Universiteit Leiden
The Netherlands

Opleiding Informatica

Is the Linux Completely Fair Scheduler Ready for Serverless Workloads?

N.J.L. Boonstra

Supervisors:
Alexandru Uta & Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

29/06/2021

Abstract

Serverless computing changed the complexity of workloads on the underlying machines. Serverless workloads consist of thousands of short-running jobs, executed in either a container, or virtual machine, which was not done so extensively before. Previous research has shown that schedulers influence the performance of the workload. In this work, we introduce a novel benchmark, `fc-microbenchmark`, to evaluate the performance of the Linux scheduler on serverless workloads. The benchmark is built on top of AWS Firecracker, which manages microVMs for the Amazon Lambda platform. We use a Poisson arrival process to increase the realism of the benchmark. Using the benchmark, we compared the default Linux scheduler, called CFS, to other scheduling algorithms and policies available in the Linux kernel.

We find that CFS performs well on CPU-bound workloads, but is outperformed on workloads that introduce memory-intensive jobs. As such, we conclude that CFS needs further improvements to handle serverless workloads. We propose multiple pointers for further research, namely: dynamic selection of the scheduler depending on the type of the workload, the effect of RR and FIFO on the preservation of locality and finally the time spent waiting by a process that is not preempted when scheduled by RR and FIFO.

Contents

1 Introduction

Serverless computing is an evolution in cloud computing, with the goal to make cloud computing more accessible and fine-grained [10]. Until recently cloud computing applications consisted of monolithic applications. Now, cloud computing shifts towards micro-services [12] and serverless architectures [10]. Currently, there are several providers for serverless services, notably AWS Lambda [4], Google Cloud Functions [13], Microsoft Azure Functions [17], and IBM Cloud Functions [14].

The evolution of cloud computing to serverless computing started with the emergence of virtualisation, containerisation and events as triggers. With virtualisation, the *bare metal* machine is abstracted and resource-sharing across multiple users and applications is enabled. Containerisation provides convenient orchestration and fostering of digital ecosystems. Finally, events as triggers for functions increase the flexibility of a computer program when conditions change. Before events as triggers, computer programs were designed to follow a specified path [10].

When a function is invoked by a trigger in AWS Lambda, it spawns a new microVM in which it executes the function. In production use, millions of workloads are powered by microVMs, per month. In order to efficiently spawn these microVMs, AWS Lambda uses a custom, open-source virtual machine manager called Firecracker [3].

As serverless computing relies on events as trigger and containerisation, with an entire ecosystem, to facilitate the execution of the serverless functions, running in each container, the complexity of workloads increased. The increased complexity and scale of workloads means that the task of scheduling has also increased in complexity. Currently, the *Completely Fair Scheduler (CFS)* is the default scheduling algorithm in the Linux kernel [1]. The scheduler is responsible for assigning a job to a CPU/CPU-core, on which it can be executed. It attempts to ensure that every job gets an equal share of CPU time. If the scheduler fails, then thread starvation occurs. This means that jobs take much longer, or do not finish [23]. As the CFS algorithm was introduced in Linux kernel version 2.6.23, released in 2007, before the emergence of serverless computing, we hypothesise that CFS is not optimised for serverless workloads.

The hypothesis leads to the following research question: *Is the Linux scheduler ready for serverless workloads?* In order to answer the main question, we first answer two sub-questions, namely:

1. How to check whether the Linux scheduler (Completely Fair Scheduler) performs well for serverless workloads?

2. How does the Linux scheduler perform for serverless workloads?

Our technical contribution, namely a custom benchmark, is the answer to the first sub-question. This benchmark will generate results, on which we do a performance analysis. This performance analysis gives an answer to the second sub-question. The combination of these two answers determines whether the Linux scheduler is ready for serverless workloads.

In the next section, Section 2, we formulate a precise definition for serverless computing and give an explanation of the Linux scheduler and other scheduling algorithms present in the Linux kernel. Then, in Section 3 we discuss related work. In Section 4 we explain the design of the benchmark, the technical contribution of this thesis. After this, in Section 5 we describe the experimental set-up and show the results of the experiments, which we discuss in Section 6. Lastly, we present our conclusions in Section 7.

2 Background

As this thesis relies on the concepts of serverless computing and the Linux scheduler, we provide a definition for serverless computing in Section 2.1. Subsequently, we briefly explain AWS Firecracker in Section 2.2. After this, we delve deeper into how scheduling is organised in the Linux kernel in Section 2.3. Then, we give a summary of the mechanics of the Linux scheduling algorithm, the Completely Fair Scheduler in Section 2.4. Finally, we briefly describe the other scheduling algorithms and policies we use in this work, namely FIFO, BATCH and Round-Robin in Section 2.5.

2.1 Serverless Computing

Serverless computing is a recent development in cloud computing [10]. Defining the term *serverless computing* proves to be difficult, as the definitions overlap with other paradigms, such as Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [6].

Baldini, *et al.* [6] define serverless computing by comparing the control the developer has over the infrastructure. In Infrastructure-as-a-Service (IaaS), the developer has full control over both the application, and the infrastructure. On the other hand, PaaS and SaaS abstract the infrastructure and provide the developer with prepackaged software. The difference in abstraction of the infrastructure for PaaS and serverless stems from the fact that for serverless, the infrastructure is equal for all users, namely simple containers or virtual machines that execute the serverless function, whereas the provider may tailor the infrastructure to the specific application for PaaS. As such, the developer has less control in PaaS and SaaS than in IaaS. Serverless computing is placed in-between these paradigms and provides the developer control over the application, but still abstracts the infrastructure. For a visual representation, see Figure 1.

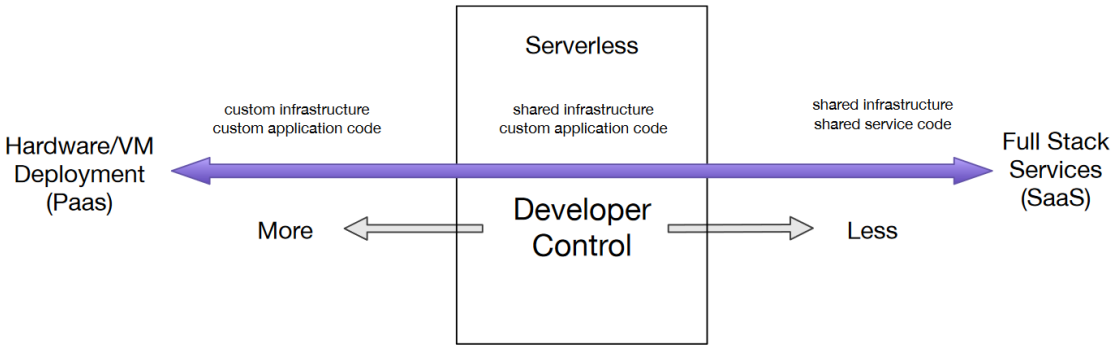


Figure 1: Relative control the developer has in different cloud computing paradigms. Figure taken from Baldini, *et al.* [6].

Another definition for serverless computing is proposed by van Eyk, *et al.* [10]:

> "Serverless Computing is a form of cloud computing which allows users to run event-driven and granularly billed applications, without having to address the operational logic"

In adddition to this definition, serverless computing providers may implement the Functions-as-a-Service (FaaS) model, for which van Eyk, *et al.* proposes the following definition:

> "FaaS is a form of serverless computing where the cloud provider manages the resources, lifecycle, and event-driven execution of user-provided function"

In this work, we use the backbone of the FaaS service provided by Amazon Web Services, the AWS Firecracker virtual machine manager [5].

2.2 Firecracker

Before Firecracker, cloud providers faced challenges when providing multi-tenancy on hosts, such as the noisy neighbour effect, where the computational demands of one workload affect the execution time of the other. Another challenge is the isolation of workloads, where one workload cannot access data of the other workload. Common solutions are to use hypervisor based virtualisation, or by using Linux containers. However, virtualisation comes with possibly unacceptable overhead, as serverless workloads need thousands of VMs. Containers, on the other hand, come with security risks. These can partly be mitigated by limiting syscalls, but this may break existing code. Firecracker aims to overcome the challenges, without the explained trade-offs [3].

Firecracker uses the Linux KVM to spawn and manage microVMs, like other virtual machine managers (VMMs). However, it differentiates itself from other VMMs by offering a minimalist virtual machine, stripped from unnecessary emulated devices and support for guest functionality. This reduces the attack surface and the memory footprint of the virtual machine (around 5MB per microVM). Furthermore, the minimal functionality greatly improves the time needed to start the application code to less than 125 milliseconds. Due to these improvements, up to 150 microVMs can be spawned per second on a single host [3]. This means the provider spends less time on orchestration of VMs and more time on executing customer code, which can be billed.

2.3 Scheduling in Linux

The approach the Linux kernel takes on scheduling changed with the introduction of the Completely Fair Scheduler. Before, the kernel supported only one algorithm (the O(1) scheduler), but now, the scheduler has a modular design. This means that multiple algorithms can be used [23].

Due to the new design, the terminology has changed. The scheduler core provides a framework in which a *scheduling class* implements the *logic of the algorithm*. Another feature is the ability to govern a scheduling class, by using a *scheduling policy*. An example of a policy is BATCH (see

Section 2.5), which tunes the default scheduler, CFS (see Section 2.4), for better handling of batched workloads [23].

In this work, we use the terms *scheduling class*, *scheduling policy*, *scheduling algorithm* interchangeably.

2.4 Completely Fair Scheduler

The Linux *Completely Fair Scheduler* was first introduced into the mainstream kernel in release 2.6.23 as the default scheduling algorithm. The CFS aims to "model an ideal, precise multi-tasking CPU on real hardware", with the goal to assure every task receives an equal slice of CPU-time [23]. For the sake of simplification, this explanation of the mechanism of CFS assumes a single-core CPU.

On an 'ideal' single-core CPU, every task that runs concurrently receives an equal share of actual resources. This means that for n tasks, the amount of CPU time for each task equals $\frac{1}{n}$. However, on a real-life single-core CPU, the resources of the CPU can only be allocated to one task at a time. This means that all other tasks have to wait until that task is completed, or preempted. When no scheduling occurs, this means that all tasks run in a serial manner. Thus, first task A must be finished, after which task B can start. If task A runs infinitely long, then task B will never receive any CPU-time. Such a situation is called *thread-starvation*. This is undesirable behaviour for a CPU that runs interactive tasks.

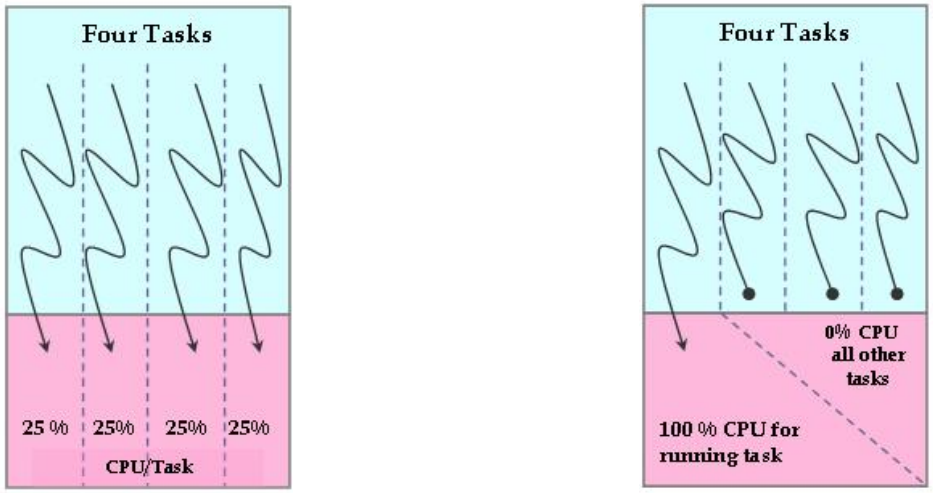


Figure 2: An illustration of the distribution of resources on an ideal multi-tasking single-core CPU on the left and a non-ideal CPU on the right. On such an ideal CPU, each task gets an equal slice of CPU power. Also, the tasks run truly parallel. However, a real-life CPU cannot run tasks truly parallel, as illustrated. Retrieved from: https://www.linuxjournal.com/node/10267.

A scheduler is designed to circumvent this problem. The Completely Fair Scheduler attempts this by creating a queue that is ordered by a metric, the *vruntime*, which is tracked for each task. The *vruntime* is the *virtual* run time of a process, inferred from the total CPU time the task already received, divided by the priority of the task. Internally, priority is named the *niceness* of a task,

with a high niceness corresponding to a low priority and vice versa. This calculation of the run time results in that the vruntime increases relatively slower for higher priority tasks, then it does for low priority tasks [8].

The queue, ordered by the vruntime, is implemented as a red-black tree. When a task is pre-empted, the new vruntime of this task is calculated and the tree is reordered, if necessary. After reordering, the leftmost task in the tree has the smallest vruntime and will be picked as the next task to execute [23].

2.5 Other Linux Scheduling Algorithms

In this work, we compare the CFS scheduler from Section 2.4 with a subset of other scheduling classes present in the Linux kernel. Our subset of these scheduling classes include the First-In-First-Out (FIFO) scheduler, the Round-Robin (RR) scheduler. In addition to these classes, we also compare the BATCH scheduling policy of CFS.

1. The **First-In-First-Out** (**FIFO**) scheduler is a simple algorithm, that does not use time slicing and does not pre-empt any task. When a task is scheduled by this algorithm, it will not be preempted, unless the task itself performs some blocking IO. When this happens, the task will be put first in queue, to ensure it gains control as soon as another task either completes, performs blocking IO, or yields. A tasks that voluntarily yields control to the scheduler will be put at the back of the queue, in contrast to a blocking task [1]. We note that Firecracker does not call `sched_yield`, which means the benchmark we created does not yield control back to the scheduler [5].

2. As an enhancement of FIFO, the **Round-Robin** (**RR**) scheduler shares many of the properties we described for FIFO. The only difference with RR, is that each task may only run for a pre-specified portion of time, the *time quantum*. If a task is not finished in the time quantum, it will be preempted and put at the end of the queue. This cycle will repeat until the task finishes execution [1].

3. Finally, the **BATCH** scheduling policy uses the same scheduling class as CFS (see Section 2.4). An important assumption used in this algorithm, is that every task is CPU-intensive. Any task that needs to wake-up often, is penalised in following scheduling decisions, effectively disfavouring it for tasks that never need waking up. Due to these differences from CFS, BATCH is useful for non-interactive workloads that want a deterministic scheduler and less scheduling overhead caused by interactivity [1].

3 Related Work

In this section, we first discuss previous research on benchmarks for serverless systems. After this, we discuss benchmarks and approaches that focus on evaluating the performance of the scheduler. To our knowledge, there is no prior research that combines both of these concepts, although van Eyk, *et al.* pose the scheduler as a possible subject for research and benchmarking [24].

In current literature, serverless computing is mainly being evaluated in terms of resource consumption, start-up latency, trigger latency, and concurrency and elasticity [24]. For example, Wang, *et al.*, provides a comprehensive comparison of performance amongst different serverless providers. Furthermore, the paper investigates the isolation of resources on each platform. The authors conclude with pointers for research, and show several weaknesses with regard to billing that can be exploited [25]. However, the paper does not investigate the underlying system, but rather investigates the underlying platform.

Research on the underlying performance of the scheduler is only done for bare-metal machines, but not for serverless platforms. For example, Bouron, *et al.*, evaluate the scheduling by comparing the Linux Completely Fair Scheduler to the FreeBSD ULE scheduler. This paper does not only evaluate the performance of the different scheduling algorithms, but also evaluates the fairness of the schedulers. One of the key observations of this paper is the fact that the performance of the scheduler depends on the type of workload. ULE performed significantly better on Apache workloads, whereas CFS performed significantly better on a scimark workload [8]. This paper inspired us to research the impact of the scheduler on serverless workloads.

To model the arrival of individual processes in a serverless environment, literature shows that a Poisson arrival process can be used to calculate the time between each arrival, the **inter-arrival time**. Mahmoudi, *et al.* show that using a Poisson arrival process to model serverless arrival rates is greatly in tune with actual measurements of the arrival rate in experiments [15]. Therefore, we use a Poisson arrival process in the `fc-microbenchmark`.

4 Technical Contribution

This section describes the micro-benchmark we specifically designed for this project, the *fc-microbenchmark* [7]. The benchmark uses the *pipenv* [19] to ensure the Python-environment is equal for each system running the benchmark, including the correct versions of the dependencies of all Python packages. Furthermore, the benchmark heavily relies on the AWS Firecracker microVM manager to create and run microVMs [5]. The choice for Firecracker stems from the fact that it is used in production by AWS Lambda [4, 5], thus increasing realism of the benchmark, as it is the largest FaaS provider [1]. Moreover, AWS Firecracker is developed as open-source software, meaning that it is freely available, whereas other microVM managers are not, or not as lightweight.

Figure 3: The step-by-step workflow of the benchmark for the end-user.

The overall design of the benchmark is straightforward: it is a collection of either Bash or Python scripts that perform specific functions, with the goal of measuring baselines, running workloads, and processing results. The configuration of the benchmark is handled using files. For instance, different types of test programs can be added by editing a single file and adding the sources of a test program to the *src* directory. The user only has to manually generate a CSV[2]-file with one or more allowed arguments for each test program, which will be re-used in the automated generation of workloads. In short, the benchmark runs a *workload*, which is a description of programs that are to be executed and their arguments. These programs described by the workload are called *test programs*. Figure 3 shows the workflow of the benchmark.

As stated above, the benchmark consists of a collection scripts in either Bash or Python that perform specific tasks, such as calculating baseline values, generating workloads, running the benchmark with a specified workload and process the results. These tasks will be elaborated on in this section in their respective order. Section 4.1 will explain the mechanism for generating workloads. Next, in Section 4.2 we go more in detail on how baseline values are measured. After this, in Section 4.3 we elaborate on the mechanics of running the benchmark. Subsequently, in Section 4.5 we will explain the machine monitor especially designed for this benchmark in Section 4.4. Lastly, we go into more detail on how the results are processed by the benchmark.

4.1 Generating Workloads

In this section, we explain the mechanism for generating workloads. In short, the workload generator is a Python script that generates a Poisson arrival pattern by using the NumPy package [18] as this provides convenient functions, increasing readability of the code.

[1] According to: https://www.oreilly.com/radar/oreilly-serverless-survey-2019-concerns-what-works-and-what-to-expect/

[2] Comma-separated value

Name	Description
primenumber	CPU-bound. Calculates the number of prime numbers in the range 0 to N, with N the program argument.
dd-workload	IO-bound. Generates a random file of S MB in size and subsequently deletes it. The file is generated using a block size of 1 MB. S is the argument for the program.
stream	Memory-bound. Based on a modified version of *stream*[16], runs the *stream* kernel X times, with X being the argument.

Table 1: A description of the test programs currently used by *fc-microbenchmark*

In the `fc-microbenchmark`, workloads are comprised of test programs. Currently, these test programs are simple, and are either CPU-bound, memory-bound, or IO-bound. Table 1 lists and briefly describes the test programs we use in this work. Test programs expect one command-line argument, which influences the run-time of the program. This means that the workload generator must be supplied with valid command-line arguments for each test program.

When running the benchmark, a workload describes the order in which test programs arrive. This can be either in a serial manner, or by using a Poisson arrival process. During workload generation, the user can choose to use the Poisson arrival process. In that case, the workload also describes the time to wait in between launching test programs.

The output of the workload generator is a CSV-file, that can be read by the benchmark script.

4.2 Determining Baseline Values

An important aspect of the benchmark is determining baseline values. These baseline values are used in predicting the run-times of a workload. Furthermore, the baseline values are used to determine the difference between the actual run-time of an individual program and the projected run-time of that program.

The baselines are measured by a specifically created script. This script should not be invoked directly, but rather by using the `start.sh` script, as this ensures that the environment is set-up correctly, see also Section 4.3 for more details. The user can specify the amount of times each baseline is measured. For this work, we chose 10 repetitions. The baselines are determined for two metrics, namely `tFC` and `tVM`. Figure 4 visualises what these metrics represent.

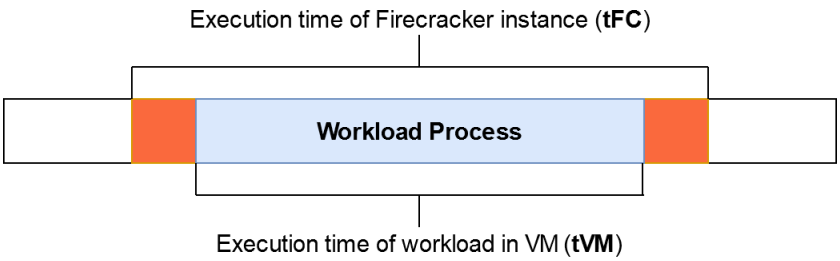


Figure 4: A visual representation of the `tFC` and `tVM` metrics.

The input for this process is a manually compiled list of arguments that are valid for the test programs. This list must be used for the creation of workloads, to ensure baseline values are present for the workload that will be executed.

4.3 Benchmarking

Figure 5 presents the overall mechanics of the benchmark step-by-step, excluding the measurement of the baselines, which is discussed in the previous section. In Step 1, we generate the workloads. After this, the user can run the benchmark. After this, in Step 2, we select a workload that will be executed. Then, the user invokes the `start.sh` script, and sets the mode to `benchmark`. This is the preferred method, as the `start.sh` script ensures that the system environment is always as equal as possible between each run of the benchmark. For instance, the script ensures that simultaneous multi-threading (SMT) is turned off. It also sets the CPU governor to `performance` and disable any turbo-boost technology, to avoid fluctuations in CPU clock and thus processing times, due to temperature increases [2]. Furthermore, the script increases the user limits of the calling shell [3] imposed by the operating system to the maximal values. Finally, the maximal value for the process IDs (PIDs) is increased, as the benchmark may serially spawn a large amount of processes, if the Poisson arrival process is not used.

[3] The user limits for the calling shell are configurable via the `ulimit` command, which is a shell built-in command. As the benchmark spawns many Firecracker instances, that all require multiple file descriptors, the regular user limits are too restrictive.

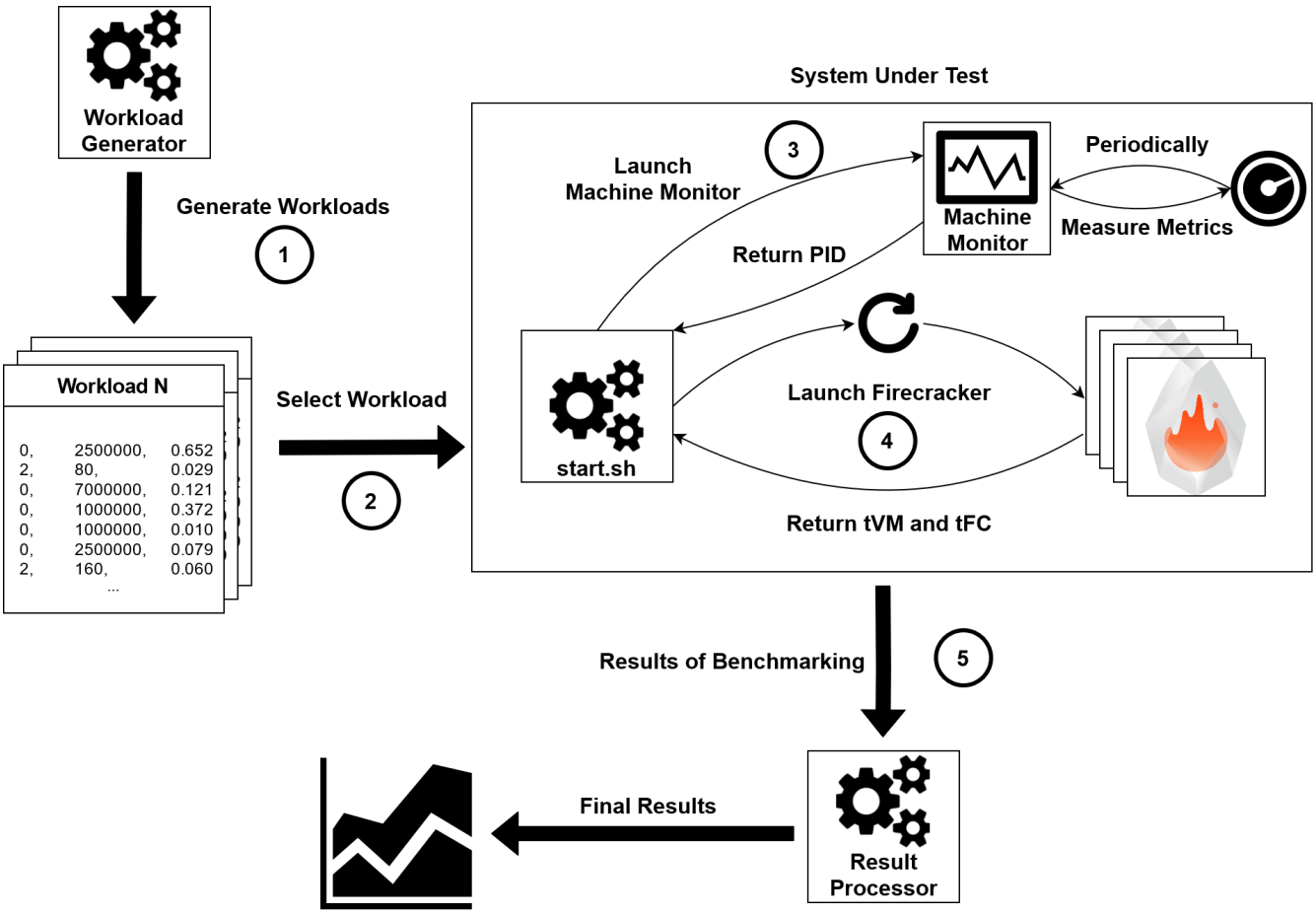


Figure 5: A detailed view on the working of the `fc-microbenchmark`.

After the system environment is set, Step 3 is executed; starting the machine monitor. Then, the workload file is read by the script and the benchmark starts. For each task in the workload file, the benchmark starts a Firecracker instance as seen in Step 4, with the task and arguments specified by the workload file. These parameters are communicated to the Firecracker instance by a subshell [4] started in the background, which waits for the Firecracker instance to start. During execution of the Firecracker instance and thus the task, the total time the Firecracker instance runs, as well as the total time the task runs inside the Firecracker instance is measured, for each task in the workload, as illustrated by Step 4. In addition to the run-times, the start time of each task is also recorded, as results are not written to the results file in the same order as they finish.

When all tasks in the workload have started, the script waits for all the tasks to finish. After this, the machine monitor is stopped and the results are stored. This leads to the final step, which is processing the results, in Step 5. We explain this step in more detail in Section 4.5.

Metric Name	Summary
CPU User time	Time spent by processes in user mode.
CPU System time	Time spent by processes in kernel mode.
CPU Idle time	Time spent doing nothing.
CPU Interrupt time	Time spent processing hardware and software interrupts and time spent waiting for I/O to finish.
CPU utilization	Percentage of CPU usage.
System Load	System load[5], normalised over one minute

Table 2: All metrics the Machine Monitor captures and the summary of these metrics.

4.4 Machine Monitor

In order to monitor resource usage on all machines that run the benchmark, we designed a straightforward machine monitor. The monitor is implemented in Python, using the `psutil` [20] package. It captures the system metrics listed in Table 2. After these metrics are captured, the monitor sleeps for a set period of time, one second by default. When the system is heavily loaded, this sleep time may vary more and may not be constant.

The capture of metrics continues indefinitely, as illustrated in Figure 5, or when the process receives a `SIGINT`. During normal execution, the calling script (`start.sh`) monitors the execution of the benchmark and when this is finished, it sends the `SIGINT` signal to the machine monitor.

4.5 Result Processing

After running the benchmark, step 5 in Figure 5, the results can be processed at automatically. The processing is done using `DataFrames`, provided by the `pandas` [22] Python package. The processing script expects the results to uphold a directory structure. In this directory structure, the root directory name contains both the scheduling algorithm and the machine abbreviation, or identifier, e.g. `RRc5n`. In this root directory, runs that are associated with a single baseline are grouped in a single sub-directory.

The first step in the results processing is collecting all measured baselines in the sub-directories of the directory provided to the processing script. From all baseline measurements found, the average baselines are calculated. Subsequently, the average baselines are used to create a prediction of the workloads on which the benchmark was run. The calculation of the predictions is straightforward: for every task in the workload, the expected execution-time is found and added to the sum of execution-times. This prediction represents the ideal execution of the workload, assuming unlimited resources and 'ideal' parallel hardware.

The next step in the process is processing the results. After a workload is executed, a CSV file is produced that contains the results per individual job in the workload. These results are not sorted,

[4] A Bash subshell, instantiated by an `&` at the end of the command.

and contain the type of job, its argument, the start-time of the job, and the run-times of both the microVM and the job inside the microVM. We sort these results on the start time, such that they appear in chronological starting order. After this, we calculate the end time of each job. Finally, we calculate the difference between the baseline run-times and the measured run-times, the `d tVM` and `d tFC`.

When the predictions and measurements are processed, we generate the concurrency graph. This graphs serves as a tool to determine whether a workload overloads the system. The graph is created by counting the amount of jobs that were executing over in a 10 second period. This is done for both the prediction and the measurement, Figure 7, in Section 5.4.1, shows a concurrency graph produced by the benchmark.

The final step of processing is done by a separate system, as this step is only taken when the system is not overloaded, whereas generating the concurrency graph, and processing the measurements and predictions is always done. This step needs more manual input. The workloads that do not overload the system must be selected by hand, and listed as input for a Jupyter Notebook. This notebook creates the graphs that show the average differences between baseline and measured run-times, as well as the graphs for resource utilisation, from the data generated by the Machine Monitor of Section 4.4. Finally, the Jupyter Notebook provides graphs that give more insight into the fairness of each scheduler the benchmark was executed on. Section 5.4 shows graphs generated by this step.

5 Experiments

In this section, we will detail the experimental setup in Section 5.1. After this, in Section 5.2 we elaborate on the specifications of the workloads that we used to generate the results. Then, we specify our methodology in Section 5.3. Finally, we discuss the results in Section 5.4.

5.1 Experimental Set-Up

We evaluated the performance of the scheduler by running the benchmark on a multitude of machines. Firstly, we used the `m510` machine of the CloudLab [9] cluster. This machine is equipped with an eight-core Intel Xeon D1548 CPU, clocked at 2.0GHz. Furthermore, the machine has 64GB of ECC DDR4 memory, clocked at 2133MHz. The machine runs Linux kernel version `4.15.0-101-generic`.

Apart from the machines of the CloudLab cluster, we also used the `apollo` machine, located at the TU Delft. This is a large-core machine, equipped with two AMD EPYC 7542 32-core processors and 256GB of memory. This machine runs Linux kernel version `5.4.0.53-generic`.

Due to issues with the `apollo` machine, we also utilised the bare metal `c5n.metal` instances of AWS. The `c5n.metal` belongs to the compute optimised category of instances AWS offers, but has more memory than the other types of this category. The processor in this machine is an Intel Xeon Platinum 8000 series CPU with 72 vCPUs running at 3.0GHz, and it has 192GB of memory. The Linux kernel version on this machine is custom built: `5.4.0-1029-aws`.

The benchmark, which is explained in Section 4, is backed by AWS Firecracker v0.21.1. The Machine monitor uses Python 3.7.6, and the `psutil` [20] package is version 5.7.3.

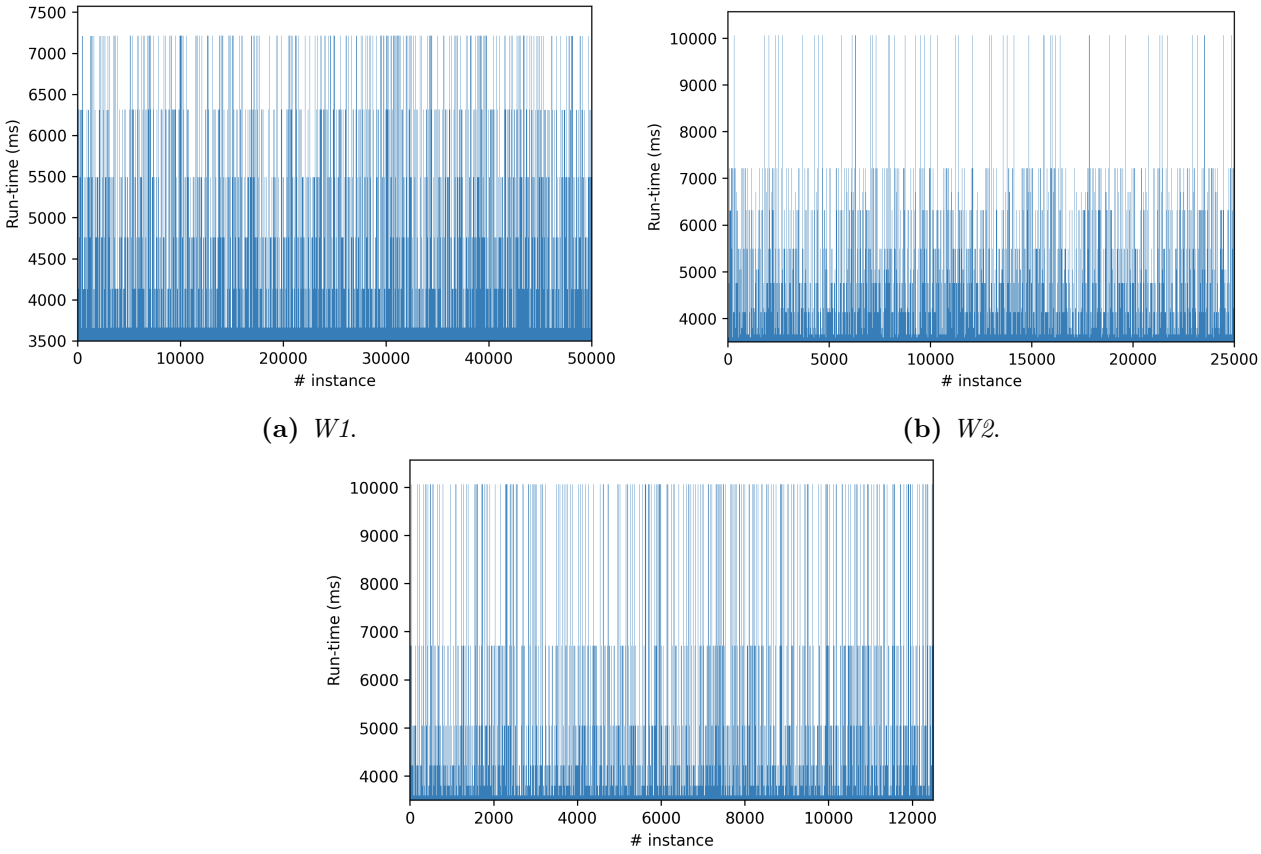
5.2 Workload Specifications

Due to the difference in specifications between the CloudLab machines and the `c5n.metal` machines, we devised different workloads for each machine. The primary parameter that differs between these workloads is the total amount of tasks executed within the time specified for the Poisson process. In all workloads, we aimed to let the workload execute for approximately one hour. For the larger workloads, the system overloads quickly due to the IO-bound programs. Furthermore, the IO implementation of Firecracker does not use flush-to-disk, which results in reduced performance [3]. Therefore, we exclude the IO-bound test programs altogether from the workload.

Workload	Amount of tasks	Run-time (h)	Portion CPU	Portion MEM
W1	50,000	1.0	1.0	0.0
W2	25,000	1.0	0.75	0.25
W3	12,500	1.0	0.0	1.0

Table 3: An overview of the workloads we use in the benchmark and we discuss in this section. We varied with the size of the workload, as well as the mix of test programs in each workload.

Furthermore, we also vary the mix of test programs for each workload size. This isolates each test program and shows the handling of pure CPU-bound, IO-bound and memory-bound tasks by the scheduler. Table 3 shows all configurations that we discuss in this chapter. In Figure 6 we show the distribution of the expected run-times of the jobs in the workload. If the workload overloaded the system too much[6], the benchmark yields no informative results. As such, many configurations were tested, before we settled for the current list. For an extensive list of workloads, including ones that overloaded the system, we refer to Table 6 in the Appendix.



(a) *W1.* **(b)** *W2.*

(c) *W3.*

Figure 6: The distribution of jobs throughout each workload. The horizontal axis indicates the number of the instance in the workload. The vertical axis shows the run-time of the job in milliseconds. In general, the distribution of jobs of different run-time is even throughout the workload. Including memory-bound jobs increases the run-time of the longest running job from around $7,250$, to around $10,000$ milliseconds.

[6] In general, we marked a configuration as too intensive if it exceeded a total execution time of 1,5 hours.

5.3 Methodology

In this section, we explain the measured metrics, and the derived metrics we need to answer the research questions posited in Section 1. In order to answer these questions, we measure the performance of CFS and assess the quantitative evidence on the fairness of CFS. We achieve this by comparing CFS, using the benchmark of Section 4, with the subset of scheduling classes of Section 2.5, namely BATCH, FIFO and RR (see Section 2.5 for precise definitions).

- Firstly, on each machine of Section 5.1, we measure the **baseline execution times** of the individual programs that compose the workloads, for each scheduler. As we are interested in the run-time of the whole Firecracker instance, and the run-time of the program inside the Firecracker microVM, we measure baselines for these metrics. These metrics are referred to as `tFC` and `tVM` respectively. The baseline measurements are performed serially, without any parallelism. This means that a baseline value corresponds to the execution of a job under *ideal* circumstances. Each baseline is measured 10 times, and the final baseline value is the average of these repetitions.

- After the baseline measurements are finished, **we predict the run-time and the concurrency of the workload**, for every scheduler, by using the baseline run-times. Concretely, this is done by adding the predicted value of `tVM` to the start-time of each respective program in the workload. The predicted results have the same format as the actual processed results, see Section 4.5 for a concise description of this process. The prediction and predicted concurrency correspond to the *ideal* execution of the workload, with infinite resources and unlimited parallelism. After this, we execute the workloads specified in Table 3 for each scheduler.

- After the results are processed (see Section 4.5), **we visualise the measurement by using a concurrency graph**. The concurrency graph gives an indication of the distribution of individual tasks over time. We use the concurrency graph as a tool to check whether a workload oversubscribes the system too much, or not at all. In addition, the graph shows how the workload performs, compared to the prediction. If a workload either oversubscribes the system too much, or too little, we do not continue analysis on that workload. For examples of concurrency graphs that show the system was too oversubscribed, or too little, we refer to Appendix A.5. Another visualisation of the measured metrics we use, is a graph that shows the evolution of run-times of an individual job over time. To make this graph more informative, we also create a histogram that bins these run-times of an individual job. The amount of bins is determined by using the Freedman-Diaconis rule [11]. The histogram also supports the calculated coefficient of variability, which we discuss later in this section.

- Apart from the measured metrics, we also calculate the overall run-time of the workload, the difference between expected run-time and measured run-time for each job, the time-deltas, and coefficients of variability. We calculate the overall run-time of a workload by adding the run-time of the job to the start-time of that job, for each job. After this, the run-time of the workload is determined by picking the largest end-time. The time-deltas are calculated by subtracting the predicted `tFC` from the measured `tFC`, which results in the `d tFC`. The `d tVM` is calculated analogously. These values provide information regarding the performance of the scheduler. Ideally, these values should close to 0, which means the scheduler is able to emulate

a fair CPU, with infinite resources and parallelism. For comparison amongst schedulers, we calculate the average of the two deltas, for each scheduler.

- Finally, we calculate the coefficient of variation, by dividing the standard deviation of the time-deltas over the mean of the time-deltas. The coefficient of variation shows the ratio between the standard deviation and the mean. As we want to make a fair comparison with respect to the coefficient of variation between the schedulers, we limited the calculation of the coefficient of variation to a pre-selected program and program argument in the workload. Under ideal circumstances, repeatedly running the same program should result in a coefficient of variance close to zero. If this is not the case, we attribute this to contention amongst programs and scheduling pressure. Furthermore, including the other arguments of the same program results in a larger variance than it should be, as for all programs, a larger argument is directly correlated with a longer run-time, thus adding all arguments of the program, or all programs, in the calculation makes the calculation less informative. Table 4 shows an overview of all metrics.

In this work, *a lower coefficient of variation is correlated with fairness*, as a lower standard deviation of the time-deltas means that all jobs get executed in a more similar time-span. In a real-world scenario, this means that comparable customers jobs are executed in the same amount of time, and thus are billed equally.

Metric	**Summary**
Run-time	The total run-time the workload of the workload, defined by the last finishing program in the workload.
`tFC`	Individual run-time of a Firecracker instance in the workload.
`tVM`	Individual run-time of a program executed in the Firecracker microVM.
`d tFC`	Difference between the predicted `tFC` and the measured `tFC`.
`d tVM`	Difference between the predicted `tVM` and the measured `tVM`.
Coefficient of variation for `tFC` & `tVM`	Ratio between the standard deviation and the mean of `tFC` and `tVM`. This value is a numeric value for fairness.

Table 4: An overview of the metrics we measure.

5.4 Results

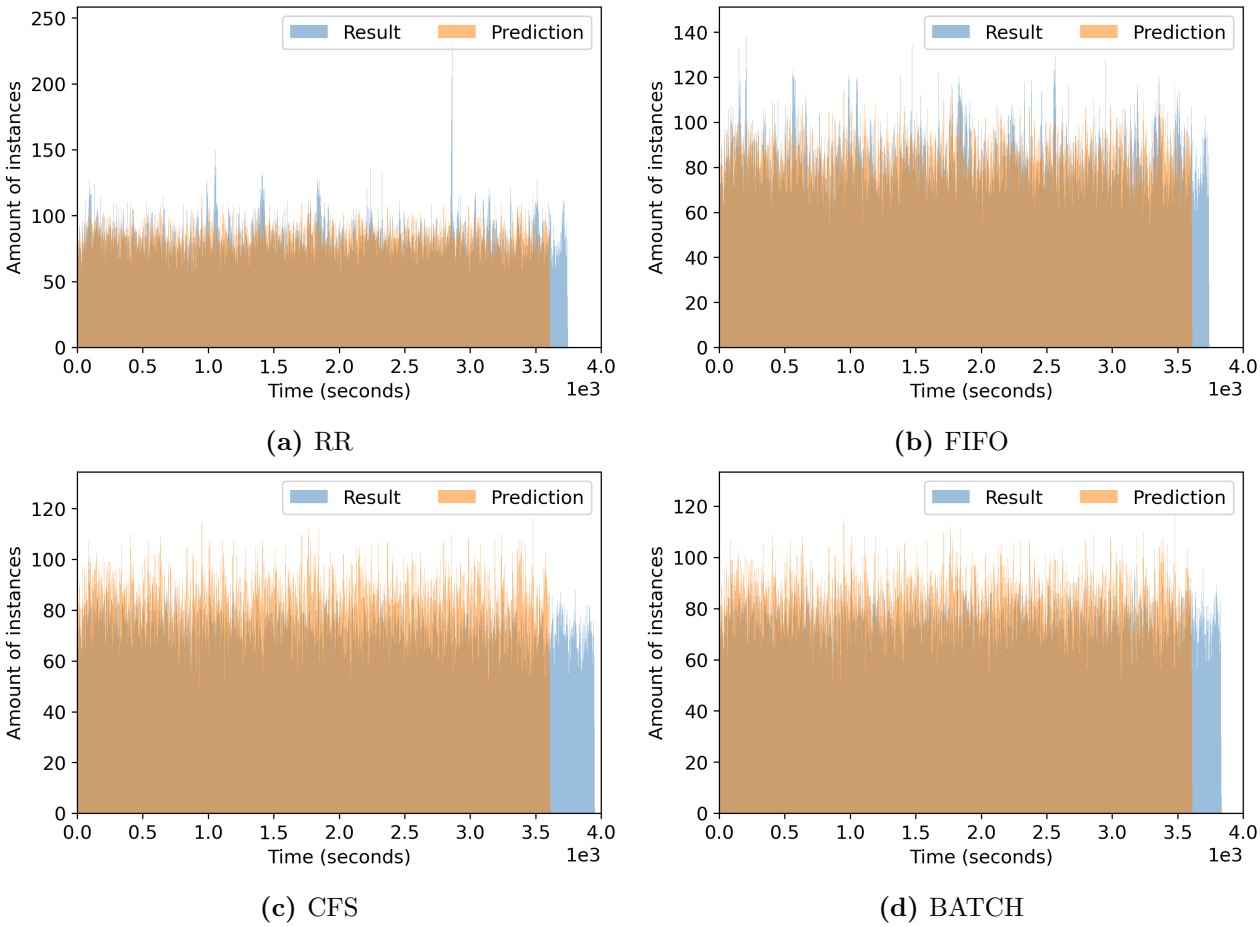
In this section, we explain the results of our benchmark that were retrieved by executing the workloads described in Section 5.2 on the `c5n.metal` machine. We chose this machine as it most closely resembles the machine used in the original paper on AWS Firecracker [3], but at reduced cost. For the results of all machines, we refer to Appendix A.

In Section 5.4.1, **Total Run-times and Deltas**, we first show the concurrency graphs, which is a tool to determine whether the system is overloaded. After this, we show the effect the scheduler has on the total execution time of a workload. Finally, we show the time-deltas, which indicate how much the scheduler achieves *ideal* execution. *For the overall run-time, CFS and BATCH are outperformed by RR and FIFO.* However, when inspecting the time-deltas, CFS and BATCH perform best, especially for the delta of the `tFC` metric.

After this, in Section 5.4.2, **Resource Utilisation of different Schedulers**, we delve deeper into the results by analysing the effect the schedulers have on the resource utilisation. For this analysis, the outcome is the same as for the total run-time: *RR and FIFO outperform both CFS and BATCH.*

Finally, in Section 5.4.3, **Numerical Analysis**, we perform a numerical analysis on a pre-selected, repeated job inside the workload, which yields the coefficient of variance for this job. The coefficient of variance is used as quantitative evidence for the fairness of the schedulers. We find that *the coefficient of variance is dependent on the type of workload.* With regard to the schedulers, this result leads to the observation that *CFS and BATCH perform worse on workloads that include memory-bound jobs.*

5.4.1 Total Run-times and Deltas

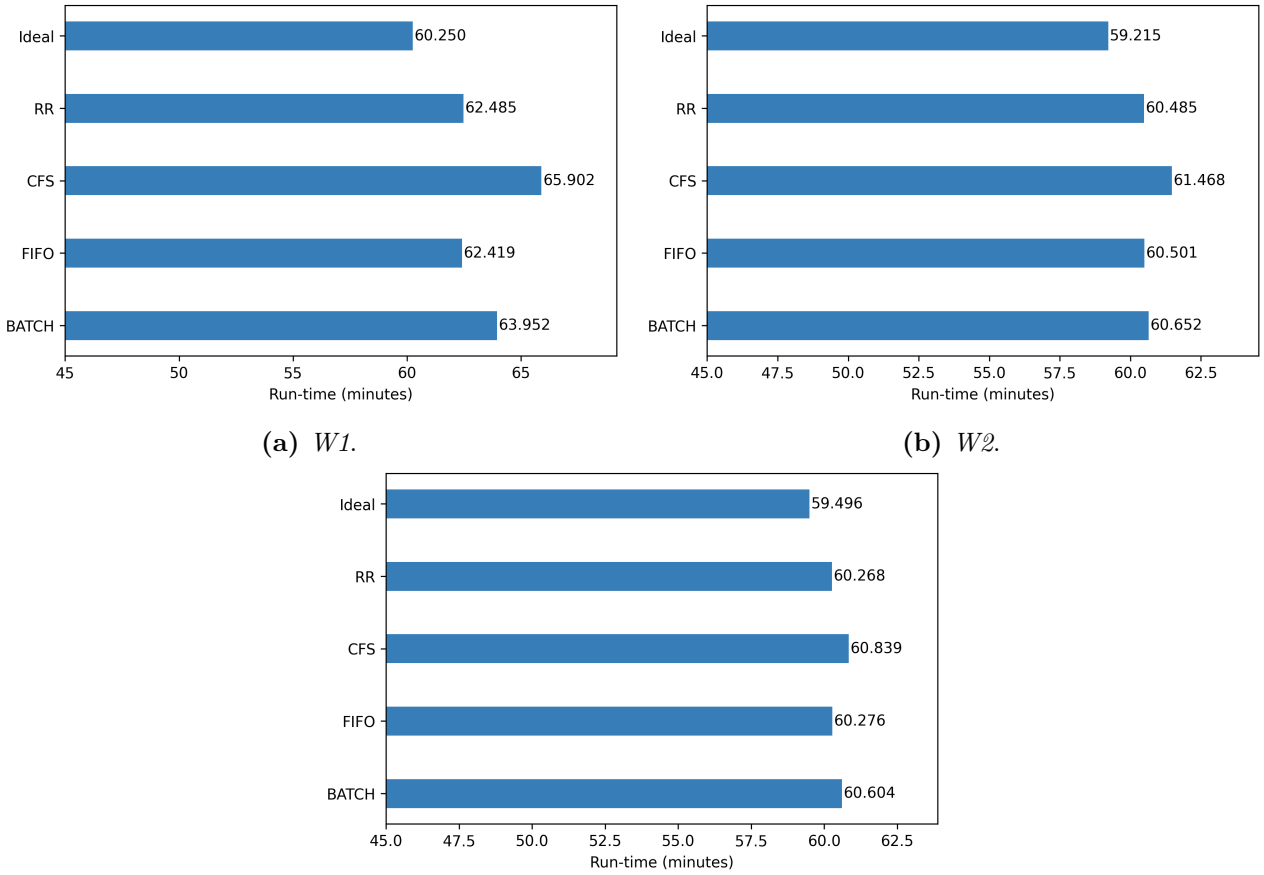


(a) RR **(b)** FIFO

(c) CFS **(d)** BATCH

Figure 7: The amount of concurrent instances per unit of time of the four different schedulers for workload *W1*. The selected unit of time to bin concurrent events in, is 1 second. For CFS, the prediction expects more concurrency in each bin, than the results shows. For BATCH, FIFO and RR, the predicted concurrency and the actual concurrency differ less.

To start the evaluation, we inspect the concurrency graphs of each workload, for each scheduler. Figure 7 shows the graphs for all schedulers for workload *W1* of Table 3. The workload does not overload the system, as the difference between the predicted concurrency and the actual concurrency is not large. For the graphs of the other two workloads, and the workloads on the other machines, we refer to Figure 16 and Figure 17 in the Appendix. The discrepancy between the predicted and measured concurrency in Figure 7c is the largest for CFS, compared to BATCH, FIFO and RR. The simpler schedulers, RR and FIFO, outperform the more complex schedulers, BATCH and CFS. Also, the measured run-time of CFS exceeds the predicted run-time with the largest margin, approximately 350 seconds, opposed to approximately 150 seconds for RR.



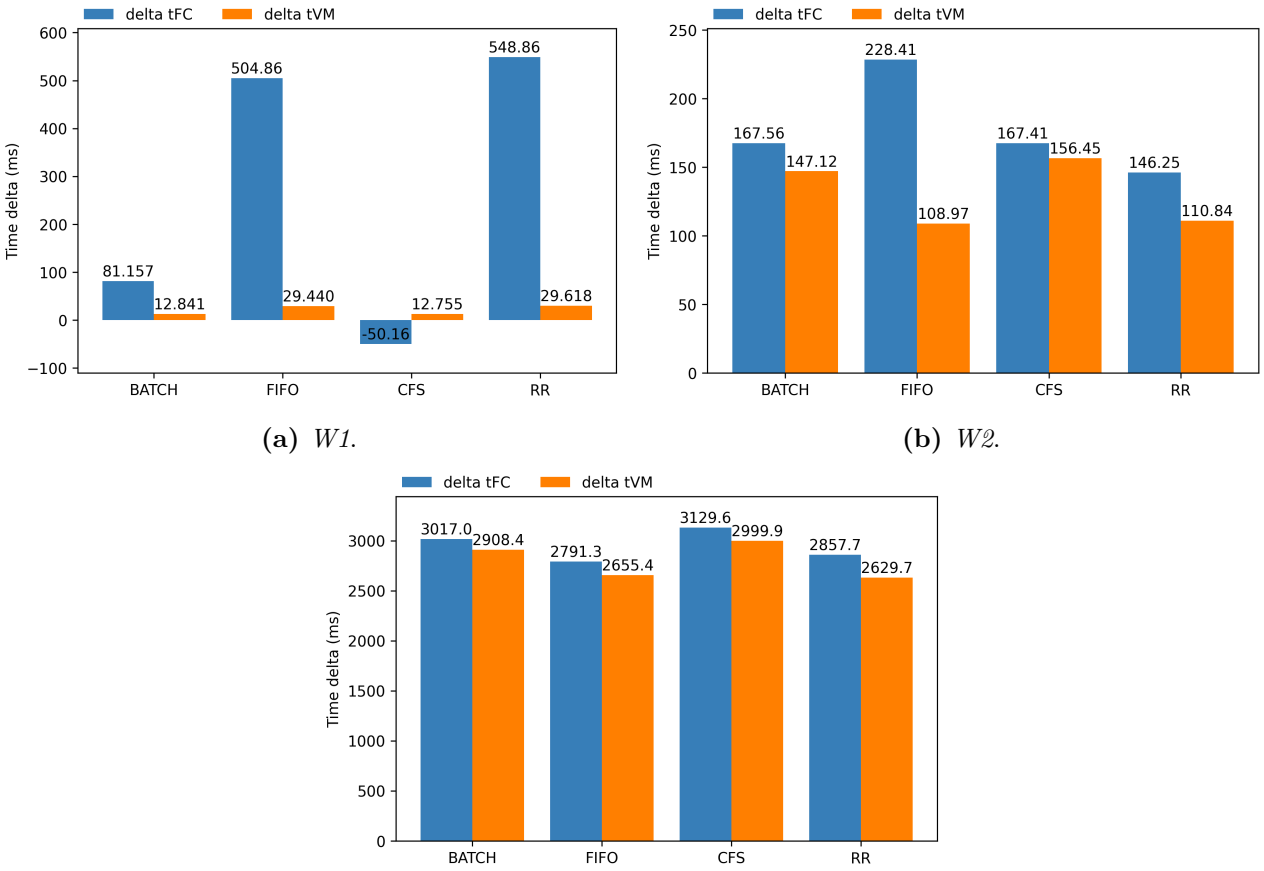
(a) *W1*.

(b) *W2*.

(c) *W3*.

Figure 8: This figure shows the run-times of the workloads on the `c5n.metal` machine, in minutes. On the vertical axis, the label describes which run-time bar corresponds to which scheduler. The `ideal` is the time calculated by the prediction, as described in Section 5.3.

After inspection of the concurrency graphs, we compare the total run-time of the workloads to an *ideal*. Figure 8 shows the total run-time for each scheduler. We omit error bars, as the performance variation of the individual tasks in the workload is reduced, due to the minimal run-time of 60 minutes. In this figure, the `ideal` is the average of the predicted run-times for each scheduler. It is clear that no scheduler performs as well as the ideal. For this metric, CFS is outperformed by all other schedulers, although the difference for workloads *W1* and *W2* is less than one minute. However, for workload W3, the difference is more than 2,5 minutes between CFS and the best performing scheduler. In general, the simpler schedulers, RR and FIFO, perform the best out of the four tested schedulers.



(a) *W1.*

(b) *W2.*

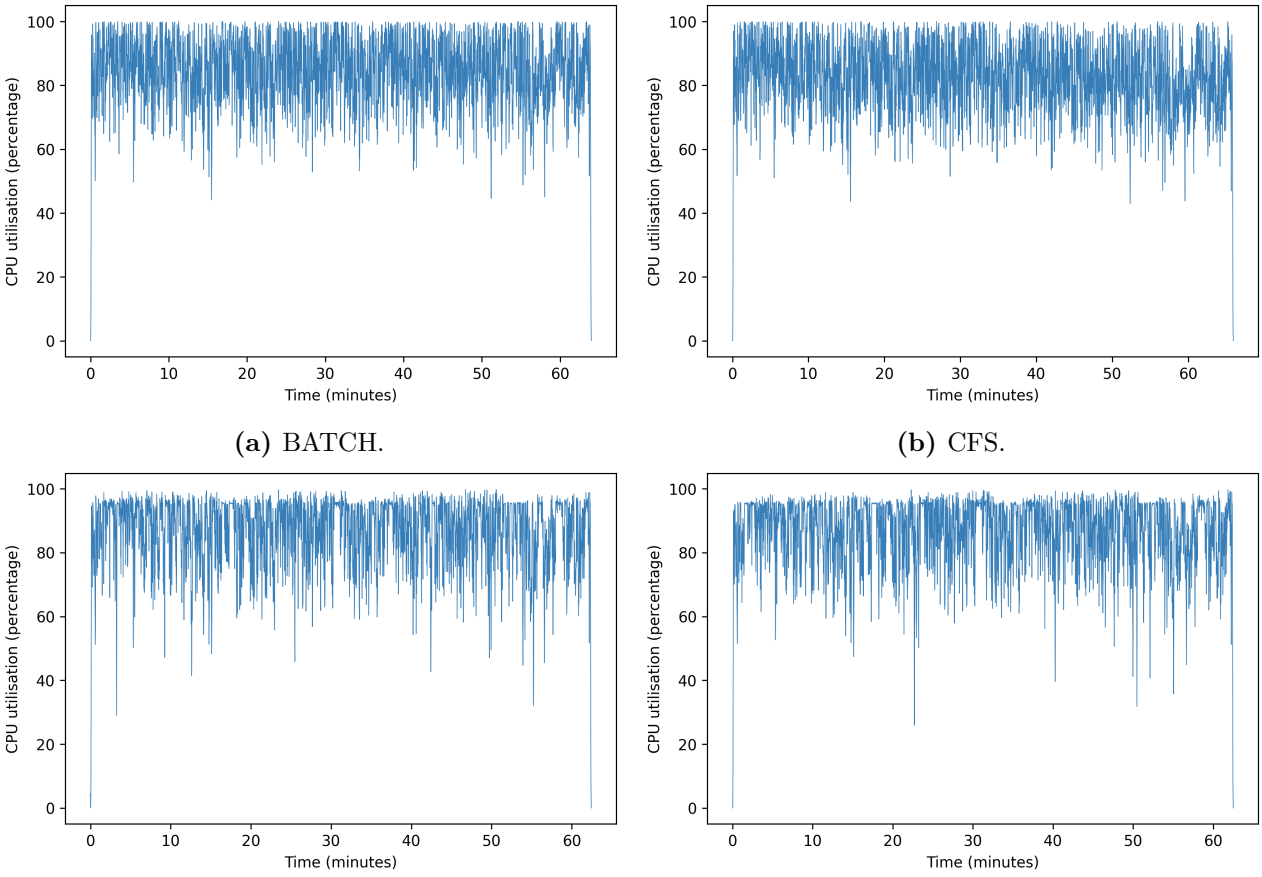
(c) *W3.*

Figure 9: Results of the average of the calculated time-deltas for each workload and each scheduler.

The time-deltas show the average deviation of jobs in the workload, compared with the baseline run-times of each job. The baseline run-time is retrieved by measuring the run-time of individual jobs, without any concurrency, see also Section 4.2. In the benchmark, we measured the time-deltas of both the Firecracker instance as a whole, and the job executed inside the Firecracker microVM. Figure 9 shows the aggregated time-deltas for each scheduler and workload. Again, we do not show error bars, as the time-deltas are the average of all tasks in the workload. These tasks differ in nature, as the run-times of the tasks differ significantly by design. Therefore, the resulting standard deviations and thus the error bars will be non-informative. This effect is illustrated in Figure 15 in Appendix A. If the time-deltas are closer to zero, this indicates that job run-time approaches the run-time we expect it to have on an ideal CPU. As can be seen in Figure 9a, the deltas of CFS are very close to zero. In fact, the time-delta of the Firecracker instance is below zero, indicating it executes faster than we predicted. This result also suggests that CFS does perform as well as an *ideal* CPU on this workload, on a per-task level. On this large CPU-only workload, the more complex schedulers, CFS and BATCH, outperform the simpler schedulers, RR and FIFO, by a significant margin. When we take the run-time into consideration, there is a discrepancy between the run-time we expect based on the time-deltas in Figure 9a and the run-times of Figure 8a. We hypothesise this is due to the overhead of the more complex schedulers.

In contrast to the time-deltas of the CPU-only workload, the time-deltas of the memory-only and mixed workloads paint a different picture. For instance, Figure 9c shows that the time-deltas increase at least sixfold for the memory-only workload. Interestingly, CFS and BATCH perform worse than RR and FIFO. If we mix the memory-bound jobs with CPU-bound jobs, this performance difference amongst the schedulers is mitigated slightly, see Figure 9b. Now, FIFO shows very large differences between the time-deltas of the Firecracker instance and the job, but RR still outperforms the other two schedulers. The difference between performance of CFS and BATCH on the workloads that include memory-bound jobs and the CPU-only workload are interesting for scheduling cloud jobs, as it may be beneficial to change the scheduler if the type of workload is known beforehand.

5.4.2 Resource Utilisation of different Schedulers



(a) BATCH. **(b)** CFS.

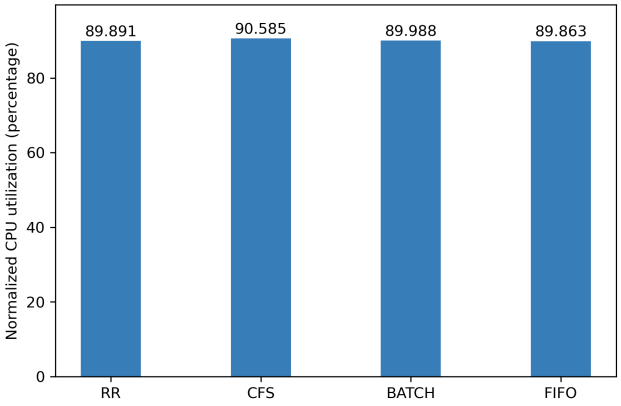
(c) FIFO. **(d)** RR.

Figure 10: The CPU utilisation over time for all schedulers when executing workload *W1*. The vertical axis shows CPU utilisation as a percentage.

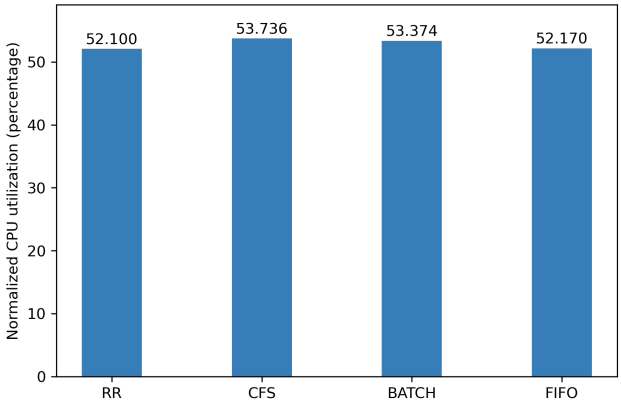
In order to better understand the discrepancy in run-times and measured concurrency amongst the different schedulers, we also inspect the results generated by the machine monitor (see Section 4.4). Figure 10 shows the CPU utilisation over time for all schedulers, for workload *W1*. As these graphs

have many data-points, we calculated the Area Under the Curve (AUC) of these graphs, to create a better method for comparison.

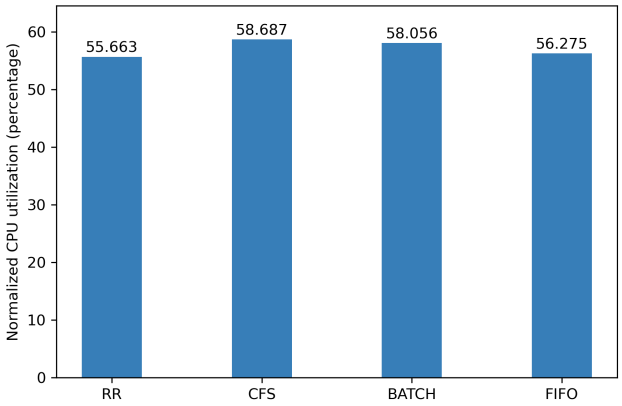
The AUC shows how much CPU-time the benchmark as a whole needed. In order to compare the AUCs of the different schedulers, we normalised these values by dividing the AUC with the *maximal possible usage*, which is 100% CPU usage over one hour. Figure 11 shows the final result. For all workloads, the AUC is a smaller value than the *maximal usage*. This is to be expected, as 100% CPU usage indicates an overloaded system, which is undesirable. Although the differences are small, for all AUCs, the total CPU utilisation for CFS and BATCH is higher than for FIFO and RR. Again, the less complex schedulers outperform the more complex schedulers. For example, in Figure 11a the AUC for all schedulers is close to 90% of the *maximal usage*, but CFS has the highest value, followed by BATCH. We hypothesise that the higher CPU utilisation of CFS and BATCH stems from the fact that they use specific algorithms for inserting and removing tasks from their job queues, with a complexity of $O(\log n)$. FIFO and BATCH on the other hand, just insert and remove from their queues with a complexity of $O(1)$. When accumulating all these insertions and deletions over time, the difference in complexity may be an explanation for the difference in scheduler overhead.



(a) *W1.*



(b) *W2.*



(c) *W3.*

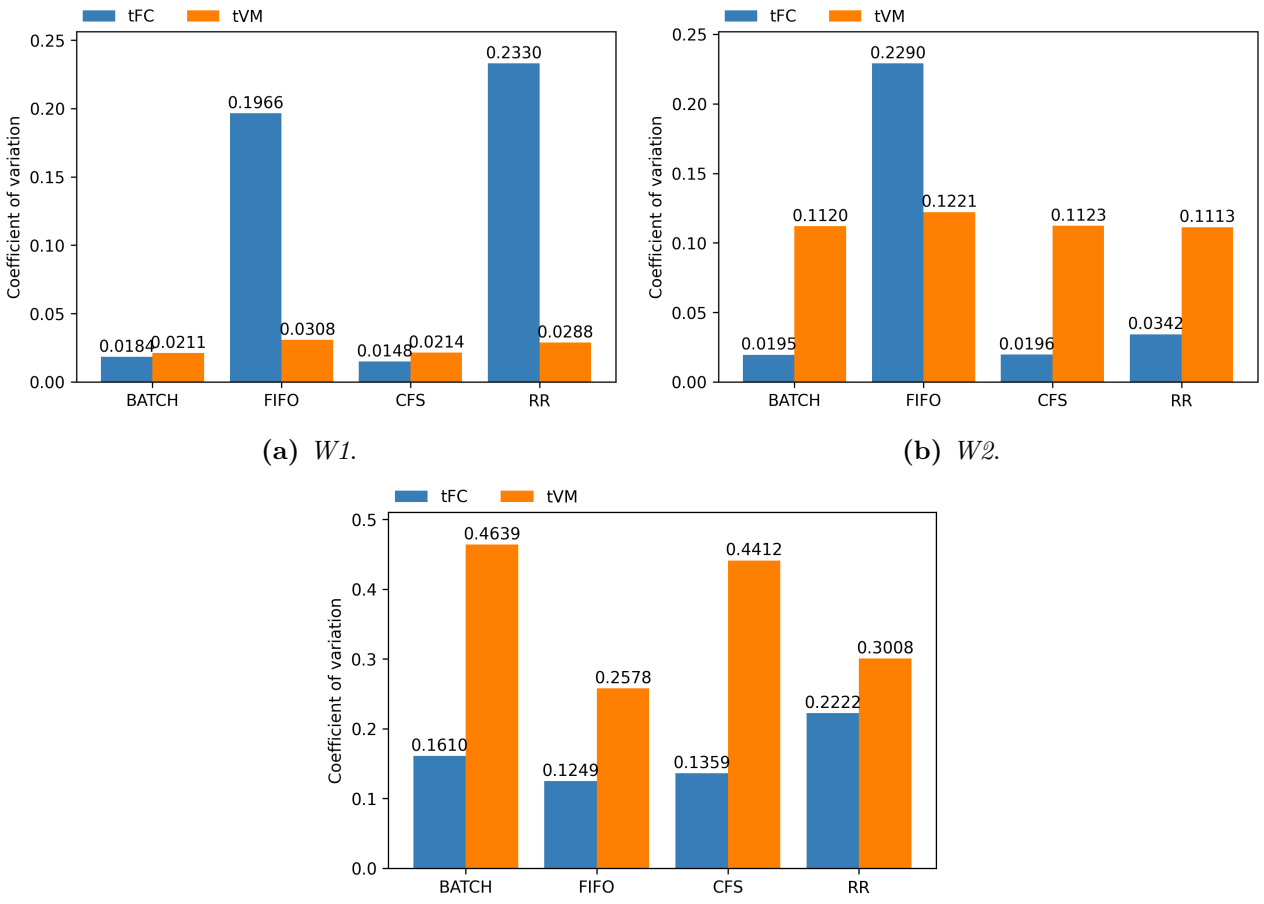
Figure 11: In this figure, we show the area under the curve (AUC) of the CPU utilisation of all four schedulers for the three workloads. The *maximal usage* refers to the value retrieved for a 100% CPU usage for one hour. All other AUC values are divided by this *maximal usage*, for comparison.

5.4.3 Numerical Analysis

Workload ID	Program ID	Program Argument
W1, W2	0	5,500,000
W3	2	20

Table 5: The selected programs for which we calculate the coefficient of variation.

From the results shown in the earlier sections, CFS does not perform as well as RR and FIFO. However, the time-deltas for the CPU-only workload have shown that, although CFS seems to incur overhead, for the individual execution times of the jobs, it performs closer to the *ideal* CPU it tries to model, than the other schedulers. To verify this, we calculated the coefficient of variation. The coefficient of variation provides insight into how the deviation of the run-times relates to the mean. The smaller the coefficient of variation, the more the run-times adhere to the run-times of a fair processor. If the coefficient of variation equals zero, there is no variance, thus all tasks are scheduled completely fair with respect to each other. In each workload, we selected a specific program, for which we calculated the coefficient of variance. Table 5 shows the selected programs.



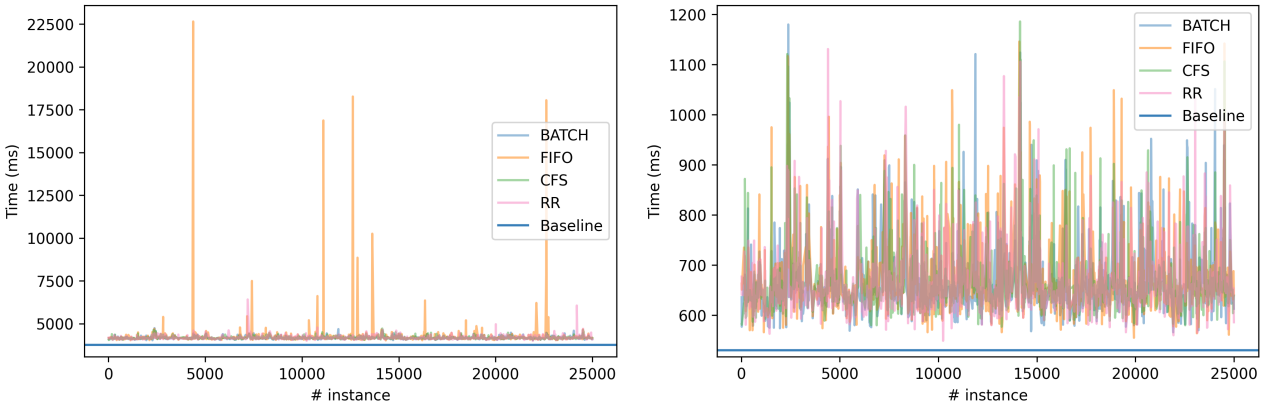
(a) *W1*.

(b) *W2*.

(c) *W3*.

Figure 12: The coefficient of variation for the three workloads we analyse in this section. The vertical axis shows the value for the coefficient of variation, which is unit-less. As we measure both the run-times of the Firecracker instance and the job, there are two values per scheduler for the coefficient of variation.

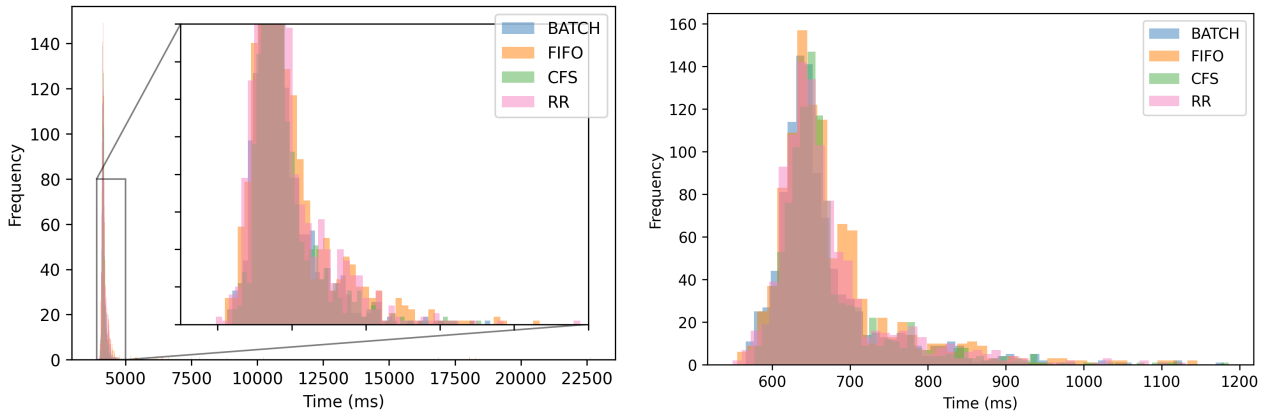
Figure 12 shows the variance and coefficient of variation per scheduler, per workload. For workload *W1*, see Figure 12a, CFS and BATCH show a lower value for the coefficient of variation, which indicates that the schedulers are fairer than FIFO and RR, both for `tFC` and `tVM`. However, if we add the memory-bound programs to the workloads, the fairness of the schedulers changes. Figure 12b shows that the coefficient of variation of `tFC` for all schedulers is approximately equal. The coefficient of variation for `tVM` is still most favourable for CFS and BATCH. Once the workload consists of only memory-bound tasks, the coefficient of variation shows a different picture of that of Figure 12a, as illustrated by Figure 12c. In this case, FIFO and RR achieve much better fairness with respect to `tVM` than CFS and BATCH, with FIFO being the most fair. With respect to `tFC`, FIFO is the winner, followed by CFS and BATCH. It is important to note that a lower `tVM` is desirable for the customer as it represents fairness amongst jobs of customers, whereas a lower `tFC` is desirable for the provider, as the microVM is the responsibility of the provider.



(a) `tFC`. **(b)** `tVM`.

Figure 13: The run-time of the selected program when executing at different indices in workload *W2*. The horizontal axis shows the index of the program in the workload. The vertical axis shows execution-time of this program in milliseconds.

For a better understanding of the coefficient of variation, we selected the same programs in each workload as in Figure 12, but plot the run-times of each individual program. The result is shown in Figure 13. We only show the results of workload *W2*. For the same plot, but for the other workloads, we refer to Figure 20 in the Appendix. In Figure 13a, we note that FIFO spikes regularly. A spike means that the individual task runs long. Examples of spikes in Figure 13a are visible before the 5,000$^{\text{th}}$ instance for FIFO, as well as multiple FIFO spikes between the 10,000$^{\text{th}}$ and 15,000$^{\text{th}}$ instance. On the other hand, Figure 13b does not exhibit this behaviour for one, isolated scheduler. We have no quantitative evidence to explain these spikes, but we hypothesise a possible explanation in Section 6.



(a) `tFC`. **(b)** `tVM`.

Figure 14: The histogram of the individual run-times of the selected program for workload *W2*. The bin size is determined by the Freedman-Diaconis rule [11]. The horizontal axis shows the run-time in milliseconds.

In addition to the individual run-times, we also created a histogram from this data, to further illustrate the effect of the coefficient of variance. Again, we only show the results for workload *W2*. For the results of the other workloads, we refer to Figure 21 in the Appendix. Figure 14 shows the histogram. The width of the histogram is directly correlated with the coefficient of variance. As seen in Figure 12b, the coefficient of variance for FIFO is highest. In Figure 14a, this is reflected by small spikes. These are not visible due to the main spike around the $4,000$ milliseconds value. The existence of these spikes is, however, verifiable by inspecting Figure 13a, where FIFO exhibits several large spikes in run-time. In general, the coefficient of variance for `tVM` for the different schedulers are close to each other, for workload *W2*, as confirmed by the histogram of Figure 14b.

6 Discussion and Further Research

As shown by the results, CFS does not perform as well as FIFO and RR for workloads that incorporate memory-bound tasks. However, when the workload consists of mostly CPU-bound tasks, it is the best choice. This provides another possibility for optimisation for the cloud provider. If type of workload is known beforehand, the provider can choose to change the scheduler, to maximise the throughput of tasks. We believe this should be subject of further research.

Although our data does not show a clear reason why CFS has a higher coefficient of variation for the tasks running inside the microVM when executing a memory-bound workload, we hypothesise this is due to the amount of preemption that occurs. When the scheduler preempts a memory-bound task, this leads to a loss of locality. The CPU cache needs to be filled with data for the next task in the scheduler's queue. Logically, less preemption means that locality is preserved, and thus less cache-misses occur, positively impacting total run-time. Furthermore, preemption incurs additional overhead by the scheduler itself. When many preemptions occur, this small overhead adds up, negatively impacting total run-time. As FIFO aims to not preempt, and RR preempts after a set interval, generally leading to less preemptions than CFS, we hypothesise that RR and FIFO preserve locality better, which is why these schedulers perform better on memory-bound workloads. Again, this is a pointer for further research.

Furthermore, the point raised in Section 5.4.3, on Figure 13 is interesting for further research. As stated in Section 2.5, FIFO and RR do not preempt, unless the time quantum expired, in case of RR, or when a program performs preempting system calls. Furthermore, we note that Firecracker does not perform these system calls and neither do the test programs. This may result in situations where a program is not performing any useful computation, or just waiting, but is not preempted by the scheduler. We hypothesise that this happens during the large spikes in Figure 13a for the FIFO scheduler.

7 Conclusion

In this work, we determined whether the Completely Fair Scheduler is ready for serverless workloads. To this end, we created the `fc-microbenchmark` as a method to run workloads and evaluate the performance of the execution of the workload, in a reproducible manner. In order to check the performance of the CFS scheduler, we compared the performance of CFS with other schedulers in the Linux kernel when executing the `fc-microbenchmark`.

From the results we gathered, we conclude that with respect to performance CFS is not ready. With respect to fairness, which CFS tries to provide, it performs well for CPU-bound workloads. However, for workloads that incorporate more memory-bound tasks, CFS is outperformed by both the Round-Robin and First-In-First-Out schedulers. As memory is an important factor in FaaS applications [21], we conclude that CFS is not ready for serverless workloads, when workloads incorporate memory-bound tasks, as in that case, CFS loses both in terms of performance and fairness to FIFO and, to a lesser extent, RR.

References

[1] *sched(7) Linux Programmer's Manual*, kernel 5.8.9 edition, 8 2019.

[2] B. Acun, P. Miller, and L. V. Kale. Variation among processors under turbo boost in hpc systems. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[3] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 419–434, 2020.

[4] Amazon. AWS Lambda. https://aws.amazon.com/lambda/. Retrieved on 11-12-2020.

[5] AWS. Firecracker. https://firecracker-microvm.github.io/. Retrieved on 5-8-2020.

[6] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[7] N. Boonstra. `fc-microbenchmark`. https://github.com/NJLBoonstra/fc-microbenchmark.

[8] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena. The battle of the schedulers: Freebsd ULE vs. Linux CFS. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*, pages 85–96, 2018.

[9] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[10] V. Eyk. Cloud Serverless is More : From PaaS to Present Cloud Computing. 2018.

[11] D. Freedman and P. Diaconis. On the histogram as a density estimator: L 2 theory, 1981.

[12] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Google. Google cloud functions. https://cloud.google.com/functions/. Retrieved on 11-12-2020.

[14] IBM. IBM Cloud Functions. https://www.ibm.com/cloud/functions/. Retrieved on 11-12-2020.

[15] N. Mahmoudi and H. Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, 2020.

[16] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/. Retrieved on 9-10-2020.

[17] Microsoft. Azure functions. https://azure.microsoft.com/en-us/services/functions/. Retrieved on 11-12-2020.

[18] T. Oliphant. Numpy. https://numpy.org/. Retrieved on 11-12-2020.

[19] pipenv Maintainer Team. `pipenv`. https://pypi.org/project/pipenv/. Retrieved on 5-8-2020.

[20] G. Rodola. `psutil`. https://pypi.org/project/psutil/. Retrieved on 11-12-2020.

[21] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[22] The pandas development team. pandas-dev/pandas: Pandas 1.2.2, Feb. 2021.

[23] L. Torvalds. Linux kernel, *Documentation/scheduler/sched-design-CFS.rst*. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

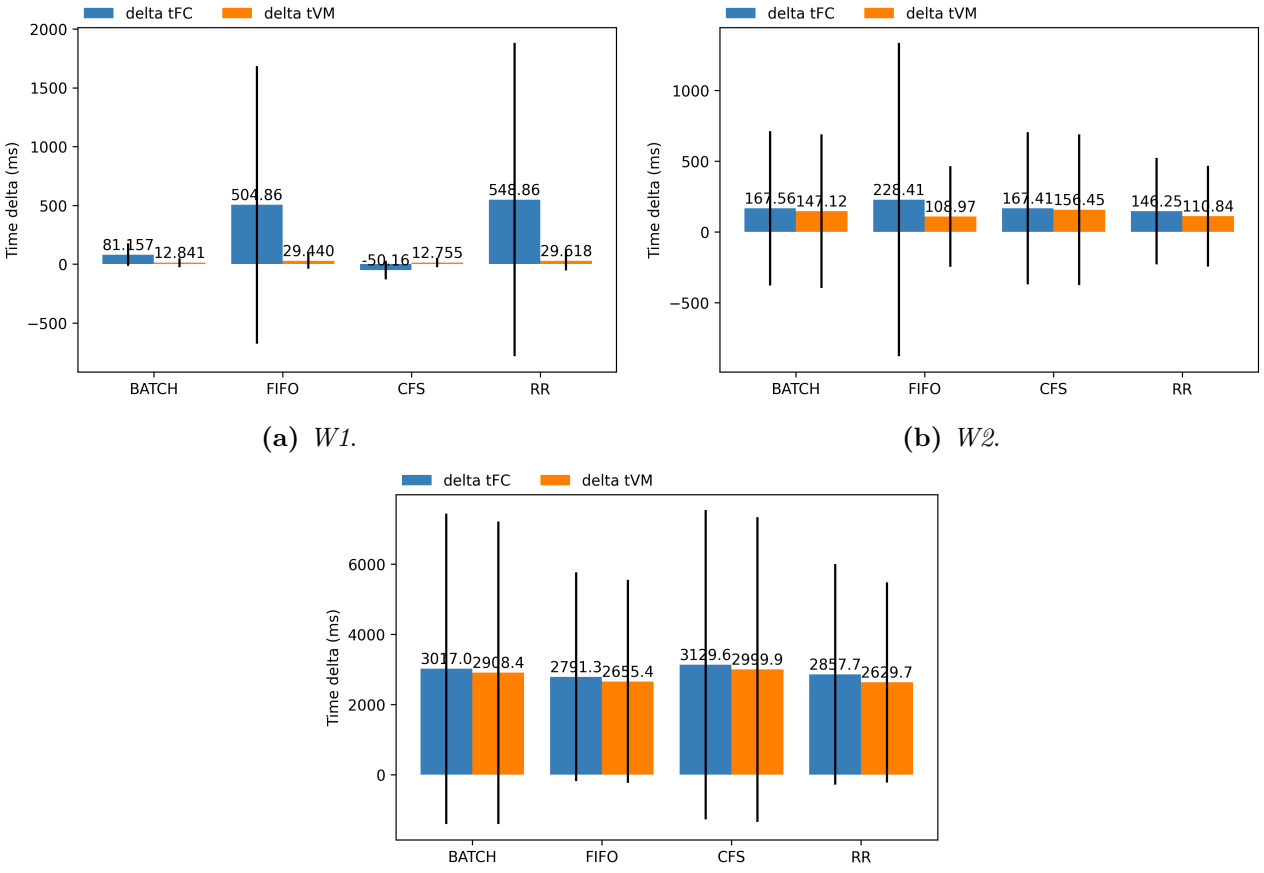
[24] E. Van Eyk, J. Scheuner, S. Eismann, C. L. Abad, and A. Iosup. Beyond microbenchmarks: The spec-rg vision for a comprehensive serverless benchmark. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 26–31, 2020.

[25] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

A Appendix

A.1 Total Run-times and Deltas

We show the time-delta plot, including the error bars, in Figure 15. This figure illustrates that the error bars are not informative, due to the different nature of the tasks in the workload.

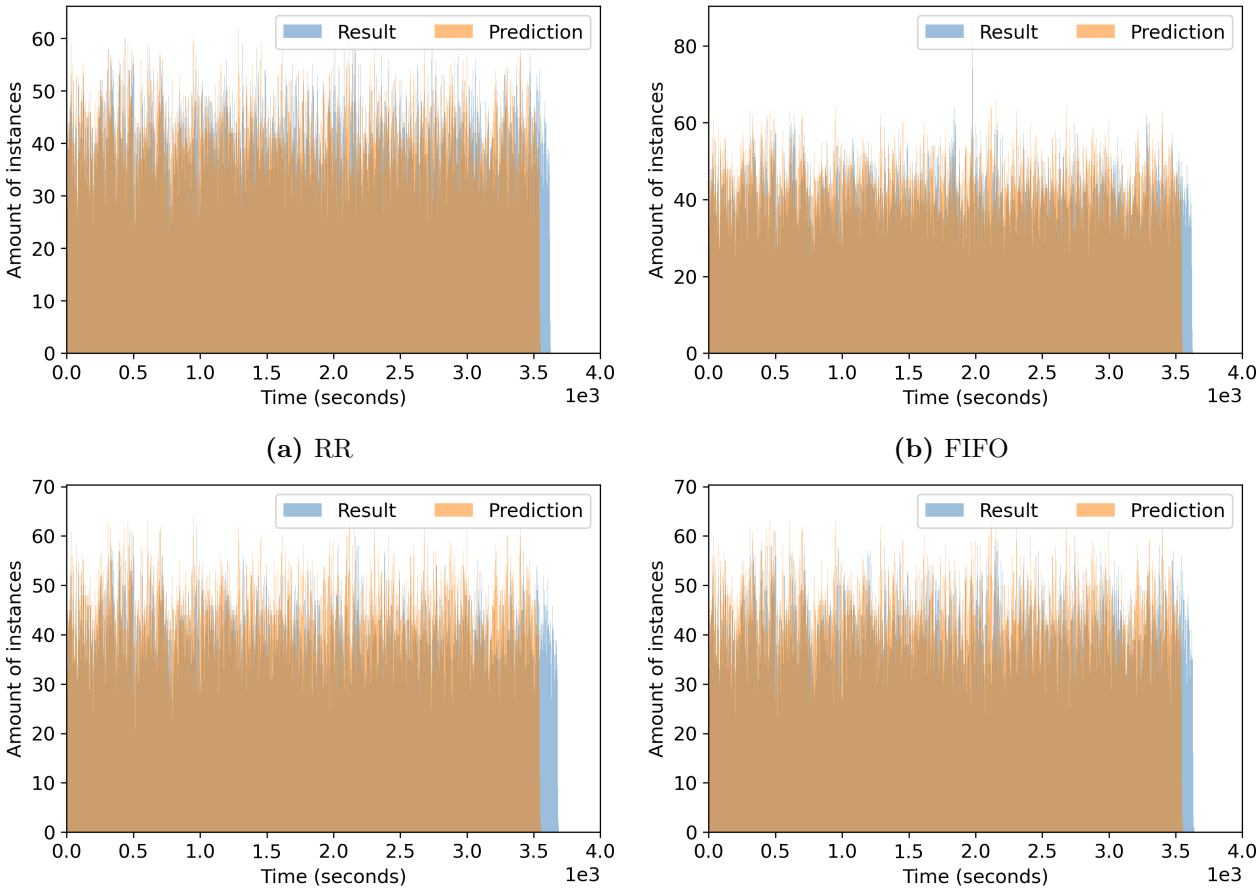


(a) *W1.* **(b)** *W2.*

(c) *W3.*

Figure 15: Results of the average of the calculated time-deltas for each workload and each scheduler, including the error bars.

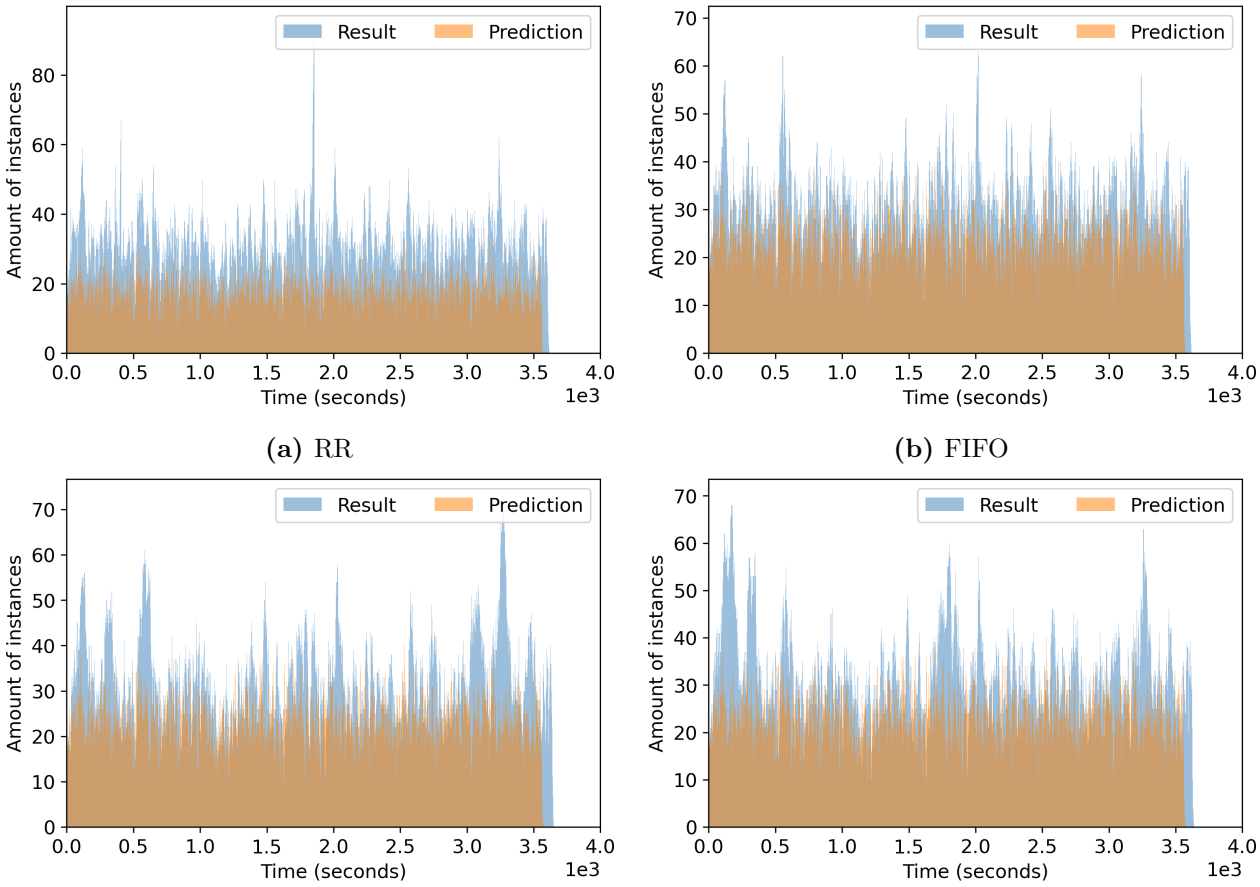
The concurrency histograms of the workloads we do not discuss in Section 5.4.1 are shown in Figure 16 and Figure 17.



(a) RR **(b)** FIFO

(c) CFS **(d)** BATCH

Figure 16: The amount of concurrent instances per unit of time of the four different schedulers for workload *W2*. The selected unit of time to bin concurrent events in, is 1 second. For CFS, the prediction expects more concurrency in each bin, than the results shows. For BATCH, FIFO and RR, the predicted concurrency and the actual concurrency differ less.



(a) RR

(b) FIFO

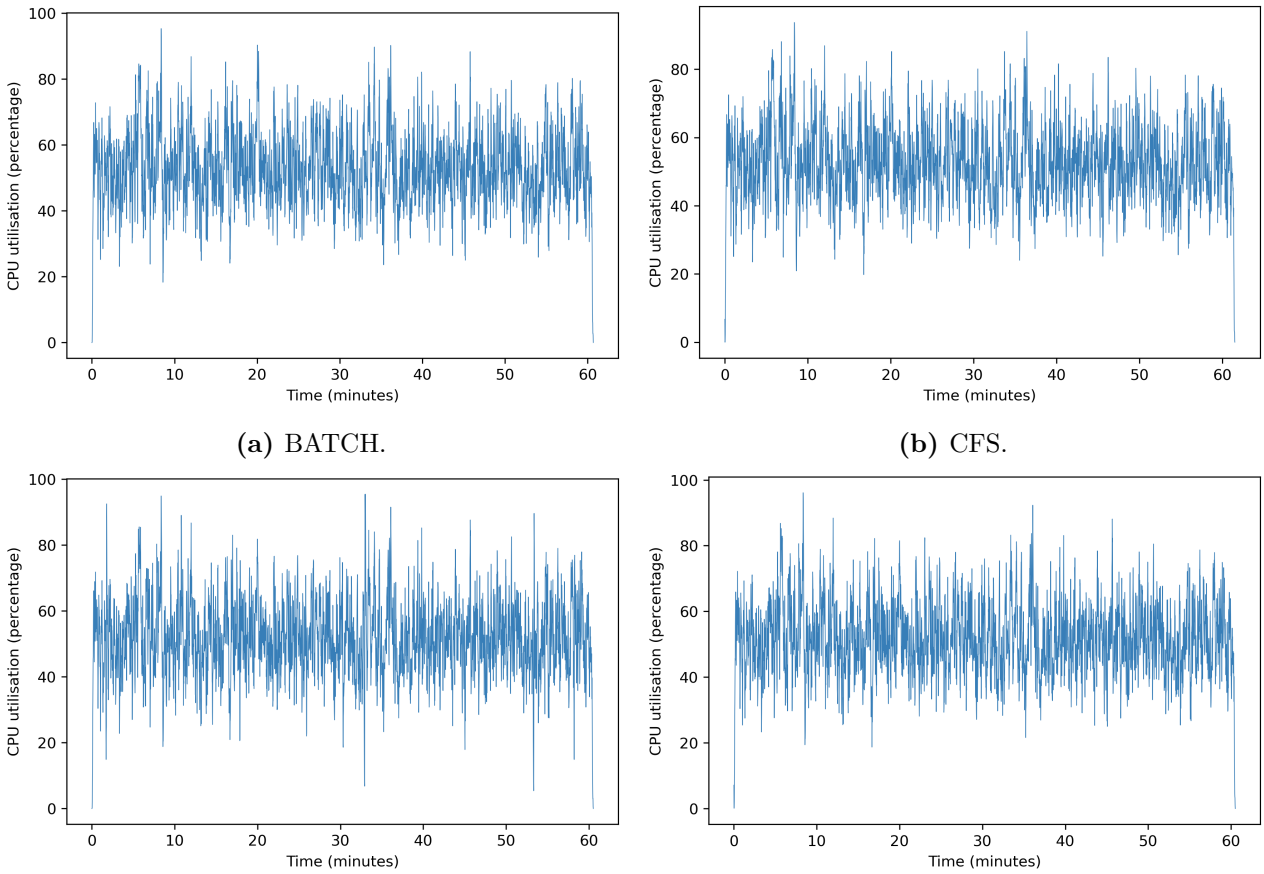
(c) CFS

(d) BATCH

Figure 17: The amount of concurrent instances per unit of time of the four different schedulers for workload *W3*. The selected unit of time to bin concurrent events in, is 1 second. For CFS, the prediction expects more concurrency in each bin, than the results shows. For BATCH, FIFO and RR, the predicted concurrency and the actual concurrency differ less.

A.2 Resource Utilisation of different Schedulers

The raw results for the CPU utilisation over time for workloads *W2* and *W3* which were excluded from Section 5.4.2.



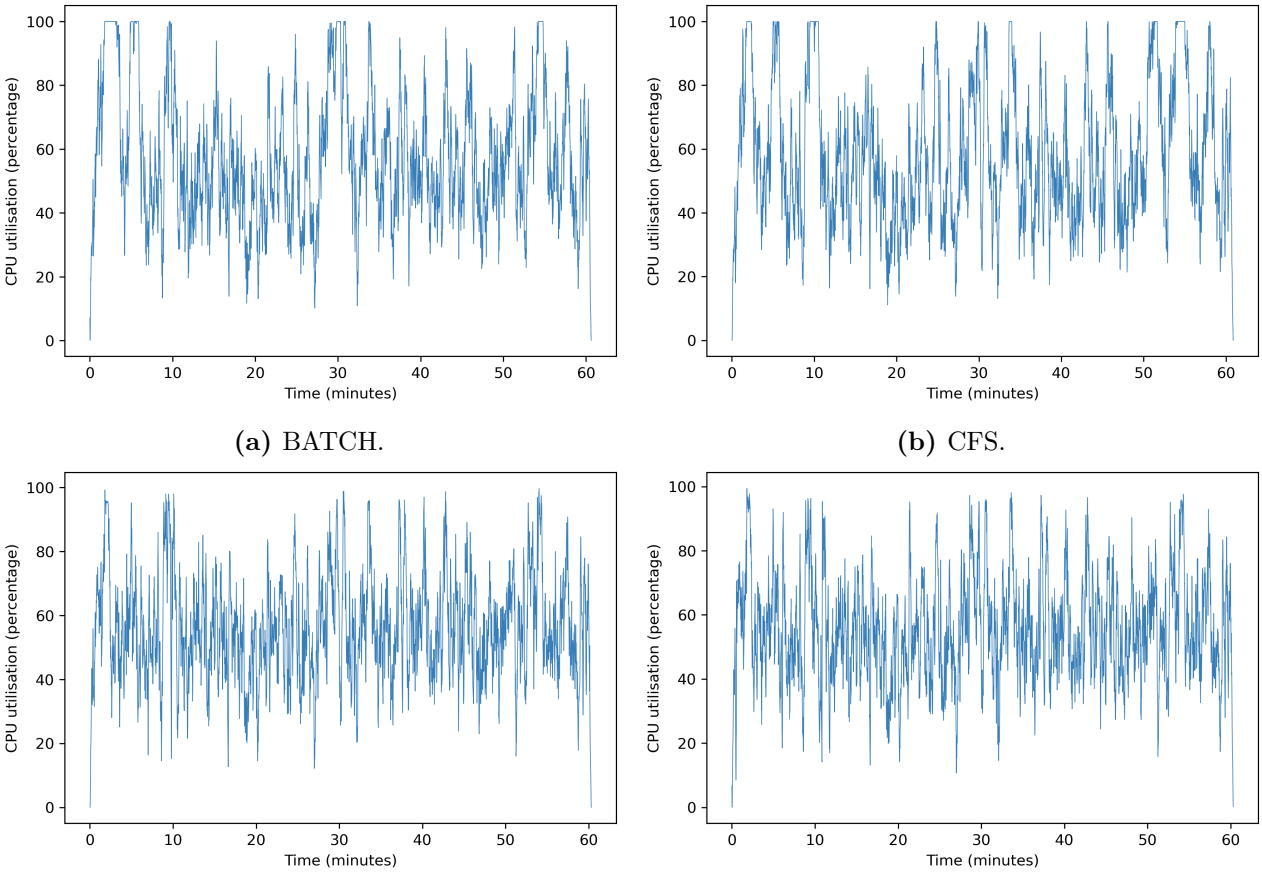
(a) BATCH.

(b) CFS.

(c) FIFO.

(d) RR.

Figure 18: The CPU utilisation over time for all schedulers when executing workload *W2*. The vertical axis shows CPU utilisation as a percentage.



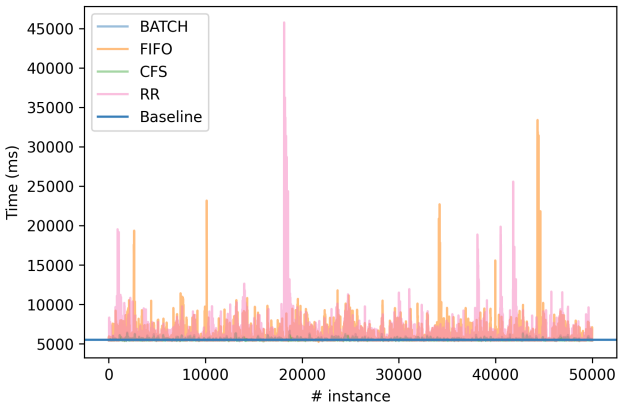
(a) BATCH. **(b)** CFS.

(c) FIFO. **(d)** RR.

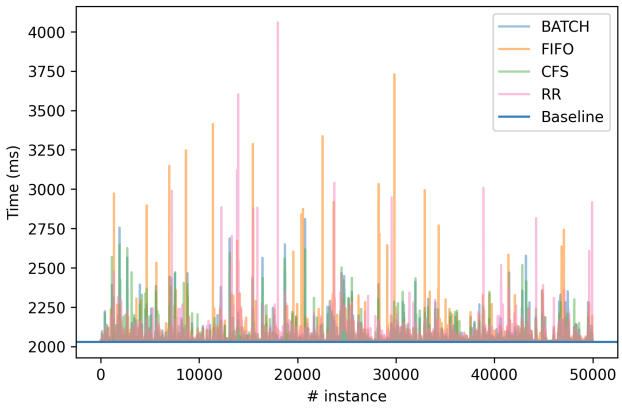
Figure 19: The CPU utilisation over time for all schedulers when executing workload *W3*. The vertical axis shows CPU utilisation as a percentage.

A.3 Numerical Analysis

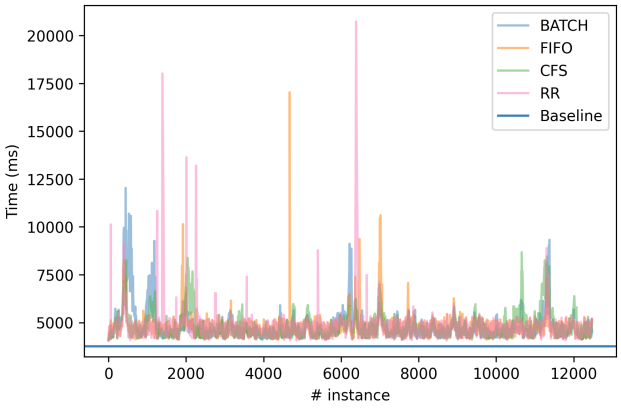
The figures of the workloads we do not discuss in Section 5.4.3:



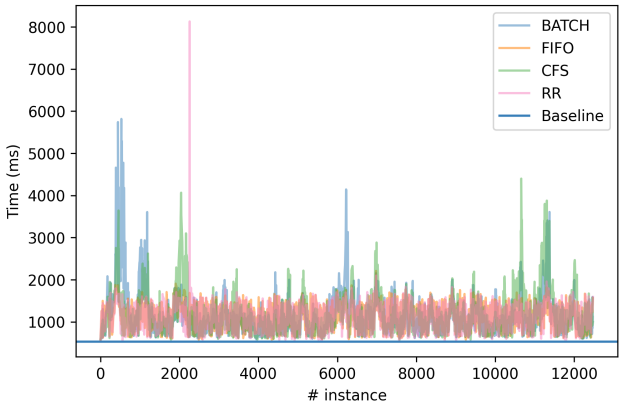
(a) *W1* `tFC`.



(b) *W1* `tVM`.

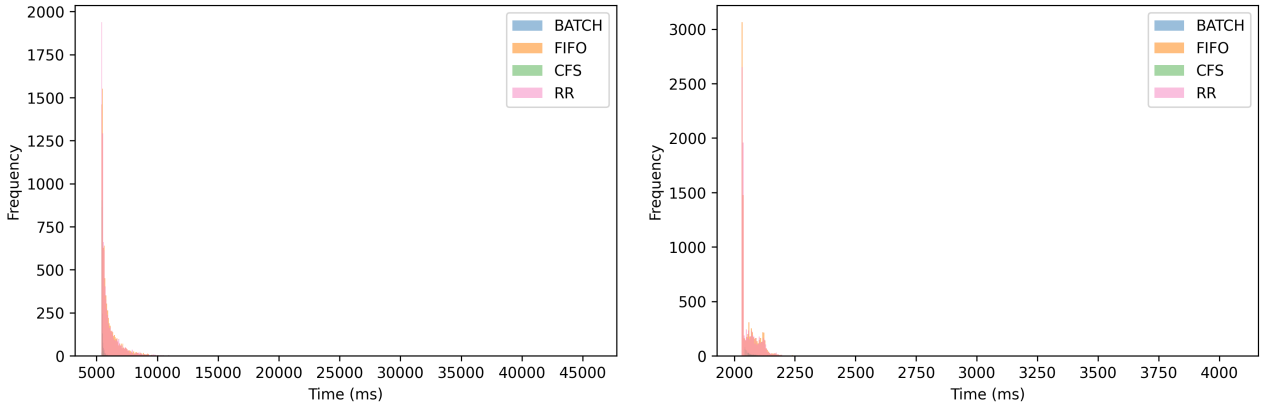


(c) *W3* `tFC`.



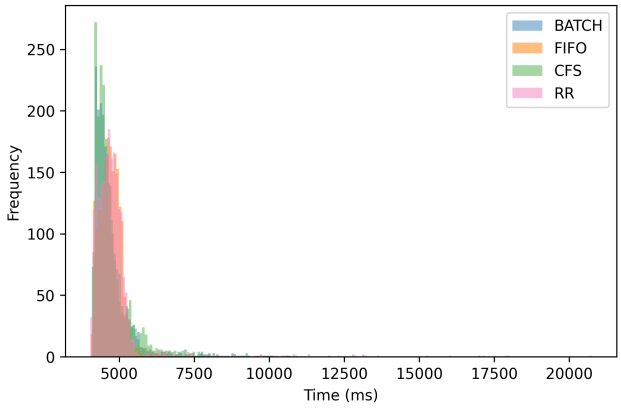
(d) *W3* `tVM`.

Figure 20: The run-time of the selected program when executing at different indices in workload *W1* and in workload *W3*. The horizontal axis shows the index of the program in the workload. The vertical axis shows execution-time of this program in milliseconds.

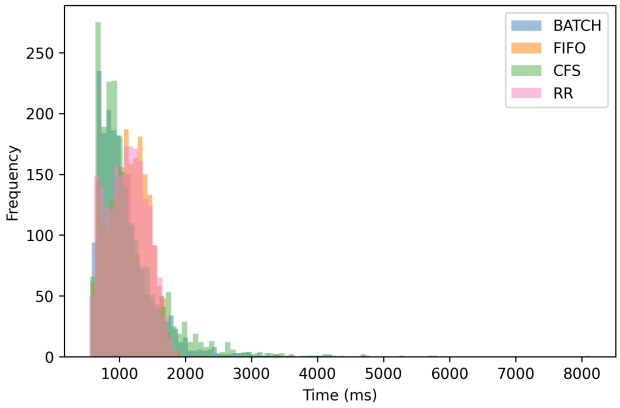


(a) *W1* `tFC`.

(b) *W1* `tVM`.



(c) *W3* `tFC`.



(d) *W3* `tVM`.

Figure 21: The histogram of the individual run-times of the selected program for workloads *W1* and *W3*. The bin size is determined by the Freedman-Diaconis rule [11]. The vertical axis shows the run-time in milliseconds.

A.4 Workloads

Table 6 lists all workloads which were used in experimentation. For all results, we refer to the GitHub repository [7].

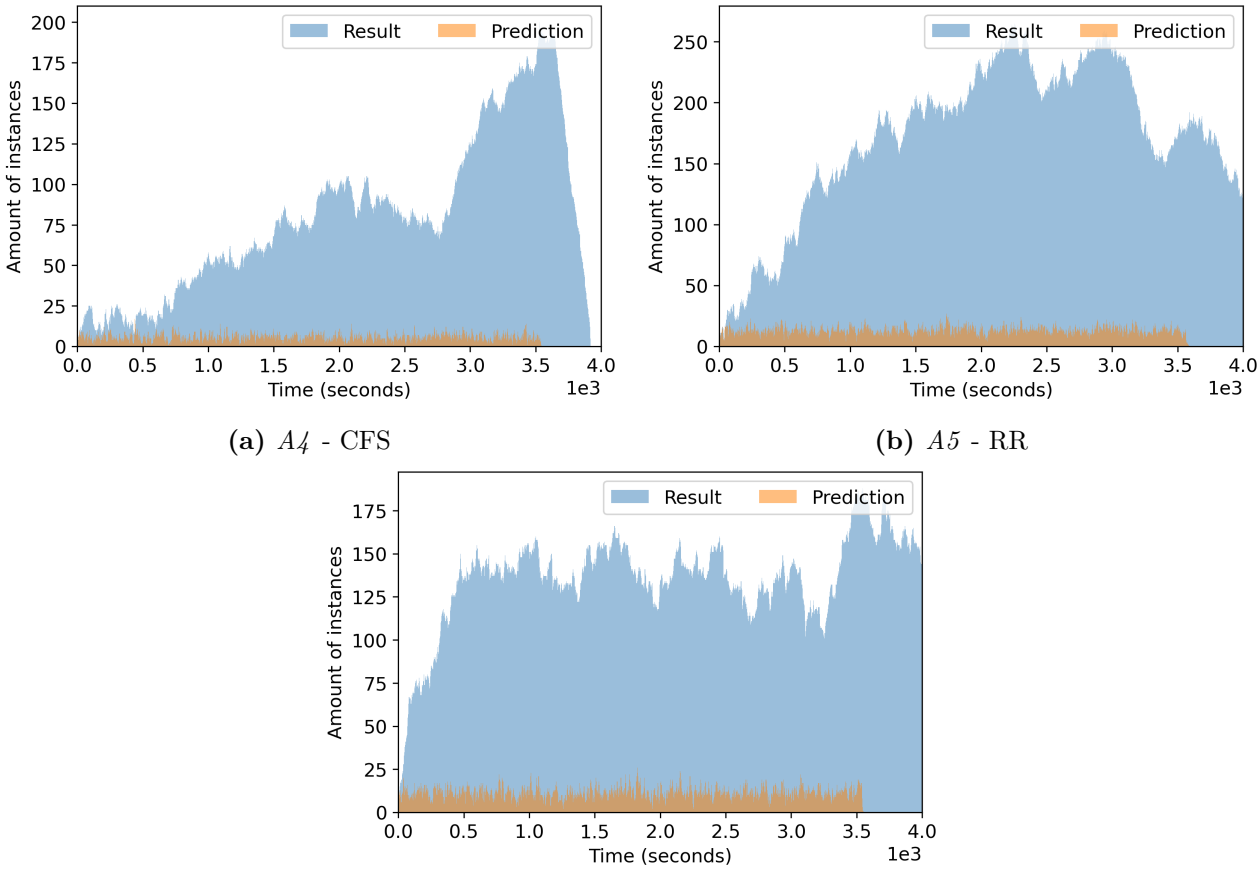
Workload	Amount of tasks	Run-time (h)	Portion CPU	Portion MEM	Portion IO
A1	2,500	1.0	1.0	0.0	0.0
A2	2,500	1.0	1.0	1.0	1.0
A3	2,500	1.0	0.0	0.0	1.0
A4	2,500	1.0	0.0	1.0	0.0
A5	5,000	1.0	1.0	1.0	1.0
A6	5,000	1.0	0.0	1.0	0.0

Table 6: A list of all workloads we created and ran on at least one machine and scheduler. This list excludes the workloads defined in Table 3.

[7] https://github.com/NJLBoonstra/fc-microbenchmark

A.5 Examples of Rejected Workloads

In Section 5.3, we explain workloads can either oversubscribe the system too much, or not at all. This section shows examples of workloads that oversubscribe the system too much in Figure 22. Examples of workloads that do not stress the system enough are shown in Figure 23.

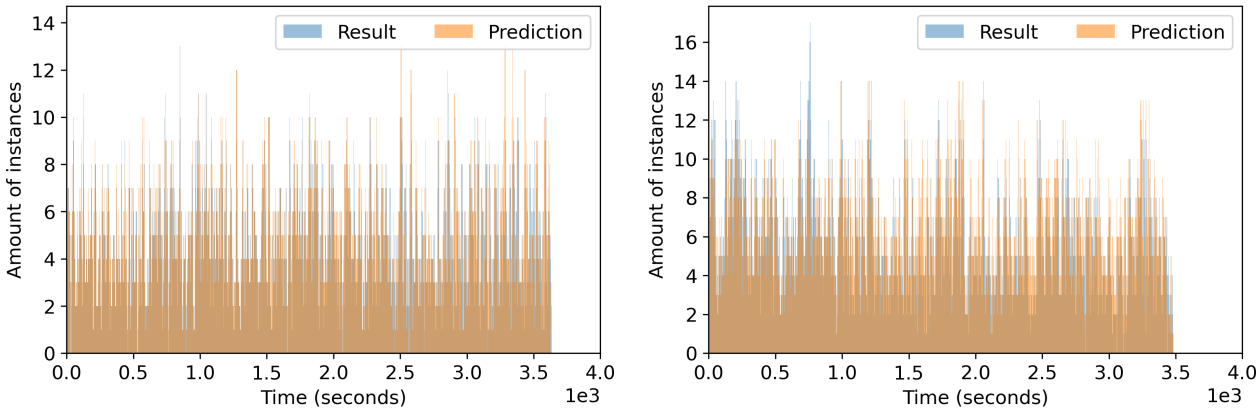


(a) *A4* - CFS

(b) *A5* - RR

(c) *A6* - FIFO

Figure 22: Concurrency histograms that oversubscribed the machine too much and were rejected for further analysis.



(a) *A1* - CFS

(b) *A2* - FIFO

Figure 23: Concurrency histograms that do not impose enough load on the machine and were rejected for further analysis.