchmark results provided by our service. These data can be accessed directly using API exposed by the back-end webpage application.

5.2. Automatic cost-effective selection of virtual machines

The cloud provider also offers additional services, whose task is to simplify the transition from an on-premise server to their cloud infrastructure. An example of such a solution is Doppler[18], created by Microsoft. It is a part of Azure Data Migration Assistant and recommends the proper server during migration from the on-premise SQL Server database to Microsoft Azure SQL. It advises the instance for a new database based on low-level metrics such as CPU and memory usage, input/output operations per second (IOPS), and latency. Based on these data, Doppler calculates the probability of throttling if the given instance is chosen, which can be presented as a plot to a user. The disadvantage of this solution is that it is limited to only one specific solution of Microsoft Azure and, consequently, works for a small subset of virtual machines. Although Doppler is an automatized solution that bases recommendations on hardware metrics, the customer cannot use it when choosing the virtual machine instance for general use. Additionally, our solution can be extended to provide information about multiple cloud providers, contrary to Doppler, which focuses on Microsoft Azure products.

Another example of automatic cost-effective virtual machine selection is OPTIMUSCLOUD[32]. It is a configuration system for performance-per-cost optimization that can be used with Cassandra and Redis, popular NoSQL databases. The system adjusts cluster and database configurations to maximize performance per dollar with a given budget and minimum throughput. Firstly, the system starts with a configuration based on historical traces. Afterward, during the work of a database, OPTIMUSCLOUD can adjust configurations online to adapt dynamically to the changing workload. As a part of optimization, the system finds optimal cluster configurations, which can consist of different EC2 instance types. The system uses EC2 instance parameters such as memory size to predict performance and select the optimal configuration. Although our system cannot automatically choose the most cost-efficient instance, it does not require priory historical traces. For this reason, our service can provide cost-efficient configuration from the beginning after writing the proper queries. Also, similar to Doppler, OPTIMUSCLOUD is limited to databases, contrary to our solution, which can be used for any case.

5.3. Benchmark services for virtual machine instances

Websites presenting benchmark results are other services that help with the cost-effective instance selection problem. An example of such a web page is ClickHouse Hardware Benchmark[19]. ClickHouse Hardware Benchmark runs ClickBench, a popular benchmark for comparing OLAP databases, using ClickHouse but on different hardware. In this way, the benchmark can compare different instances. In a similar way works, Spare Cores [21]. This service benchmarks instances from different vendors using the div16 method from the stress-ng benchmark[20]. This type of website allows for comparing results from multiple different vendors. However, they usually run only one benchmark. In contrast, our service is more extensible and allows for running an unlimited number of benchmarks, which users can also provide. This feature allows the user to compare instances from different views.

SPEC CPU authors, the benchmark used during evaluation, also maintain a webpage on which they publish their benchmark results[45]. The user can access the results of many systems with different CPUs from multiple manufacturers, such as Asus, Cisco, and Dell. It is valuable because the SPEC CPU is not publicly available, so the users do not have to buy the license to see results. However, contrary to our service, the website is focused on on-premise servers rather than cloud vendors' machines. Additionally, SPEC CPU's webpage provides limited functionalities to search results and does not allow comparisons, which makes it less user-friendly than our service. Moreover, as it is the SPEC CPU benchmark's webpage, it only presents this benchmark's results and ignores other machine characteristics, such as network bandwidth.

6. Conclusion and Future Work

6.1. Conclusion

This thesis presented TUManyBenchmarks - the extensible benchmarking service for comparing EC2 instances based on hardware metrics. Its main goal is to provide users with detailed instance characteristics that cloud providers typically do not make available, such as the vCPU's performance. We aimed to create a service whose central features are transparency, guaranteeing the collected results' trustworthiness, and extensibility, allowing users to add new benchmarks.

We started by collecting requirements and, based on them, designing the architecture consisting of GitHub Actions, the MongoDB database, the benchmarking service, and the webpage. The GitHub Actions are responsible for simplifying and automatizing the process of adding the new benchmarks. The benchmark service executes benchmarks contributed by the users to the GitHub repository and manages AWS infrastructure during that process. The database stores data about benchmarks, instances, and results, which are later accessed by the webpage application integrated with DuckDB Wasm.

Finally, we evaluated the service and its practical applications. We set up multiple benchmarks and demonstrated example observations. For instance, we showed how the performance and cost-efficiency of the processors from different manufacturers improved over the years. We observed that at the beginning, the Graviton processors characterized a slow single-core performance, but it significantly improved with each new generation. Additionally, a lack of hyperthreading provided better multi-core performance compared to Intel and AMD instances with the same number of vCPUs despite slower single-core processing. We also use TUManyBenchmarks to investigate resource isolation in AWS. Our main findings are that AWS uses resource isolation for newer instances to fairly share the L3 cache and memory bandwidth. Moreover, we observed that resource isolation for AMD processors differs from that for Intel and Graviton processors. Furthermore, we created example models that help users select the most cost-effective instance for their use case. We also presented the idea that the service's data can be used to create or extend performance models based on benchmark results. Moreover, we described how our service can simplify benchmarking by removing many responsibilities from a user. The service helps the user to concentrate on the benchmark by, for example, automatically managing infrastructure or configuring communication and synchronization between nodes. Lastly, we compared TUManyBenchmarks to the existing solutions, showing that it combines the advantages of different types of services. All of these showed that TUManyBenchmarks could be helpful at every benchmarking phase: benchmark preparation, execution, and result analysis. For this reason, the implemented service is valuable for everyone who is interested in hardware metrics and who wants to know the instance's possibilities and limitations to choose the most cost-efficient one.

6.2. Future Work

6.2.1. Further improvement of user experience

TUManyBenchmarks helps users choose the most proper instances easily and intuitively via the web application, but we still see an area for improvement. The service allows high freedom of operations on data by providing the query console, allowing users to write their own queries. However, to improve user experience and speed up the writing of queries, the service should offer a set of prepared queries. Such queries would be valuable for the users to familiarize themselves with the service and its possibilities. Additionally, they could use these prepared queries as a base for more complex ones, reducing the time spent writing them.

Another helpful feature would be saving the user's previously executed queries in the local storage and enabling access to them even after reopening the web page later. This functionality would enable users to quickly return to previously written queries without spending time rewriting them again. It would be especially beneficial for complex queries or series of queries that modify the local database. This feature would help restore the database to its previous state and quickly continue work with the service after reopening the web page.

The last feature that would improve user experience is adding more types of plots. Currently, the service supports only scatter and line plots. They cover the many use cases and can be used for most benchmarks. However, expressing all benchmark results using only these two plots is impossible. For this reason, to further improve user experience, the service should support more plot types. For example, users may want to use 3D plots to present results depending on multiple parameters.

6.2.2. Support for other AWS resources

The AWS offers countless different services. Many of them are often used together with EC2, and it is valuable to compare how efficiently different instances can work with the given services. One such service is Simple Storage Service (S3), an AWS service allowing file storage. As it is one of the most commonly used services, there are benchmarks to, for example, analyze maximal download throughput[50]. That allows the user to find the instance best suited for work with S3.

The current implementation does not support creating any resources other than EC2 and required by it (e.g., subnetworks, security groups) during benchmarks. However, we see a value in such benchmarks because EC2 is often used together with other resources like the mentioned S3. Analyzing and comparing how different EC2 instances work with various services would allow users to create more cost-efficient products. Enabling executing benchmarks with other AWS services would require extending the existing configuration file template by new fields, including proper libraries, and implementing functions managing services' state.

6.2.3. Support for other price information

The EC2 instance price depends on multiple factors. The current service implementation only supports information about on-demand and spot instance prices from one region (`us-east-1`) for machines with Linux without preinstalled software. The missing price aspects should be included to deliver better transparency and provide the user with a full view of the possible prices.

The most fundamental price factor is the pricing model - on-demand, spot, or reserved. In order to enable users to make better price comparisons, the service should implement the missing pricing model - reserved instances[3]. The price of reserved instances depends on the three options: time, payment, and convertibility. The customer can buy a reserved instance for one or three years. Also, the user can pay for them in three ways: all upfront, partial upfront, and no upfront. The last option is convertibility, the possibility of exchanging a machine for another with equal or higher value. All combinations of these three options are possible, making 12 options for each instance.

The other price factor is region because the prices can differ between them. Consequently, one region's most cost-efficient virtual machine does not have to be the best in another. Additionally, not all instances are available in each region. For these reasons, the analysis done only for `us-east-1` may not be accurate, and the possibility of analyzing the prices and availability per region can be a valuable asset for the user.

The final factor is the software run on the instance. The prices for different operating systems (e.g., Linux, Windows, Red Hat) can differ. Moreover, AWS offers instances with preinstalled SQL Server versions, which influences the price the customer must pay due to additional software. Also, the cloud customer can buy some instances as dedicated hosts (a physical server reserved for them [2]), which also impacts the price. The customer may need a dedicated host while running software on the license of other companies, such as Oracle. Including these different configurations would also be valuable for some users.

In order to support so many configurations with different prices in each region, the service should have an additional table in DuckDB. This table would replace the current two columns in the `instances` table and contain all the parsed information about prices from all regions. As in the current implementation, the service should still periodically call AWS API because the prices can change over time.

6.2.4. Support for other cloud providers

The cloud is a large market with many different providers. Among them, the most popular is AWS, which belonged to about one-third of cloud market shares in Q2 2024 [46]. However, other influential cloud providers, Microsoft Azure and Google Cloud Platform (GCP), consecutively have 25% and 11% of cloud market shares. The current version of the service supports running benchmarks only on machines offered by AWS. For this reason, a service does not include hardware metrics of a significant part of the market. To achieve one of the main goals - transparency - our service should also allow for executing benchmarks for other vendors' virtual machines. This would enable users to compare offers from different vendors and choose the best one.

It is possible to extend the current implementation to other cloud providers because the service utilizes only the fundamental resources available in each provider's offer, such as virtual machines and private networks. Each cloud provider delivers APIs to create these resources. Their similarity enables the use of abstraction for infrastructure creation and the hiding of implementation details for each provider. This can be achieved, for example, by using a design pattern of the abstract factory. Its task is to provide an interface for initializing groups of related objects without depending on their concrete implementations[42]. While benchmark execution, the service would choose a cloud provider-specific factory implementation to create all required resources in the desired environment and hide all implementation details. The classes responsible for executing benchmarks do not have to be changed as they use generally available technologies such as Ansible and SSH, which can be used with any virtual machine.

Additionally, the service runs benchmark on Ubuntu, an operating system available for all cloud providers, ensuring compatibility between them.

A. Appendix

A.1. Source code

The source code of the service together with benchmark files and configuration files is available in the following GitHub repository: `https://github.com/Szczepaniak-M/TUManyBenchmarks`

A.2. Sample configuration.yml

```
configuration:
    name: Example benchmark name
    description: Example benchmark description
    directory: example
    cron: "0 */4 * * *"
    instance-number: 2
    instance-types:
      - t2.micro
      - t3.micro
    instance-tags:
      - - 1 GiB Memory
        - 1 vCPUs
      - - 0.5 GiB Memory
        - 1 vCPUs
nodes:
  - node-id: 0
    ansible-configuration: ansible-0.yaml
    output-command: python3 format_output.py benchmark_output.txt
  - node-id: 1
    ansible-configuration: ansible-1.yaml
    benchmark-command: ./benchmark-1 > benchmark_output.txt
    instance-type: c7gn.16xlarge
    image-x86: ubuntu-with-benchmark-iso-x86
    image-arm: ubuntu-with-benchmark-iso-arm
  - node-id: 2
    benchmark-command: ./benchmark > benchmark_output.txt
```

```
plots:
  – type: scatter
    title: Example scatter plot
    yaxis: Time [ms]
    series:
      – y: minimum_output_scalar
        legend: Minimum value
      – y: maximum_output_scalar
        legend: Maximum value
  – type: line
    title: Example line plot
    xaxis: Execution number
    yaxis: Time [ms]
    yaxis–log: 2
    series:
      – x: increasing_values
        y: output_array
        legend: Some name of series
      – x: some_custom_x_input
        y: other_output_array
        legend: Some other name
series–other:
  – non_plot_series
```

Listing A.1: configuration.yml: Full example.

A.3. Sample MongoDB instance document

```
{
    "_id": "66bc6b23a81e8672065b06bb",
    "name": "t2.micro",
    "vCpu": 1,
    "memory": 1,
    "network": "Low to Moderate",
    "tags": [
        "Family t2",
        "1 vCPUs",
        "1 GiB Memory",
        "Low to Moderate Network",
        "i386",
        "x86–64",
        "Hypervisor Xen"
    ],
    "benchmarks": [
        {
            "benchmark": "66a0ffdef57a93fb782a29c3",
            "results": [
                {
                    "timestamp": 1723624583,
                    "values": {
                        "maxRoundTripServer": 0.47,
                        "roundTripServer": [0.43, 0.44, 0.45, 0.47, 0.41, 0.42, 0.4, 0.41, 0.42, 0.43 ]
                    }
                }
            ]
        }
    ]
}
```

Listing A.2: Example of MongoDB's document storing information about instance.

A.4. Multi-core SPEC CPU 2017 IntRate Peak metric results

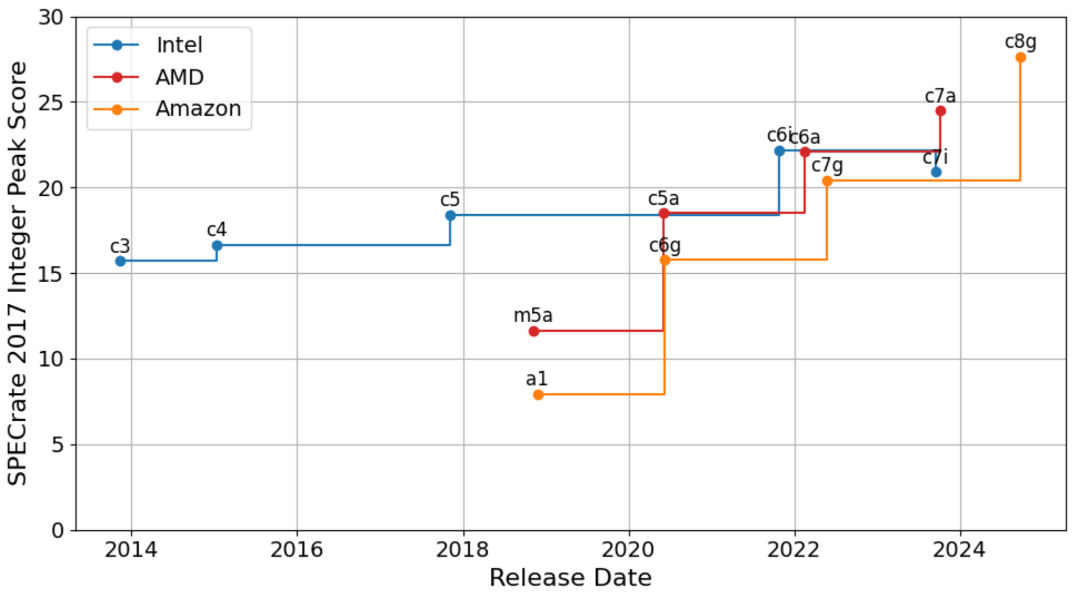


Figure A.1.: SPECrate 2017 Integer Peak values over time for multi-core processing using half of the vCPUs.

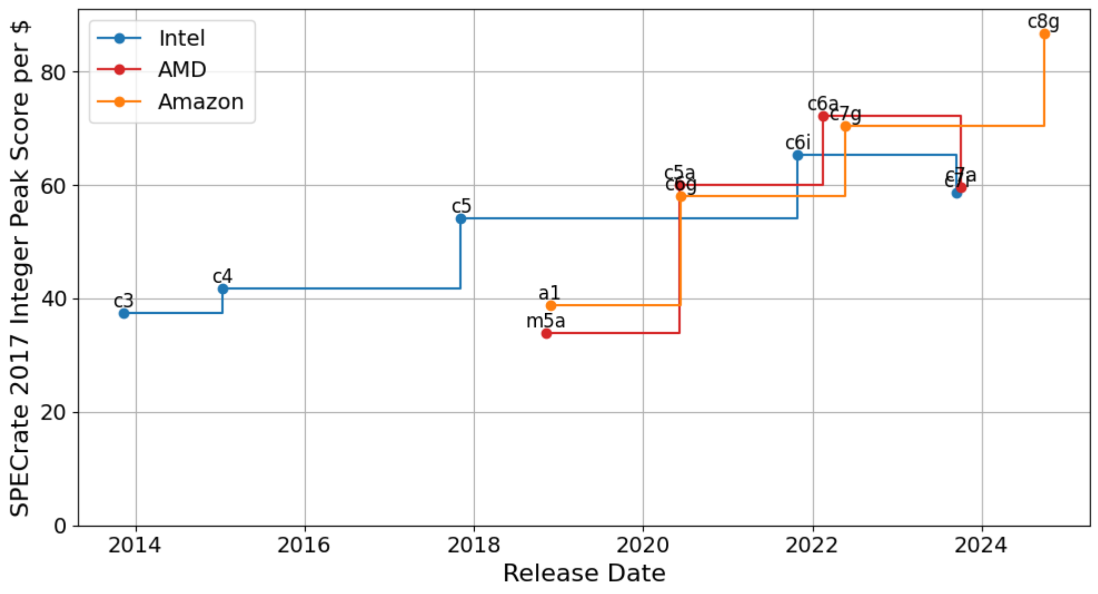


Figure A.2.: SPECrate 2017 Integer Peak per Dollar values over time for multi-core processing using half of the vCPUs.

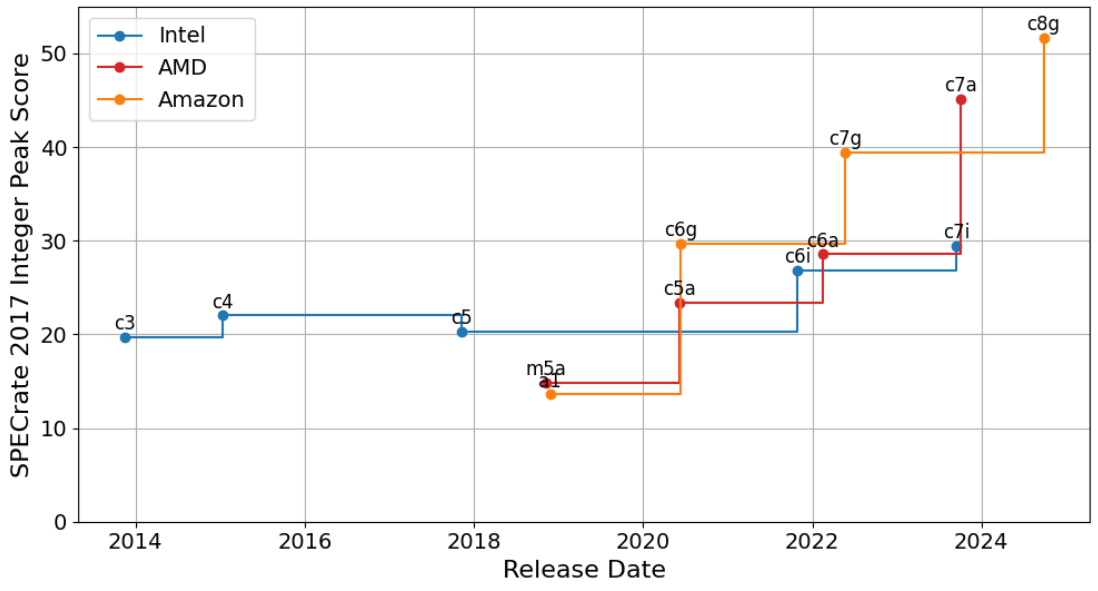


Figure A.3.: SPECrate 2017 Integer Peak values change over time for multi-core processing using all vCPUs.

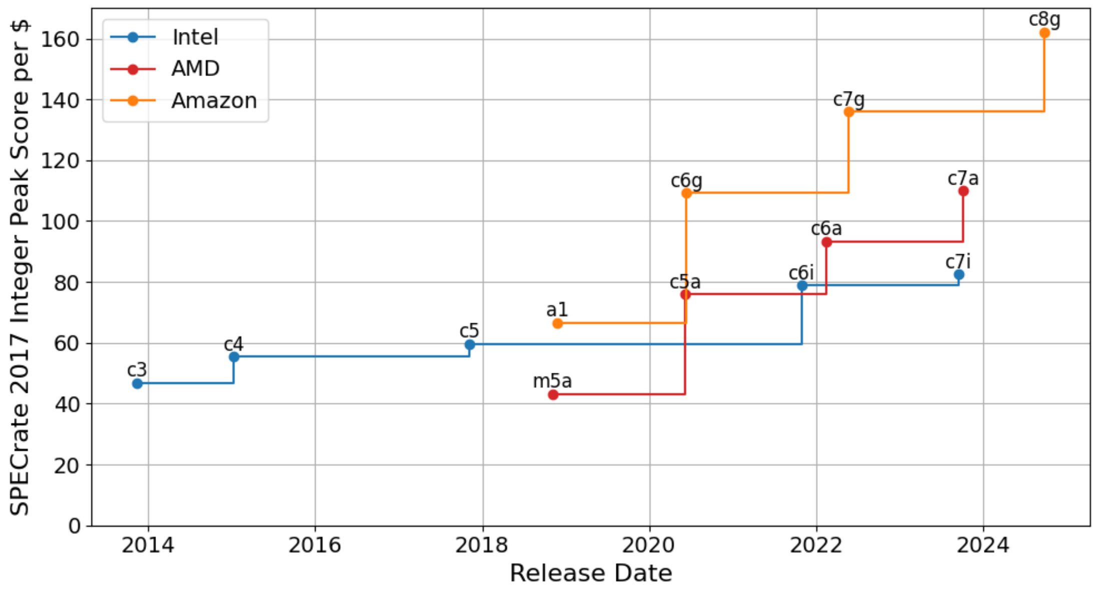


Figure A.4.: SPECrate 2017 Integer Peak per Dollar values over time for multi-core processing using all vCPUs.

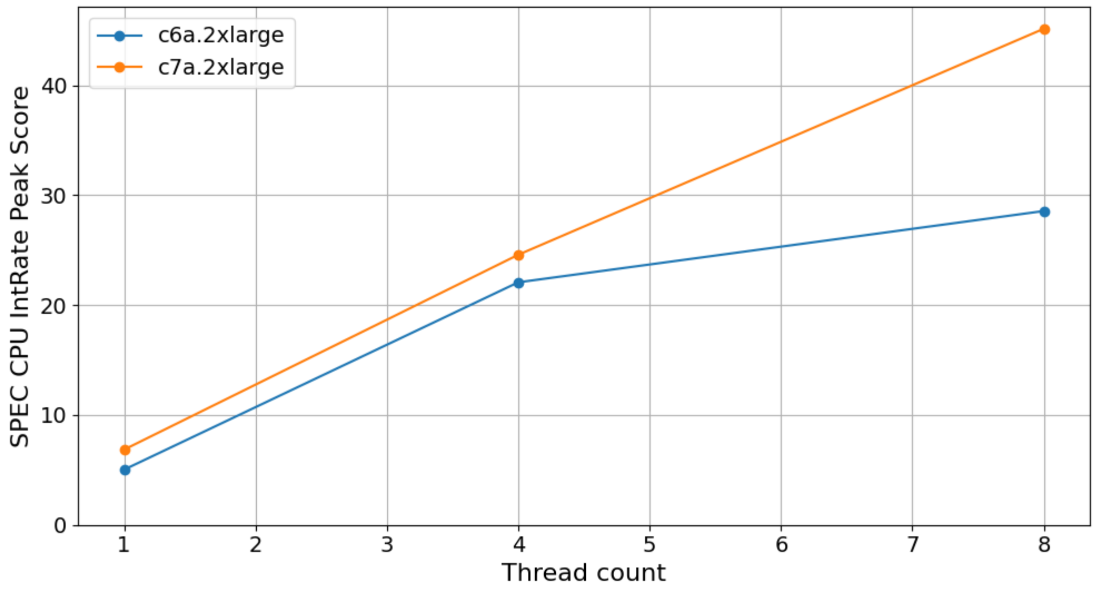


Figure A.5.: SPECrate 2017 Integer Peak values for different number threads for c6a.2xlarge (hyperthreading) and c7a.2xlarge (no hyperthreading).

A.5. Additional cache latency benchmark results

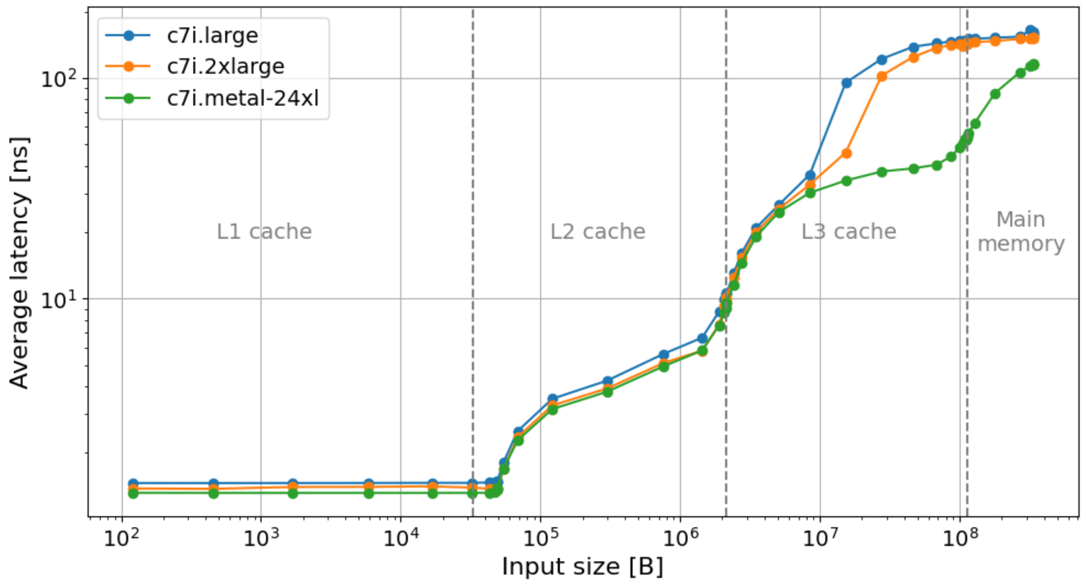


Figure A.6.: Cache latency benchmark results for instance family c7i.

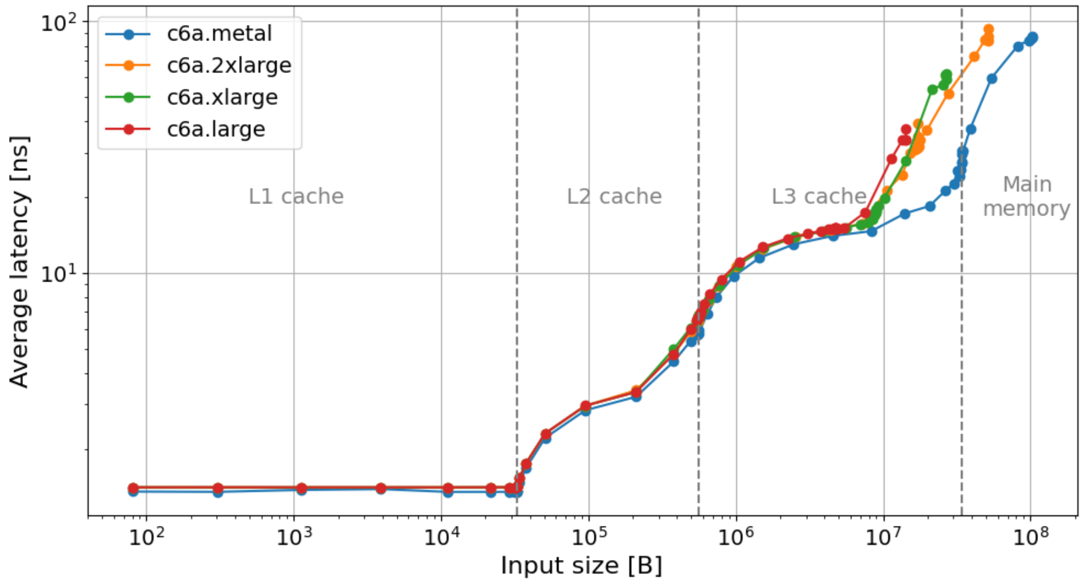


Figure A.7.: Cache latency benchmark results for instance family c6a.

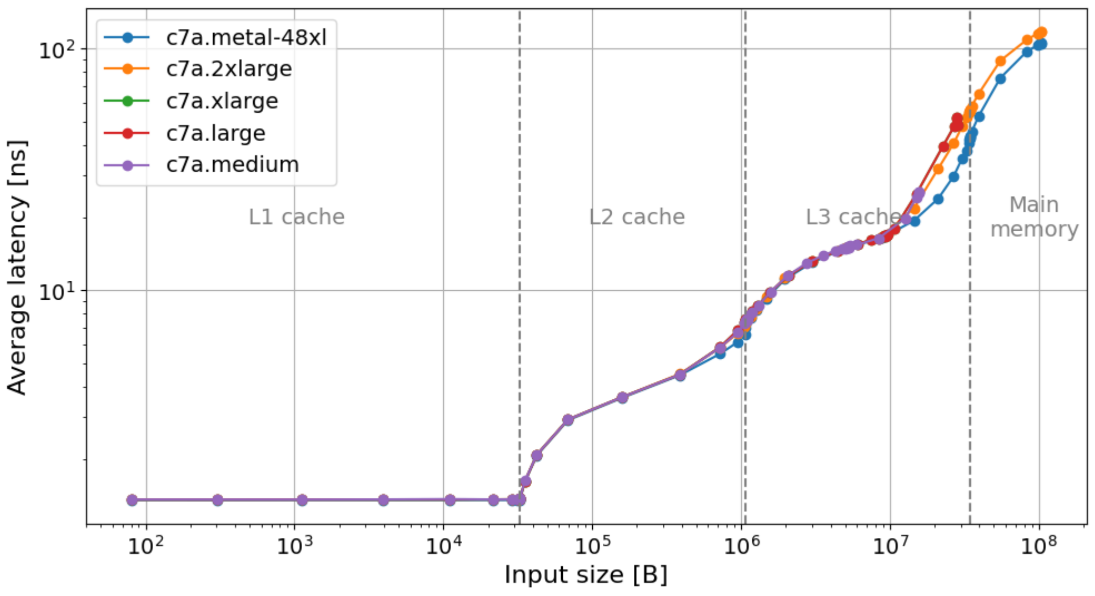


Figure A.8.: Cache latency benchmark results for instance family c7a.

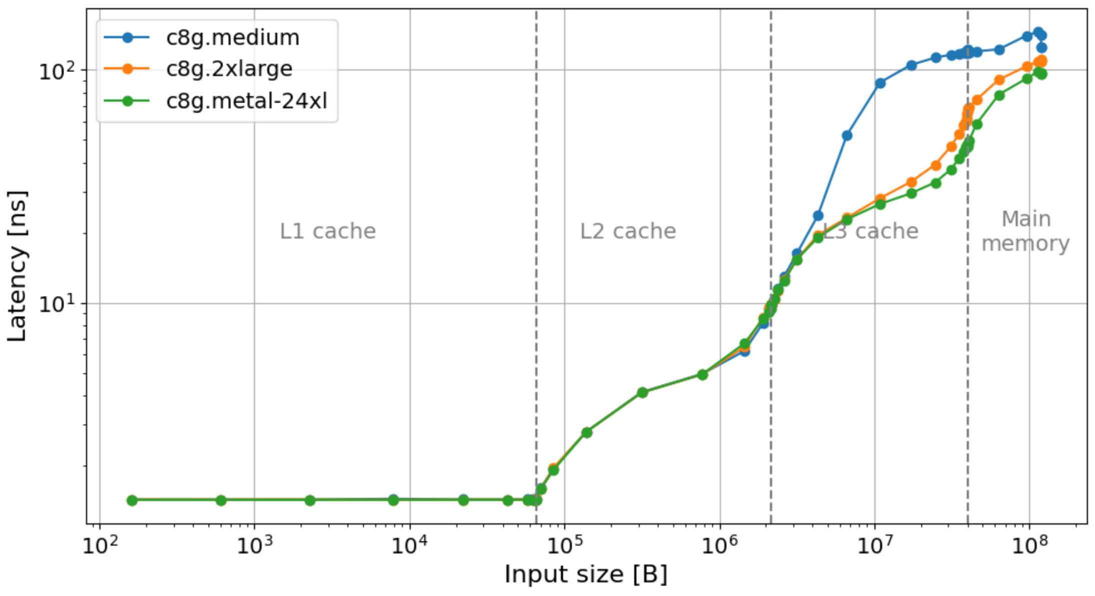


Figure A.9.: Cache latency benchmark results for instance family c8g.

A.6. Additional memory bandwidth benchmark results

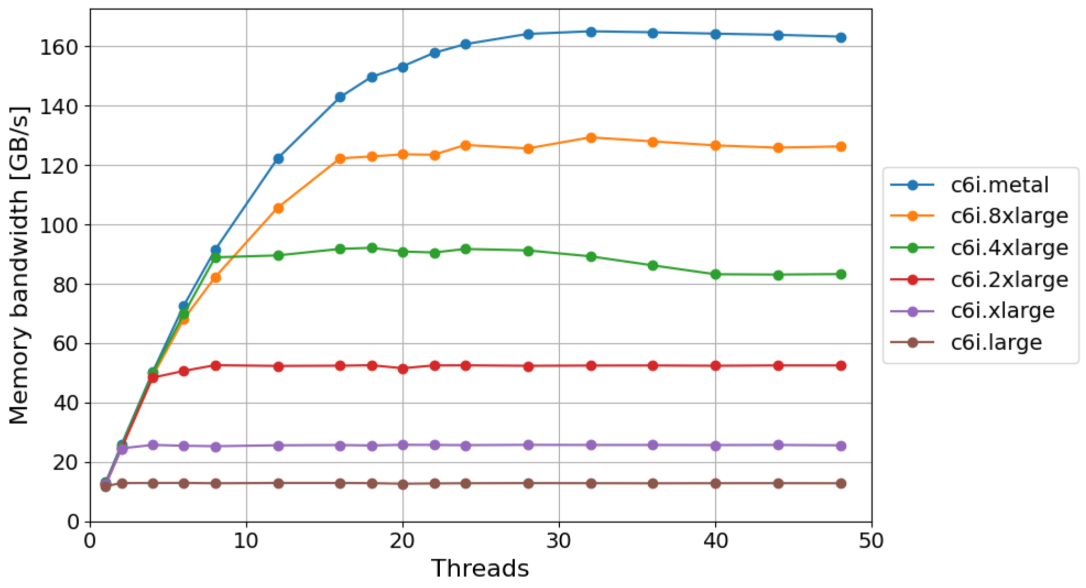


Figure A.10.: Memory Bandwidth benchmark results for instance family c6i.

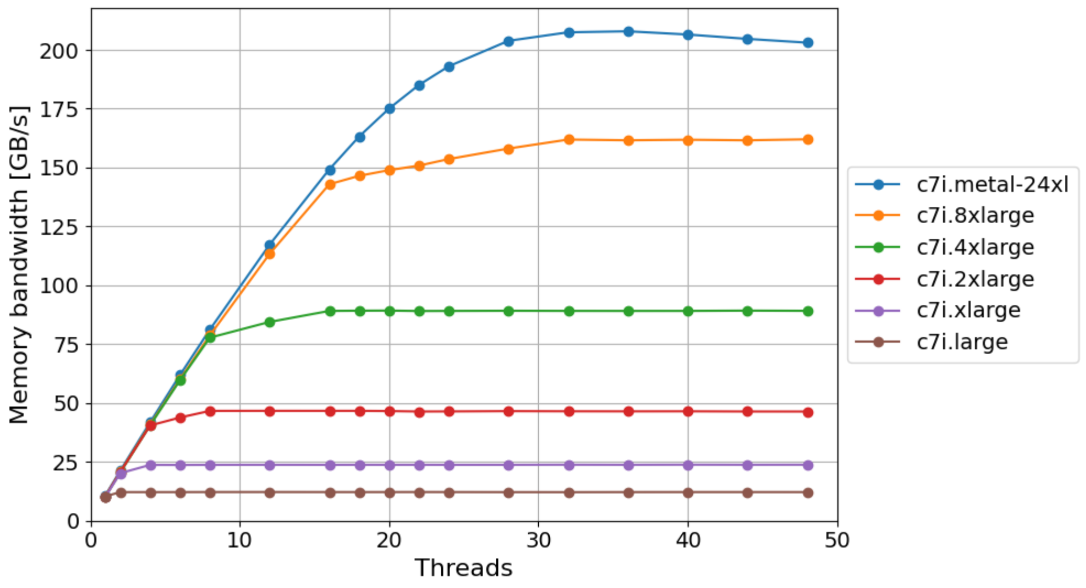


Figure A.11.: Memory Bandwidth benchmark results for instance family c7i.

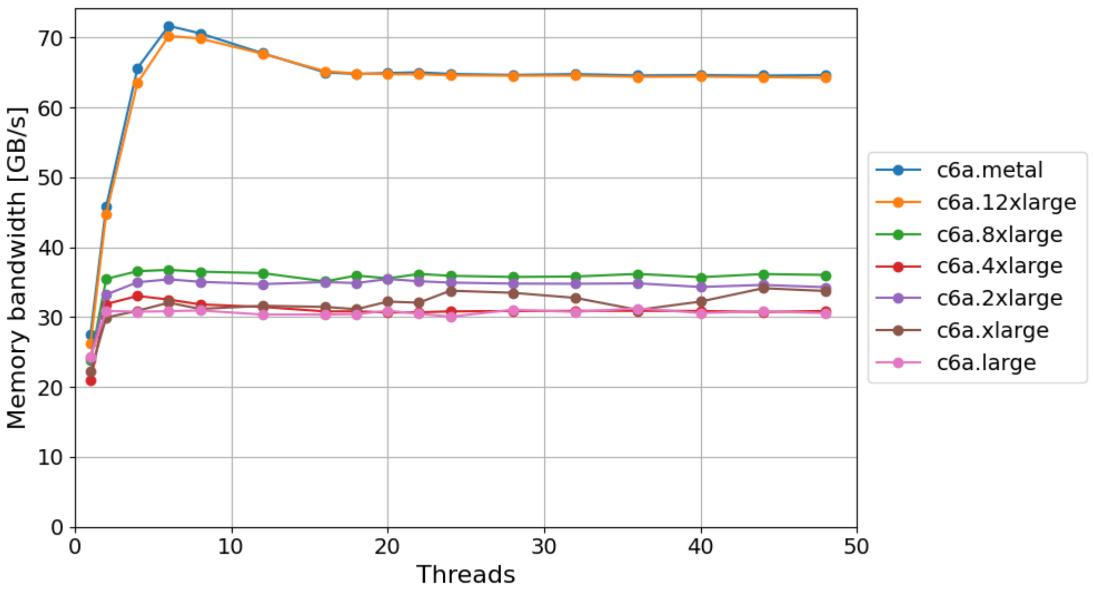


Figure A.12.: Memory Bandwidth benchmark results for instance family c6a.

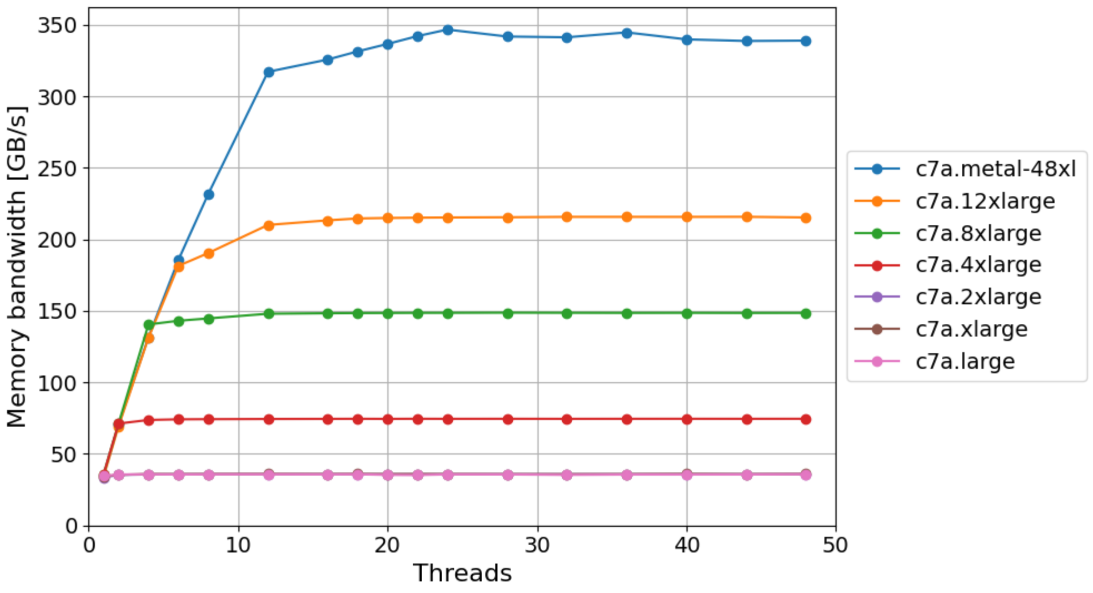


Figure A.13.: Memory Bandwidth benchmark results for instance family c7a.

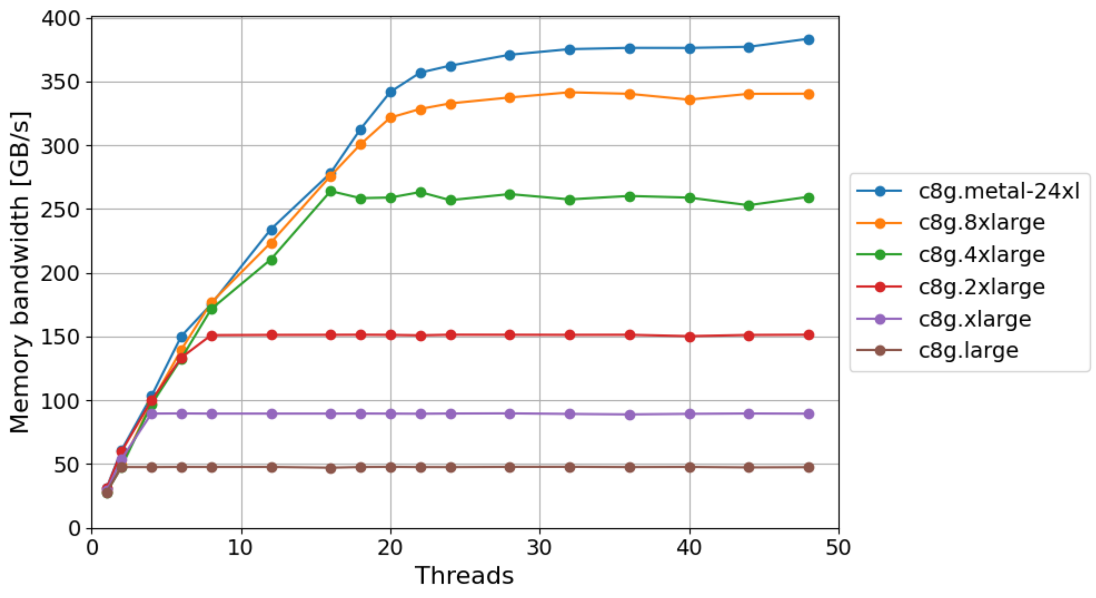


Figure A.14.: Memory Bandwidth benchmark results for instance family c8g.

A.7. View calculating GB downloaded by instance in given time

```sql
CREATE OR REPLACE VIEW cumulative_bandwidth_view AS
WITH unnest_benchmarks AS (
    SELECT id, unnest(benchmarks) AS benchmark
    FROM instances
),
bandwidth_values AS (
    SELECT
        id,
        (benchmark->>'$.results[*].values.sum_bandwidth[*]')::DOUBLE[]
            AS bandwidth_array
    FROM unnest_benchmarks
    WHERE benchmark->>'$.name' = 'Network␣Bandwidth'
),
bandwidth_values_with_index AS (
    SELECT
        id,
        generate_subscripts(bandwidth_array, 1)::INT AS idx,
        unnest(bandwidth_array) AS bandwidth,
    FROM bandwidth_values
),
bandwidth_averages AS (
    SELECT id, idx, avg(bandwidth) AS average
    FROM bandwidth_values_with_index
    GROUP BY id, idx
)
SELECT
    id AS instance_id,
    idx AS time,
    SUM(average) OVER (PARTITION BY id ORDER BY idx) / 8
        AS cumulative_bandwidth,
FROM bandwidth_averages
```

Listing A.3: DuckDB SQL code to create view with GB downloaded by instance in given time.

Abbreviations

AMI	Amazon Machine Image
AWS	Amazon Web Services
CAT	Cache Allocation Technology
CD	Continuous Delivery
CI	Continuous Integration
EC2	Elastic Compute Cloud
GCP	Google Cloud Platform
MBA	Memory Bandwidth Allocation
OLAP	Online Analytical Processing
PR	Pull Request
QoS	Quality of Service
SDK	Software Development Kit
SPA	Single Page Application
S3	Simple Storage Service

VPC	Virtual Private Cloud

List of Figures

List of Listings

List of Tables

Bibliography

[1] Alexey Milovidov. *ClickBench: A Benchmark for Analytical Databases.* https://benchmark.clickhouse.com/ Accessed: 2024-08-24. 2022.

[2] Amazon Web Services, Inc. *Amazon EC2 Dedicated Hosts.* https://aws.amazon.com/ec2/dedicated-hosts/ Accessed: 2024-09-26.

[3] Amazon Web Services, Inc. *Amazon EC2 Reserved Instances.* https://aws.amazon.com/ec2/pricing/reserved-instances/ Accessed: 2024-11-06.

[4] Amazon Web Services, Inc. *AWS EC2 instance network bandwidth Documentation.* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html Accessed: 2024-09-16.

[5] Amazon Web Services, Inc. *AWS Elastic Compute Cloud (EC2) Documentation.* https://docs.aws.amazon.com/ec2/ Accessed: 2024-08-26.

[6] Amazon Web Services, Inc. *AWS IP Addressing Documentation.* https://docs.aws.amazon.com/vpc/latest/userguide/vpc-ip-addressing.html Accessed: 2024-08-26.

[7] Amazon Web Services, Inc. *AWS SDK Documentation.* https://aws.amazon.com/sdk-for-kotlin/ Accessed: 2024-08-24.

[8] Amazon Web Services, Inc. *AWS Service Quotas Documentation.* https://docs.aws.amazon.com/servicequotas/latest/userguide/intro.html Accessed: 2024-10-12.

[9] Amazon Web Services, Inc. *AWS Spot Instances Documentation.* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html Accessed: 2024-08-26.

[10] Amazon Web Services, Inc. *AWS Virtual Private Cloud (VPC) Documentation.* https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html Accessed: 2024-08-24.

[11] Amazon Web Services, Inc. *AWS VPC Internet Gateway Documentation.* https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Internet_Gateway.html Accessed: 2024-08-26.

[12] Amazon Web Services, Inc. *AWS VPC Route Tables Documentation*. https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Route_Tables.html Accessed: 2024-08-26.

[13] Amazon Web Services, Inc. *AWS VPC Security Groups Documentation*. https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html Accessed: 2024-08-26.

[14] Amazon Web Services, Inc. *AWS VPC Subnets Documentation*. https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html Accessed: 2024-08-24.

[15] AMD, Inc. *AMD64 Technology Platform Quality of Service Extensions*. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf Accessed: 2024-10-28. Feb. 2022.

[16] ApexCharts. *ApexCharts.js Documentation*. https://apexcharts.com/ Accessed: 2024-08-31.

[17] Ben Manes. *Caffeine Cache*. https://github.com/ben-manes/caffeine Accessed: 2024-08-31.

[18] J. Cahoon, W. Wang, Y. Zhu, K. Lin, S. Liu, R. Truong, N. Singh, C. Wan, A. Ciortea, S. Narasimhan, and S. Krishnan. "Doppler: automated SKU recommendation in migrating SQL workloads to the cloud." In: *Proc. VLDB Endow.* 15.12 (2022), pp. 3509–3521. ISSN: 2150-8097. DOI: 10.14778/3554821.3554840.

[19] I. ClickHouse. *ClickHouse Hardware Benchmark*. https://benchmark.clickhouse.com/hardware/. Accessed: 2024-09-19. 2024.

[20] Colin Ian King, et al. *stress-ng*. https://github.com/ColinIanKing/stress-ng. Accessed: 2024-09-19.

[21] S. Cores. *Spare Cores - Cloud Compute Resources*. https://sparecores.com/servers. Accessed: 2024-09-19. 2024.

[22] DuckDB Foundation. *DuckDB JSON Extension Documentation*. https://duckdb.org/docs/extensions/json Accessed: 2024-09-02.

[23] ESnet and L. B. N. Laboratory. *iPerf - The TCP, UDP, and SCTP Network Bandwidth Measurement Tool*. Accessed: 2024-10-01.

[24] GitHub, Inc. *GitHub Actions Documentation*. https://docs.github.com/en/actions/ Accessed: 2024-08-24.

[25] Google. *Angular Documentation*. https://angular.dev/overview. Accessed: 2024-09-29.

[26] Grand View Research Inc. *Cloud Computing Market Size, Share & Trends Analysis Report By Service (SaaS, IaaS), By Deployment, By Enterprise Size, By End-use, By Region, And Segment Forecasts, 2024 - 2030*. https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry Accessed: 2024-09-01. 2024.

[27] A. J. Herdrich, M. D. Cornu, and K. M. Abbasi. *Introduction to Memory Bandwidth Allocation*. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html. Accessed: 2024-10-21. 2019.

[28] JetBrains. *Kotlin Coroutines Documentation*. https://kotlinlang.org/docs/coroutines-basics.html Accessed: 2024-08-31.

[29] JetBrains. *Kotlin Documentation*. https://kotlinlang.org/docs/home.html Accessed: 2024-08-31.

[30] A. Kohn, D. Moritz, M. Raasveldt, H. Mühleisen, and T. Neumann. "DuckDB-wasm: fast analytical processing for the web." In: *Proceedings of the VLDB Endowment* 15.12 (2022), pp. 3574–3577. ISSN: 2150-8097. DOI: 10.14778/3554821.3554847.

[31] V. Leis and M. Kuschewski. "Towards cost-optimal query processing in the cloud." In: *Proc. VLDB Endow.* 14.9 (2021), pp. 1606–1612. ISSN: 2150-8097. DOI: 10.14778/3461535.3461549.

[32] A. Mahgoub, A. M. Medoff, R. Kumar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi. "OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud." In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 189–203. ISBN: 978-1-939133-14-4.

[33] Marcin Kolny. *Scaling Up the Prime Video Audio/Video Monitoring Service and Reducing Costs by 90%*. https://web.archive.org/web/20240805183535/https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90 Accessed: 2024-09-13. Amazon Web Services, Inc., 2024.

[34] Mellanox Technologies. *SockPerf: Network Benchmarking Utility*. https://github.com/Mellanox/sockperf. Accessed: 2024-10-01.

[35] Microsoft. *TypeScript - Overview*. https://www.typescriptlang.org/. Accessed: 2024-09-29.

[36] MongoDB, Inc. *MongoDB Aggregation Pipeline Documentation*. https://www.mongodb.com/docs/manual/core/aggregation-pipeline/ Accessed: 2024-08-31.

[37] MongoDB, Inc. *MongoDB Documentation*. https://www.mongodb.com/docs/manual/ Accessed: 2024-08-31.

[38] K. T. Nguyen. *Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family*. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html Accessed: 2024-10-21.

[39] Nils Knieling. *Google Cloud Compute Engine Machine Type Comparison*. https://gcloud-compute.com/instances.html. Accessed: 2024-09-14.

[40] M. Raasveldt and H. Mühleisen. "DuckDB: an Embeddable Analytical Database." In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1981–1984. ISBN: 9781450356435. DOI: 10.1145/3299869.3320212.

[41] Red Hat, Inc. *Ansible Documentation*. https://docs.ansible.com/ Accessed: 2024-08-24.

[42] Refactoring Guru. *Abstract Factory Design Pattern*. https://refactoring.guru/design-patterns/abstract-factory. Accessed: 2024-09-28.

[43] Standard Performance Evaluation Corporation (SPEC). *SPEC CPU 2017 Benchmark*. https://www.spec.org/cpu2017/. Accessed: 2024-10-01.

[44] Standard Performance Evaluation Corporation (SPEC). *SPEC CPU®2017 Overview*. https://www.spec.org/cpu2017/Docs/overview.html. Accessed: 2024-10-01.

[45] Standard Performance Evaluation Corporation (SPEC). *SPEC/OSG Result Search Engine*. https://www.spec.org/cgi-bin/osgresults. Accessed: 2024-10-07.

[46] Synergy Research Group. "Cloud Market Growth Stays Strong in Q2 While Amazon, Google and Oracle Nudge Higher." In: *Synergy Research Group* (2024). https://www.srgresearch.com/articles/cloud-market-growth-stays-strong-in-q2-while-amazon-google-and-oracle-nudge-higher Accessed: 2024-09-28.

[47] The Apache Software Foundation. *Apache MINA SSHD Documentation*. https://mina.apache.org/sshd-project/ Accessed: 2024-08-31.

[48] Vantage. *Amazon EC2 Instance Comparison*. https://instances.vantage.sh/. Accessed: 2024-09-14.

[49] Vantage. *Azure Instance Comparison*. https://instances.vantage.sh/azure/. Accessed: 2024-09-14.

[50] D. Vassallo. *s3-benchmark*. https://github.com/dvassallo/s3-benchmark. Accessed: 2024-09-19.

[51] VMware Tanzu. *Spring Framework*. `https://spring.io/projects/spring-framework`. Accessed: 2024-09-28.

[52] W3C. *WebAssembly - Documentation*. `https://webassembly.org/`. Accessed: 2024-09-30.