Scheduling algorithms for improved CI performance

Masters thesis in Algorithms, Languages and Logic

Master's thesis in Computer science and engineering

Zacharias Faleberg Nilsson
Freddy Abrahamsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Scheduling algorithms for improved CI performance

Masters thesis in Algorithms, Languages and Logic

Zacharias Faleberg Nilsson
Freddy Abrahamsson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Scheduling algorithms for improved CI performance
Masters thesis in Algorithms, Languages and Logic
Zacharias Faleberg Nilsson
Freddy Abrahamsson

© Zacharias Faleberg Nilsson, 2023.
© Freddy Abrahamsson, 2023.

Supervisor: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering
Advisor: Jim Fischer, Volvo Technology AB
Examiner: Nir Piterman, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Scheduling algorithms for improved CI performance
Masters thesis in Algorithms, Languages and Logic
Zacharias Faleberg Nilsson
Freddy Abrahamsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Continuous Integration (CI) is a software engineering practice aimed at making it easier to integrate code from many developers in a shared code base. An important part of this process is automated building and testing of new code, resulting in various jobs that should be executed on servers controlled through a CI system.

This project evaluates the performance of seven algorithms for job scheduling in a Jenkins based CI environment. Using five separate performance metrics, we look at the trade-offs between different algorithms and discuss their relevance to the experience of using the CI service. Evaluations were mainly carried out in a simulated environment, modeling the Jenkins server, build servers and workloads. The obtained results are then compared to a smaller data set generated on a real Jenkins server, verifying the validity of the simulated results.

The results indicate that the performance of the scheduler can be improved by using algorithms based on known data about the jobs being executed. By selecting a suitable algorithm, it is possible to reduce the cost, as defined by the chosen cost model, compared to the default first-in-first-out strategy used by Jenkins. Comparing the simulated results against data from a real Jenkins server shows that the simulated model is a good representation of the real system, strengthening the validity of the obtained results.

Keywords: Scheduling, Continuous Integration, Jenkins.

Acknowledgments

This thesis work was conducted in the Propulsion Software Factory at Volvo Technology AB. We would like to thank them for providing the opportunity and for welcoming us to the team. We would also like to thank the whole CI team, and especially Alexander Sandberg, for supporting us during the project, explaining the details of the system and helping us move forward.

Finally, a warm thank you to our examiner Nir Piterman for valuable feedback and to our supervisor Ahmed Ali-Eldin Hassan for his continuous support along the thesis work, encouraging us and suggesting ways to solve the problems we faced.

Zacharias Faleberg Nilsson & Freddy Abrahamsson, Gothenburg, June 2023

Contents

List of Figures xi

List of Tables xiii

Glossary xv

Acronyms xvii

Symbols xix

1 Introduction 1
- 1.1 Terminology . . . 2
- 1.2 Research questions . . . 2
- 1.3 Limitations . . . 2
- 1.4 Related work . . . 2
- 1.5 Outline of this report . . . 3

2 Background 5
- 2.1 Continuous Integration . . . 5
  - 2.1.1 Jenkins . . . 6
- 2.2 Description of the environment . . . 7
  - 2.2.1 Job characteristics . . . 7
  - 2.2.2 Job data . . . 8
- 2.3 Scheduling . . . 8
  - 2.3.1 Online vs. offline scheduling . . . 9
  - 2.3.2 Preemptive scheduling . . . 9
  - 2.3.3 Clairvoyant scheduling . . . 9
  - 2.3.4 Stochastic scheduling . . . 9
- 2.4 Measuring performance . . . 9
  - 2.4.1 Linear cost functions . . . 10
  - 2.4.2 Non-linear cost functions . . . 10
- 2.5 Scheduling algorithms . . . 10
  - 2.5.1 Single queue algorithms . . . 10
  - 2.5.2 Separate queues . . . 11
  - 2.5.3 Avoiding starvation . . . 12

3 Methods **13**
3.1 Testing in the simulator . . . 13
3.1.1 Comparing different algorithms . . . 13
3.1.2 Expected vs. actual processing times . . . 15
3.1.3 Effects of adding more agents . . . 15
3.1.4 Simulated jobs and environment . . . 15
3.2 Simulator implementation details . . . 16
3.2.1 Job generation . . . 16
3.2.2 Limitations . . . 17
3.3 Jenkins implementation . . . 17
3.3.1 Priority Sorter plugin . . . 17
3.3.2 Data retrieval . . . 18
3.3.3 Job dispatcher . . . 19

4 Results **21**
4.1 Simulating real jobs and pipelines . . . 21
4.1.1 Comparing different algorithms . . . 21
4.1.2 Completion times for individual jobs . . . 25
4.1.3 Queue times for individual jobs . . . 26
4.1.4 Summary of algorithm performance . . . 28
4.1.5 Expected vs. actual processing times . . . 28
4.1.6 Adding more agents . . . 29
4.2 Simulating individual jobs . . . 31
4.2.1 Makespan . . . 31
4.2.2 Waiting times . . . 31
4.3 Jenkins results . . . 32
4.3.1 Makespan . . . 32
4.3.2 Average waiting times . . . 33

5 Conclusion **35**
5.1 Discussion . . . 35
5.1.1 Relevance of the performance metrics . . . 35
5.1.2 Correlation between metrics . . . 36
5.1.3 Algorithm performance . . . 37
5.1.4 Results from the Jenkins server . . . 38
5.1.5 Ethical considerations . . . 38
5.2 Future work . . . 39
5.2.1 Job prediction module . . . 39
5.2.2 Machine learning approaches . . . 39
5.2.3 Discounted total weighted completion time algorithm . . . 39
5.2.4 More algorithms for Jenkins server . . . 39
5.3 Conclusion and summary . . . 40

Bibliography **41**

A Box plot reading guide **I**

List of Figures

2.1 Overview of the CI process 6
2.2 Overview of the pipeline structure 7
2.3 Histogram displaying the processing times for past executions of a job. 8

3.1 The cost of a pipeline according to the discounted pipeline lifetime metric. 14
3.2 Schematic overview of the simulator 16
3.3 Structure of the Jenkins plugin 18

4.1 Makespan for pipeline simulations 22
4.2 Average pipeline lifetimes during simulations 23
4.3 Maximum pipeline lifetime during simulations 24
4.4 Discounted pipeline cost during simulations 25
4.5 Total weighted completion time during simulations 25
4.6 Average queue times during simulations 26
4.7 Maximum queue times during simulations 27
4.8 Total weighted completion times for schedulers with access to (a) the actual processing time of each job or (b) an expected value for the processing time. 29
4.9 Total weighted completion times for different numbers of executors . . 30
4.10 Makespan for simulated jobs 31
4.11 Average queue times for simulated jobs 32
4.12 Maximum queue times for simulated jobs 32
4.13 Makespan for jobs executed on the Jenkins server 33
4.14 Average queue times for jobs on the Jenkins server 33
4.15 Maximum queue times for jobs on the Jenkins server 34

A.1 Sample boxplot showing the median, quartiles and outliers for a set of values. I

List of Tables

4.1 Algorithm performance summary 28

Glossary

Definitions marked with (Jenkins) are taken directly from the official Jenkins glossary [1].

Term	Definition
Agent (Jenkins)	An agent is typically a machine, or container, that connects to a Jenkins controller and executes tasks when directed by the controller.
aging factor	An increased weight or priority given to jobs waiting in a queue to avoid starvation.
clairvoyant (scheduling)	A clairvoyant scheduler is one with access to information such as processing time for the jobs being scheduled.
competitive ratio	Ratio between the cost of a schedule produced by an online algorithm and an optimal schedule produced for an offline version of the same problem.
completion time	Time at when a job finishes processing.
Controller (Jenkins)	The central, coordinating process that stores configuration, loads plugins, and renders the various user interfaces for Jenkins.
Core (Jenkins)	The primary Jenkins application (`jenkins.war`) that provides the basic web UI, configuration, and foundation upon what plugins can be built.
Executor (Jenkins)	A slot for execution of work defined by a Pipeline or job on a Node. A Node may have zero or more Executors configured which corresponds to how many concurrent Jobs or Pipelines are able to execute on that Node.
interquartile range	Distance between the lower and upper quartiles in a data set..
Jenkins	Jenkins is an open source software for automating builds.

Term	Definition
Job (Jenkins)	A user-configured description of work that Jenkins should perform, such as building a piece of software, run a test etc.
Label (Jenkins)	User-defined property for grouping Agents, typically by similar functionality or capability. For example `linux` for Linux-based agents or `docker` for Docker-capable agents.
makespan	Completion time of the last job in a batch to be completed.
Master	Deprecated term.
offline (scheduling)	In an offline scheduling problem, the full set of tasks is known to the scheduler when the scheduling process start, which means the full schedule can be determined before any execution starts.
online (scheduling)	In an online scheduling problem, the full set of tasks is unknown to the scheduler when the scheduling process start.
Pipeline (Jenkins)	A user-defined set of stages to build, test and deploy software using Jenkins..
Plugin (Jenkins)	An extension to Jenkins functionality provided separately from Jenkins Core.
preemptive (scheduling)	a preemptive scheduler is one that can interrupt ongoing jobs and restart them at a later point in time.
release time	The release time of a job j is the time at which j becomes available for processing.
Slave	Deprecated term.
Stage	Part of a Jenkins pipeline, such as *build* or *test*.
starvation	The situation where a job is denied access to required resources forever due to a continuous flow of incoming jobs with higher priority.
version control system	Software used to keep track of changes in a codebase.
weight	Job specific property denoting the importance of a job.

Acronyms

CI Continuous Integration.

FIFO First In-First Out.

HDF Highest Density First.

IQR *interquartile range.*

LDF Lowest Density First.
LIFO Last In-First Out.
LPT Longest Processing Time.

SMRTS Shortest Maximum Remaining Time in Stage.
SPT Shortest Processing Time.
STRTS Shortest Total Remaining Time in Stage.

WLPT Weighted Longest Processing Time.
WSPT Weighted Shortest Processing Time.

Symbols

The following variable names and symbols are used throughout the report

C_j	Random variable describing the completion time of job j.
c_j	Completion time of job j.
i	A machine for executing tasks.
J	A set of jobs.
j	A single job.
p_{ij}	Processing time of job j on machine i in the case of unrelated machines.
p_j	Processing time of job j.
P_{ij}	Random variable describing the processing time of job j on machine i in the case of unrelated machines.
P_j	Random variable describing the processing time of job j.
r_j	Release time of a job j.
t	Time variable.
w_j	Weight of job j.
ρ	competitive ratio.

1
Introduction

Modifying or extending a piece of software consists of multiple steps, such as writing new code, testing the new code, integrating the new code into the existing code base and deploying a new version of the product. Integration is historically a big challenge, especially if multiple developers work in parallel. If several different teams have worked for weeks on different features, there is a high likelihood that their code will not fit together perfectly.

Continuous Integration (CI) is a software engineering practice where developers continuously integrate smaller chunks of new code into the the existing code base instead of building large features which are then integrated into the main product at a single point [2]. As part of the CI process, developers frequently commit their code into a shared code repository with the goal of resolving conflicts within the code as early as possible. When a developer adds new code, that code is automatically built (if needed) and tested using automated test frameworks [3][4]. Running tests automatically on the new code when it is committed to the repository also helps developers catch, and remove, bugs within the code as soon as possible.

Jenkins is an open source automation tool for the CI process, forked from the Hudson project in 2011 [5]. The Jenkins environment consists of a *controller* that handles various *job*s, such as building and testing code or running simulations, and assigns them to *agent*s where the jobs are executed in slots called *executor*s. Jobs can run on a regular schedule, or be triggered by events such as a developer committing some code to a repository. If no executors are available when a job is submitted, it is placed in a queue, waiting for a suitable executor to become available [6].

This project investigates how the performance of a Jenkins based CI environment can be improved using a scheduling algorithm that takes known characteristics of each job and agent into account instead of using the default Jenkins scheduler that schedules jobs using a simple *First In-First Out* (FIFO) based approach. The project was carried out at Volvo Technology, in the *Propulsion Software Factory* that operates a Jenkins based CI environment serving approximately 400 software developers writing software for both conventional and electrical vehicle powertrain. In the factory, a variety of jobs run on a set of build servers (agents), managed by a Jenkins controller.

1.1 Terminology

Within the CI community, there are several ways to denote the various entities involved in the process of building and integrating software. The Jenkins project started updating the terminology used for describing a Jenkins environment in 2016 [7], with the goal of making the terminology more inclusive and better aligned with the project values. In this process, terms such as *master*, *slave*, *blacklist* and *whitelist* were replaced with *controller*, *agent*, *denylist* and *allowlist* respectively.

In this report names are primarily taken from the official Jenkins terminology, agreed upon by the Jenkins community [1]. This naming system may differ from other reports on the same subject. We advise the reader to consult the included glossary in case of confusion.

1.2 Research questions

The main research questions investigated during the project are:

1. Can the performance of the CI pipeline be improved by using scheduling algorithms utilizing existing knowledge of the jobs and machines available?
2. How do different scheduling algorithms perform compared to each other?
3. How do differences in data available to the scheduler affect its performance?

Due to the quantitative nature of scheduling, it is natural to evaluate the scheduler, and answer these questions, using some numerical values, as described in section 2.4.

1.3 Limitations

Although the algorithms implemented should be useful in a general setting, potentially with some restrictions, they will only be evaluated in the available environment at Volvo and the implemented simulator. Running the algorithm in other environments could help identify how various parameters in the environment affect the performance of the scheduler that could lead to insights about further improvements.

A very useful feature in a scheduler would be the ability to predict incoming jobs. While there are several interesting approaches to implement such a feature, it is beyond the scope of this project.

1.4 Related work

Previous master theses [8][9] have investigated ways of improving Jenkins performance. Andersson and Andersson [8] studies dynamic allocation of executors and load balancing mechanisms while Berglund and Eriksson [9] compare different algorithms measured with various performance measures under the assumption of all

machines being equal. Berglund and Eriksson specifically mentions studying different job sets and heterogeneous machines as suggestions for future work [9, p. 41-42].

1.5 Outline of this report

Chapter 1 and chapter 2 introduces the necessary concepts from continuous integration and scheduling. Chapter 3 describes the implementation of the algorithms along with design choices made in the development process. Chapter 4 presents the acquired results while Chapter 5 discusses them and the project as a whole.

2

Background

This section consists of an introduction to some basic concepts in continuous integration, and the need for efficient processing of the jobs in a CI pipeline. Further, an overview of the CI environment where the project was carried out will be presented. Finally, some basic scheduling theory, explaining the problem at hand, metrics to evaluate solutions, common algorithms to solve it and constraints that must be handled during the process.

2.1 Continuous Integration

As mentioned in chapter 1, *Continuous Integration* (CI) is built around the idea of developers integrating their code into a common code base frequently. At the extreme, this would mean all coders worked on the same copy of the code, instantly changing it for everyone involved, thereby *continuously* integrating all new features [2]. As this is not feasible in a real world scenario, CI advocates instead suggest that developers commit their code in short intervals, ranging from every few hours up to every day [2] [3].

The building and testing of new code is often performed on a set of dedicated servers, controlled by a single CI server. To streamline the integration process, building and testing is automated so that when a developer pushes some code to a central repository in a version control system, such as Git or Subversion, the CI server will create a set of jobs to build and test the new code with no interaction from the developer. If the code passes all tests, the developer can go ahead and merge the new code into the shared code base. A graphical overview of the process can be found in figure 2.1. Widely used tools for automating the CI process include Travis CI[1], GitHub Actions[2] and Jenkins.

[1] `https://www.travis-ci.com/`
[2] `https://github.com/features/actions`

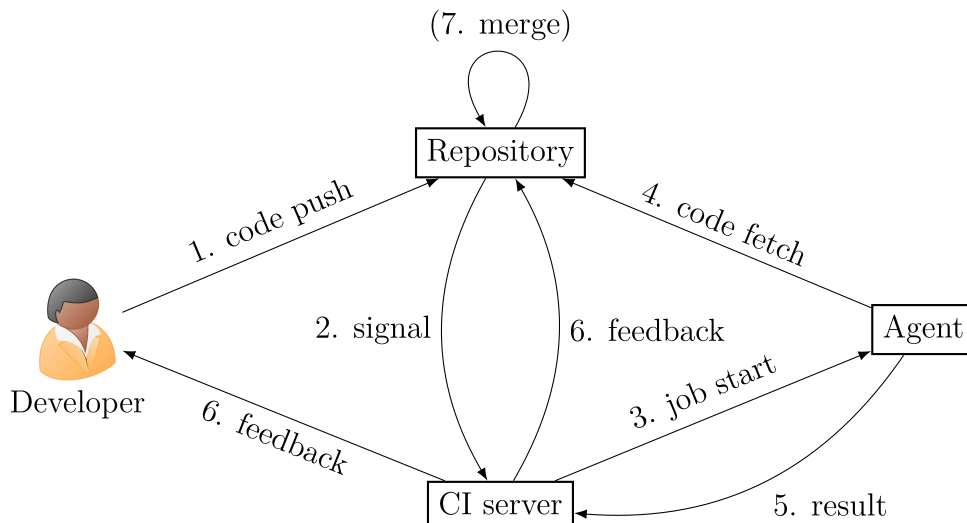


Figure 2.1: Overview of the CI process. The process begins when a developer pushes code (1) to a repository server. The repository server signals (2) the CI server that in turn starts a job (3) on an agent. The agent fetches code (4) from the repository, compiles it and runs a set of tests. The result is sent (5) back to the CI server that provides feedback (6) to the repository server and the developer. If the tests passed, the repository can allow the new code to be merged (7) into the main code base. If the process includes multiple jobs, steps 3-5 may be repeated.

2.1.1 Jenkins

Jenkins is an open source tool for automating CI processes, originally created as *Hudson* in 2005 by Kohsuke Kawaguchi [10]. Jenkins consists of a *core*, which can be extended, using *plugins*. The process of building and testing code using Jenkins is often organized in a *pipeline*, that includes a number of *stage*s executed in sequence. Each stage can include multiple jobs running in parallel on separate agents.

The default Jenkins scheduler uses a FIFO based approach to schedule jobs on agents. Jobs are kept in a single queue and when a job reaches the front of the queue, Jenkins assigns the job to an agents using an algorithm based on consistent hashing[3]. First, Jenkins hashes the name of the agents such that the hash is proportional to the amount of available executors, then Jenkins hashes the incoming job name. This creates a priority list for the jobs, where each job has a list of preferred agents[11]. When the hashing is done, the job is assigned to the most preferred agent, with the appropriate *label*s, currently available.

[3]While the details of consistent hashing are not needed in order to understand this report, the interested reader can find more details at `https://en.wikipedia.org/wiki/Consistent_hashing`

2.2 Description of the environment

As mentioned in chapter 1, this project was carried out at Volvo's Propulsion Software Factory. While the CI environment is used for many different kinds of jobs, the majority of the tasks are organized as pipelines with similar characteristics and consist of three main stages, a *pre-build* check, a *build* stage and a *post-build* stage. In each job, the pre- and post-build checks consist of a few subtasks, all executed sequentially on a single machine. The build stage consists of a number of different builds that can be executed in parallel. Each build job has some required features from the builder, and each builder has a set of capabilities.

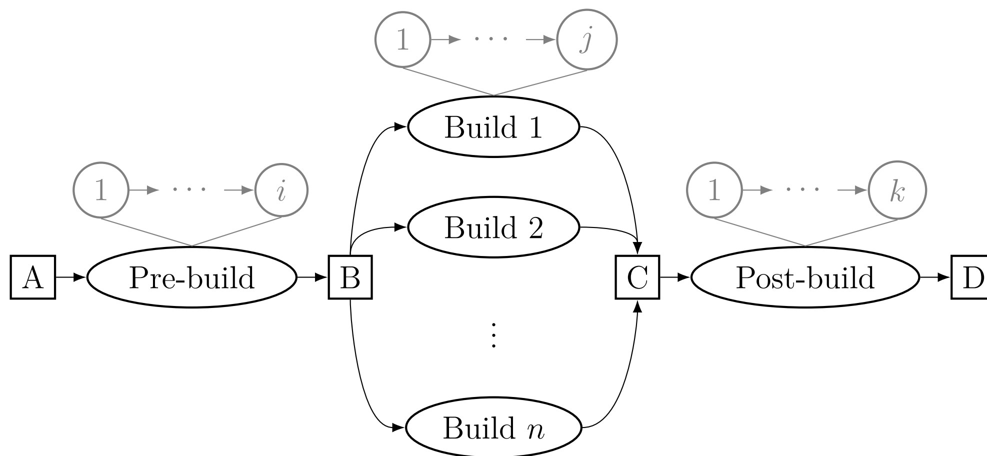


Figure 2.2: Overview of the workflow for the majority of the pipelines running in the environment. pipelines are submitted to the queue A, where they are scheduled for the pre-build stage. After passing the pre-build stage, each build job is placed in queue B, waiting for an available builder agent. When all builds are finished, a single job is placed in queue C until the post-build stage can be executed. Once the post-build stage has passed, the job ends up at point D where it is finished. In this case, the pre-build has i steps, build 1 has j steps and the post-build has k steps.

2.2.1 Job characteristics

While many of the build jobs being executed on the builders are similar, the exact processing time of a job is not known when it is submitted to the scheduler. Factors such as slowdowns on the network, speedup from caching mechanisms, and individual variations between the machines mean that the time needed to build a project may vary significantly. Figure 2.3 shows a histogram over past executions of a job, given the parameters available to the scheduler.

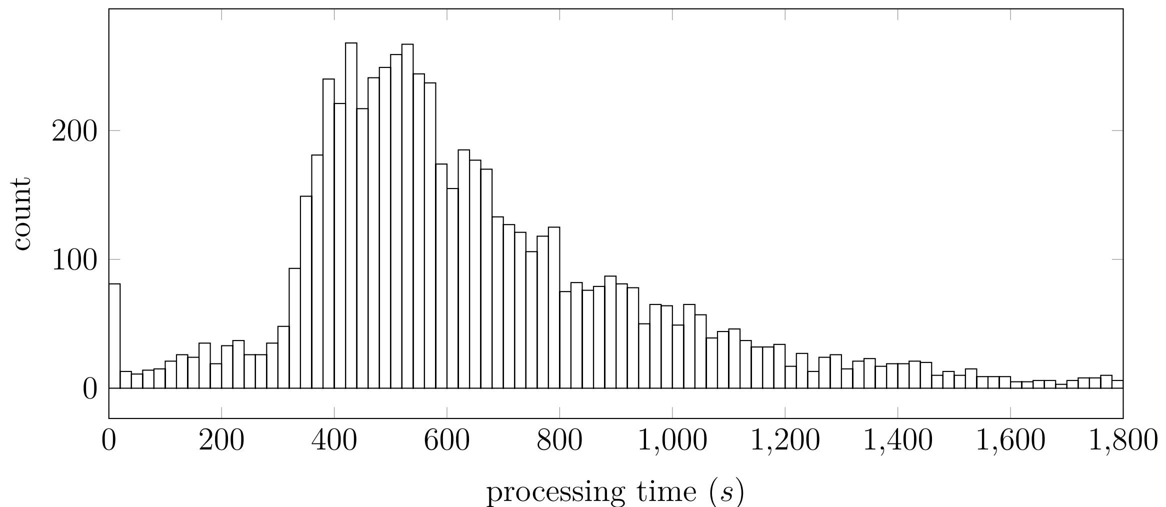


Figure 2.3: Histogram displaying the processing times for past executions of a job.

2.2.2 Job data

During the processing of a job, data about the execution is recorded at various stages of the process. This data is stored using a logging service called *Elastic* and available through an API, allowing it to be used when scheduling future jobs.

2.3 Scheduling

Scheduling "...deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives."[12, p. 1]. In computer systems, these tasks can be anything from multiple processes requiring access to a single CPU to large computational jobs submitted to a super computer or a data center. For CPU access, the scheduling is traditionally handled by the operating system [13], while tasks submitted to large clusters may be scheduled by a cluster manager such as Google Borg [14].

As described by the quote above from Pinedo [12], solving a scheduling problems means finding an algorithm for allocating resources to a set of tasks such that a certain metric is minimized (or maximized). Scheduling problems can be classified based on several characteristics, such as the amount of information available about each job during the scheduling process, whether the scheduler can interrupt and resume tasks once they have been started, whether all tasks are known to the scheduler before execution starts etc. The scheduler implemented in this project is a *clairvoyant*, *online*, non-*preemptive* scheduler working on jobs with stochastic processing times across a set of unrelated machines.

2.3.1 Online vs. offline scheduling

An *offline* scheduler is a scheduler with access to the full set of tasks to be executed in advance, meaning the full schedule can be created before any execution begins. In contrast, an *online* scheduler deals with jobs that are presented one after the other. This can be in the form of a list of jobs, presented in a specific order at $t = 0$, or each job can become available at $t = r_j$, where r_j is called the *release time* of the jobs j [12].

For an online scheduling algorithm, the *competitive ratio* is defined as the upper bound for the performance value (which should be minimized) of the scheduler, compared to an offline version of the same problem. If the cost of the schedule generated by the online scheduler is at most ρ times that of the optimal schedule, the scheduler is said to be *ρ-competitive* [12].

2.3.2 Preemptive scheduling

Preemptive scheduling is a scheduling process where a running task can be interrupted (preempted) in order to make the resource available to another task. A typical example is CPU scheduling where the CPU can only work on one process. If several processes require access to to CPU, then each will be allowed to run for a short while before the resources is handed to the next process. The opposite of a preemptive scheduler is called *non-preemptive* [13].

2.3.3 Clairvoyant scheduling

Another important characteristic of the scheduling problem is whether the scheduler has access to information about the jobs being scheduled at the time of scheduling. A clairvoyant scheduler is one that has access to the processing time of the job at the time of scheduling, meaning the information can be used as part of the scheduling process [12]. A scheduler without access to this information is called *non-clairvoyant*.

2.3.4 Stochastic scheduling

In real world scheduling problems, the processing time of a task may not be known in advance [12]. As mentioned in section 2.2.1, the build jobs running in the CI environment used in this project are affected by factors such as network delays and cache hits. Instead, in *stochastic scheduling*, the processing time of a job j on a machine i may be described by a random variable P_j [12][15].

2.4 Measuring performance

The performance of a scheduler can be evaluated using several different metrics, where the scheduler aims to minimize or maximize the chosen metric. Many of these metrics are based around the *weight*, w_j, and *completion time*, c_j, of a job j.

2.4.1 Linear cost functions

For a set of jobs, J, the *total completion time* is defined as the sum of the completion times of all jobs in J.

$$\text{total completion time} = \sum_{j \in J} c_j$$

The *makespan* is defined as the completion time of the last job in J to finish.

$$\text{makespan} = \max_{j \in J} c_j$$

Additionally, a job j may have an associated weight, w_j, indicating the importance of the job, defined by the user. From this, the total completion time metric can be extended to the *total weighted completion time*, where the weight of each job is included as well [12].

$$\text{total weighted completion time} = \sum_{j \in J} w_j c_j$$

2.4.2 Non-linear cost functions

Sometimes, the linear cost functions mentioned in the previous section do not accurately reflect the cost of a job waiting in the queue. One simple extension to the *total weighted completion time* is the *discounted total weighted completion time*, defined in equation (2.1). This cost model describes a situation where the cost per time unit of waiting for a job decreases over time, asymptotically approaching an upper limit [12]. For some *discount factor* d, the cost is defined as

$$\text{discounted total completion time} = \sum_{j \in J} w_j \left(1 - e^{-dc_j}\right) \tag{2.1}$$

2.5 Scheduling algorithms

As mentioned in section 2.1.1, the default Jenkins scheduler maintains a single queue and assigns each job to executor to agent when it is about to leave the queue. Based on this, the algorithms is divided into two groups based on whether they maintain the same structure as the default Jenkins scheduler or if they maintain separate queues for each executor.

2.5.1 Single queue algorithms

One of the simplest scheduling algorithms available, and the default used in Jenkins, is *First In-First Out* (FIFO). All jobs are placed in a single queue on arrival and

processed in the same order that they arrived in as resources become available. The opposite of FIFO is *Last In-First Out* (LIFO), treating the queue as a stack.

In the *Shortest Processing Time* (SPT) algorithm, the scheduler uses knowledge about the processing time of the job, prioritizing jobs with shorter expected processing times. The opposite of SPT is called *Longest Processing Time* (LPT) and gives higher priority to jobs with longer processing times. [12]

FIFO, LPT and SPT are all examples of unweighted algorithms where all jobs are considered equally important, and their priority is determined solely by the characteristics of the job, such as release time or processing time. When introducing weights associated with jobs, natural extensions include algorithms such as *Weighted Shortest Processing Time* (WSPT) and *Weighted Longest Processing Time* (WLPT) where jobs are processed based the ratio w_j/p_j where, again, w_j is the weight of the job and p_j is the processing time [12]. WSPT and WLPT are sometimes also referred to as *Highest Density First* (HDF) and *Lowest Density First* (LDF) respectively.

2.5.2 Separate queues

An alternative approach is to assign each job to an executor when it arrives to the scheduler, maintaining separate queues for each executor. In two recent papers [15][16], Gupta et al. introduce a new algorithm for online scheduling of jobs with stochastic processing times, on *unrelated machines*. Unrelated machine means the processing time of each job is given by p_{ij} (or P_{ij} in the stochastic case), denoting the processing time of job j when running on machine i. The algorithm aims to minimize the expected value of the total weighted completion time, $\mathbb{E}\left[\sum_j w_j C_j\right]$, and comes with a proven performance guarantee.

The algorithm is first stated for the deterministic case, with a known processing time, p_{ij} for each combination of job and machine. For a job j, the cost of assigning j to a machine i is defined as

$$\text{cost}(j \to i) = w_j \left(\left(1 + \frac{1}{c}\right) r_{ij} + p_{ij} + \sum_{k \in U_i(r_j), \frac{w_k}{p_{ik}} < \frac{w_j}{p_{ij}}} p_{ik} \right) + \sum_{k \in U_i(r_j),, \frac{w_k}{p_{ik}} \geq \frac{w_j}{p_{ij}}} w_k p_{ij} \tag{2.2}$$

where $U_i(t)$ denotes the set of jobs assigned to i at time t that have not yet been started. From here, the process to assign an incoming job j to a machine i can be described in two steps.

1. Assign j to i minimizing $\text{cost}(j \to i)$.
2. Within the queue for each machine, schedule jobs according to WSPT.

For the case with stochastic processing times, the algorithm is modified in two ways. The cost function and WSPT algorithm uses $p_{ij} = \mathbb{E}\left[P_{ij}\right]$ as a drop-in replacement for the deterministic processing time when assigning jobs to machines and sorting the queues.

Additionally, a modified starting time $S_{i,\ell}$, denotes the starting time of the ℓ:th job on i and is defined by

$$S_{i,\ell} = \max \{s_{i,\ell}, S_{i,\ell-1} + P_{i,\ell-1}\}$$

where $s_{i,\ell}$ denotes the starting time of the same job in a deterministic setting.

2.5.3 Avoiding starvation

Starvation occurs when a job is continuously denied access to the resources it needs [13]. In the scheduling scenario, this could occur, for example, if a single job is considered bad for the performance of the scheduler in a given metric. A scheduler using the SPT rule would constantly prioritize jobs with shorter processing times, causing longer jobs to get stuck in the queue as long as there are new jobs coming in with shorter processing times. The inverse would happen with an LPT scheduler, where job with shorter processing times would be denied access while there are longer jobs available.

Avoiding starvation can be done in different ways. By using a randomized algorithm, every job has a non-zero chance of getting access to resources that become available and the time spent in the queue can be modeled as a random variable with an expected value. Another possibility is to add an *aging factor* where the priority of jobs in the queue is slowly increased. Eventually this would mean that longer jobs that have waited for a long time will be prioritized before shorter jobs that just arrived into the queue [17]. For an algorithm sorting jobs based on a function sort-key(j), with lower values meaning closer to the front of the queue, this gives

$$\text{sort-key}_{\text{aging}}(j) = \text{sort-key}(j) - aq_j$$

where a is the aging factor and q_j is the time j has spent waiting in the queue.

3

Methods

In this section we discuss our approach to the problem and the details of the implemented solutions. Implementing all of the scheduling algorithms mentioned in section 2.5 would require changes to the Jenkins core, which was ruled out of scope for this project. Instead, the research questions from section 1.2 were mainly investigated by implementing a scheduling simulator, modeling a real Jenkins environment. In the simulator, different scheduling algorithms were evaluated using both sets of individual jobs and sets of pipelines, built to model pipelines running in the real Jenkins environment.

To verify that the simulator gave an accurate model of a real environment, a subset of the algorithms were tested on a real Jenkins server, implemented through a Jenkins plugin, described in section 3.3.1. The results of these tests were then compared to results from the simulator.

3.1 Testing in the simulator

As mentioned in the introduction to this chapter, the simulator was used to evaluate all algorithms considered in this project. The simulations were performed based on the research questions from section 1.2.

3.1.1 Comparing different algorithms

Questions 1 and 2 in section 1.2 are dependent on how system performance is measured. As described in section 2.4, there are several ways to do this, including total completion time, makespan etc. During the project, none of these metrics have been selected as the single most relevant, instead the different algorithms have been evaluated on each of the metrics in order to give a more complete picture of the trade-offs being made when choosing between the algorithms.

During simulations, the simulator recorded the following data about individual jobs.

- Average time spent waiting in the queue.
- Maximum time any job spent waiting in the queue.
- Total weighted completion time.

For complete pipelines, the following data was measured and reported.

- Average time from release to completion.
- Maximum time from release to completion for any pipeline.
- *Discounted pipeline lifetime*

The *discounted pipeline lifetime* is a modified version of the discounted total weighted completion time defined in equation (2.1) in section 2.4.2 and is defined as

$$\text{discounted pipeline lifetime} = \sum_{p \in P} w_p \left(1 - e^{-d(c_p - r_p)}\right)$$

Like the discounted total weighted completion time, the discounted pipeline lifetime reflects that the cost per time unit of waiting for a pipeline to complete may decrease over time. The main difference is that the exponent is based on the release time, as well as the completion time, making the cost function independent of what time the pipeline was started on the Jenkins server. A graph of the cost according to this model can be seen in figure 3.1.

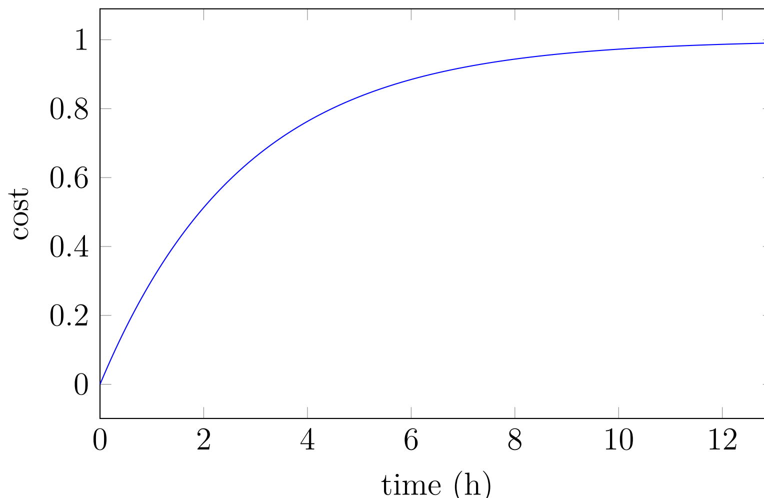


Figure 3.1: The cost of a pipeline according to the discounted pipeline lifetime metric.

In addition to the algorithms described in section 2.5, the simulations also included two algorithms based on the pipeline structure of the Jenkins jobs. The first can be described as *Shortest Maximum Remaining Time in Stage* (SMRTS). For a job in the queue, this algorithm looks at the maximum remaining (expected) processing time of any job belonging to the same pipeline stage, including jobs that are currently being processed. Priority is then given to jobs belonging to the pipeline stage with the smallest maximum remaining time. Jobs belonging to the same stage are sorted according to LPT. Mathematically, the sorting key is given by

$$\text{sort-key}(j) = \max_{j' \in S(j)} \left(\mathbb{E}\left[P_{j'}\right] - t_{j'}\right)$$

where $S(j)$ denotes the set of all jobs in the same pipeline stage as j and t_j is the elapsed time since processing of j began (0 if j has not yet been started).

The second algorithm is similar to the SMRTS algorithm, but instead looks at the sum of the remaining (expected) processing time of all jobs in the same pipeline stage. This can be described a *Shortest Total Remaining Time in Stage* (STRTS) where the sorting key is given by

$$\text{sort-key}(j) \sum_{j' \in S(j)} \left(\mathbb{E}\left[P_{j'}\right] - t_{j'} \right)$$

In order to evaluate the effects of starvation mitigation, the SPT, SMRTS and STRTS algorithms were also evaluated with an aging factor.

3.1.2 Expected vs. actual processing times

Simulating jobs based on historical data makes it possible to provide the scheduler with real values for the processing time of each job instead of just the average value of previous executions. This makes it possible to answer question 3 in section 1.2 by running the simulator with access to the actual processing time of each job instead of restricting it to looking at average processing times of past executions.

3.1.3 Effects of adding more agents

The need for scheduling arises from limitations in resources and the existence of a queue. This suggests that the answer to question 1 in section 1.2 depends on the number of agents and executors available. With more available resources, the difference in performance between different algorithms should decrease. To confirm this, identical workloads were simulated with different numbers of agents available.

3.1.4 Simulated jobs and environment

Each algorithm was evaluated on a number of pipeline sets, created from real world data, where each set contained all pipelines processed during a 24 hour period in the real Jenkins environment. Details about how the simulator generates jobs and pipelines from log data can be found in section 3.2.1.

The real Jenkins server and the available agents handle more jobs than the ones simulated during the project. This means a simulation with the same number of agents as the real world environment would have access to more resources than is used for processing the pipelines in the real environment. Therefore, the simulations used a smaller number of agents to ensure the scheduling policy affects the result.

To create an environment with unrelated machines, each agent is created with fixed processing speed, s_i, generated randomly, at the start of each simulation. This processing speed is known to the controller while scheduling and makes it possible to get different values for the processing time of a job depending on which machine the job is assigned to, according to equation (3.1). As can be seen in the equation, $s_i > 1$ corresponds to a machine processing jobs faster than average while $s < 1$ means the machine processes jobs slower than average.

$$P_{ij} = \frac{P_j}{s_i} \tag{3.1}$$

All random elements in the environment, such as the processing speeds of individual agents, were generated using a seeded random generator, ensuring that the environment is the same for all algorithms [18].

3.2 Simulator implementation details

The simulator was implemented in Python and consists of a number of modules, modeling the behavior of a real Jenkins environment. The model is centered around a controller, along with a set of agents, each running a number of executors. Jobs, and complete pipelines, can be submitted to the controller from a job creation module using different strategies. The controller can be configured to use different scheduling algorithms described in section 2.5, such as SPT, LPT or FIFO. Data is recorded throughout the simulation and compiled to a summary at the end of each run. A graphical overview of the simulator can be seen in figure 3.2.

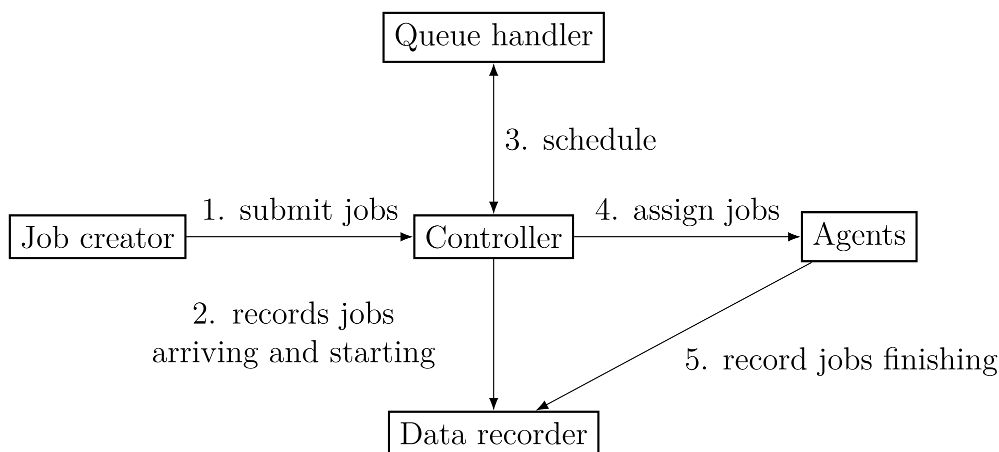


Figure 3.2: Schematic overview of the simulator. The job creating module submits jobs and pipelines (1) to the controller. The controller records the arrival (2) and schedules the job (3) using the queue handler. Once a job is due to be processed, the controller sends the job to an agent (4) where the job is processed. When the job is finished, the agent records the finishing time and becomes available to process another job.

3.2.1 Job generation

As mentioned in section 2.2.2, execution data is stored in an Elastic database. When creating a pipeline, the simulator fetches the real world creation time of the pipeline along with the processing time of the individual jobs. Using static rules, based on the known structure of the pipeline, the jobs are organized in stages that together form the complete pipeline.

During simulation, each pipeline is submitted to the controller by the job creation module (see figure 3.2) at the same time as the real world pipeline was submitted to the Jenkins server. Since the order in which the jobs are processed depends on the scheduling algorithm used by the scheduler, individual jobs are not assigned release times during the pipeline creation process. Instead, each job becomes available for processing when the previous stages of the pipeline are finished during the simulation.

3.2.2 Limitations

When building a simulator, it is impossible to get an absolute copy of the real world. The simulator must therefore make some simplifications in its model of the environment.

The real world agents have multiple executors on them, allowing them to better utilize multi core CPUs when executing jobs. This does however mean that parts of the execution of a job could suffer slowdowns when executed on an agent that is executing another job simultaneously. For example, two jobs started very close to each other in time on the same agent could be fetching code from the repository at the same time, sharing the available network bandwidth. The log data used in the simulator does not include information about simultaneous executions on the same agent, so modeling this slowdown was deemed out of scope for the project. Therefore, the simulator does not consider slowdowns due to jobs being processed in parallel on the same agent. Instead, each machine has a static processing speed, affecting all jobs, as described in section 3.1.4.

3.3 Jenkins implementation

For comparability to the simulator, the algorithms tested were SPT, LPT and FIFO. Each of the algorithms was evaluated using the makespan, average waiting time and maximum waiting time. These results were then compared with results of similar tests from the simulator. The implementation consists of three main parts, a customized scheduler, a module to retrieve data about previous executions and a script to submit jobs to the queue. An overview of the system setup can be found in figure 3.3.

The tests were carried out on a Jenkins server with access to a number of agents. On this server jobs were created, simulating real world jobs. The jobs consisted of a simple Python script using the sleep function to occupy the executor for set amount of time. This resulted in a queue for the builders that was controlled by the Priority Sorter plugin. The plugin would sort according to what algorithm was active. The algorithms available were SPT, LPT, FIFO and SPT with aging.

3.3.1 Priority Sorter plugin

In order to evaluate the scheduling algorithms mentioned in section 3.1.1 on a real Jenkins server, a customized scheduler was implemented as a plugin for Jenkins.

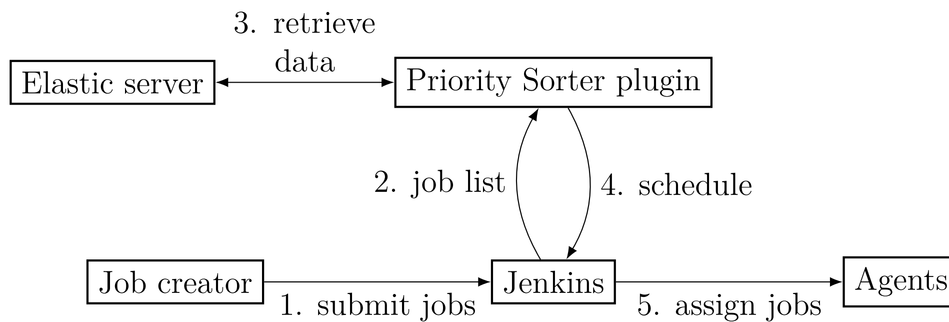


Figure 3.3: Schematic overview of the plugin. The job creator submits jobs to the controller (1). The Jenkins controller sends the Priority Sorter the current list of jobs (2). Priority Sorter retrieves data stored in the Elastic database for decision making (3), then returns the schedule for that list of jobs (4). Once a job is due to be processed, the controller sends the job to an agent where the job is processed (5).

This plugin is based on a previously existing plugin named *Priority Sorter* that enables static priorities for different jobs [19]. These priorities are then used for sorting the queue. After modifying the plugin, these priorities are instead set dynamically each time a job is submitted to the controller, based on data from previous builds, fetched from the Elastic server.

The priority value, s, can range from 1 to 150, where low numbers represent the front of the queue. When a new job is submitted to the queue, it's priority is set based on data from previous executions. The job is then added to the queue and the queue is sorted according to the active sorting rule, determined by the active scheduling algorithm. If aging is active, the sorting value is modified as described in section 2.5.3.

For SPT the expected processing time was fetched from the Elastic database using parameters from the incoming job. The expected processing time was then scaled such that $s \in [1, 150]$. For LPT, the priority was calculated the in a similar way, but the priority was set to $s' = 150 - s$. When using FIFO, all priorities were set to $s = 50$. If jobs in the queue had the same priority, the front of the queue would be determined through their time in the queue. If the processing time for a job can not be found on the Elastic server while using LPT or SPT, the job gets assigned a pre-defined priority.

3.3.2 Data retrieval

As mentioned in section 2.2.2, execution data is logged to an Elastic server that stores the information in its database. This database is then connected to and data can be retrieved from the server to the client. This data includes several parameters, such as processing time, agent hostname, job name, etc. Each time an incoming job gets presented to the Priority Sorter plugin, the data retrieval module will be called with parameters from the incoming job and return an expected processing time of that job. Since the process time of the incoming job is non-deterministic the returned process time from the Elastic server will be the average of previous jobs

with similar characteristics.

3.3.3 Job dispatcher

For creation of jobs, a job dispatcher was implemented in Python. This job dispatcher script ran on the Jenkins server in order to quickly create jobs for the scheduler. The dispatcher picked a random job from a pool and sent the selected job into Jenkins using an HTTP POST request. The job pool consists of 23 commonly executed jobs on the live Jenkins system. The random generator used when picking jobs used a seed to always ensure [18] that the same random set of jobs were presented to each algorithm. Jobs were chosen and sent each second for 360 seconds. After this amount of time the dispatcher would wait until all executors on the Jenkins server were finished and then start a new iteration of jobs.

4

Results

In this chapter we describe the results from both simulations and test runs on a real Jenkins server. These results are presented as boxplots, showing the spread of the values with the median value highlighted. All boxplots in this chapter are constructed with a whisker limit of 1.5 times the interquartile range (IQR), meaning any value that is more than 1.5 IQR above the upper quartile or 1.5 IQR below the lower quartile are marked individually as outliers [20]. To mark the outliers, a circle (○) is used. For more details about how to read the box plots, see appendix A.

4.1 Simulating real jobs and pipelines

In this section, we show the results from the simulator. Sections 4.1.1 to 4.1.4 shows how various algorithms perform under different metrics while sections 4.1.5 and 4.1.6 show the effects of having more accurate data available or adding more agents to the environment.

4.1.1 Comparing different algorithms

Figure 4.1 shows the makespan for the evaluated algorithms. The lowest value is generally achieved by the LPT algorithm with a median makespan of 83 315 s (23 h, 8 min, 35 s). The LPT algorithm also has the smallest IQR, indicating a large proportion of the values being close to the median. The greedy algorithm by Gupta et al. shows the highest values, with a median makespan of 102 020 s seconds (28 h, 20 min, 20 s). Makespan values below 24 h (86 400 s) show that some of the simulated days do not have jobs coming in near the end of the 24 hour period.

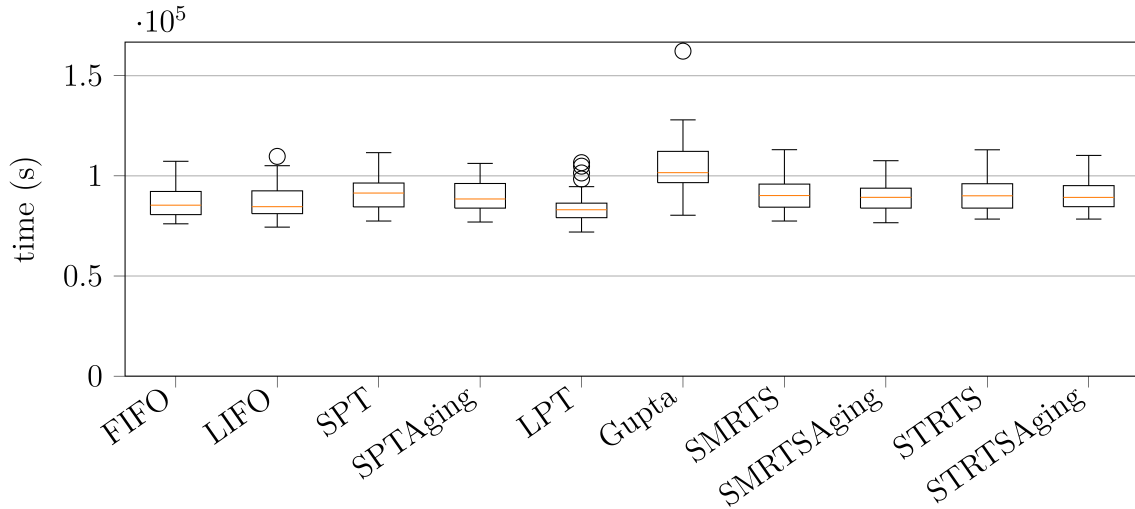


Figure 4.1: Box plot summarizing the makespan for sets of pipelines submitted over a time period of 24 hours. The lowest values are generally achieved by the LPT scheduler, with the greedy algorithm from Gupta et al. having the highest values. Values more than $1.5 \cdot \text{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

Figures 4.2 and 4.3 summarize the values for average lifetime of a pipeline and maximum lifetime of a pipeline respectively. Figure 4.2 shows that the lowest average lifetime is generally achieved by the algorithms based on complete stages rather than individual jobs with STRTS producing a median value of 4468 s (1 h, 14 min, 28 s) and SMRTS having a median of 5032 s (1 h, 23 min, 52 s). The longest average lifetimes are produced by the LPT, with a median of 17 901 s (4 h, 58 min, 21 s).

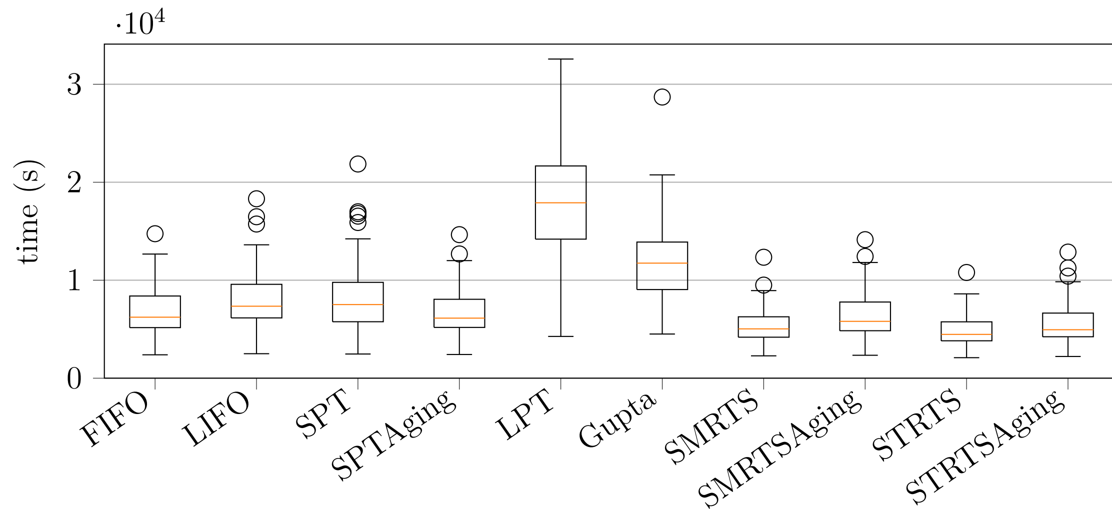


Figure 4.2: Average lifetime of a pipeline when scheduling based on different algorithms. The lowest values are generally achieved when scheduling based on complete stages rather than individual jobs, with the lowest median achieved by the STRTS algorithm. Values more than $1.5 \cdot \text{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

Figure 4.3 shows that the maximum lifetime observed for any pipeline is generally higher for all algorithms based on the characteristics of jobs or stages than for the default FIFO scheduler, indicating a risk of starvation. The median value for the FIFO scheduler is the lowest at 33 898 s (9 h, 24 min, 58 s) while the Gupta algorithm has the highest median value at 57 371 s (15 h, 56 min, 11 s). Adding an aging factor to these algorithms helps remediate the problem of starvation while still achieving lower values for the average pipeline lifetime.

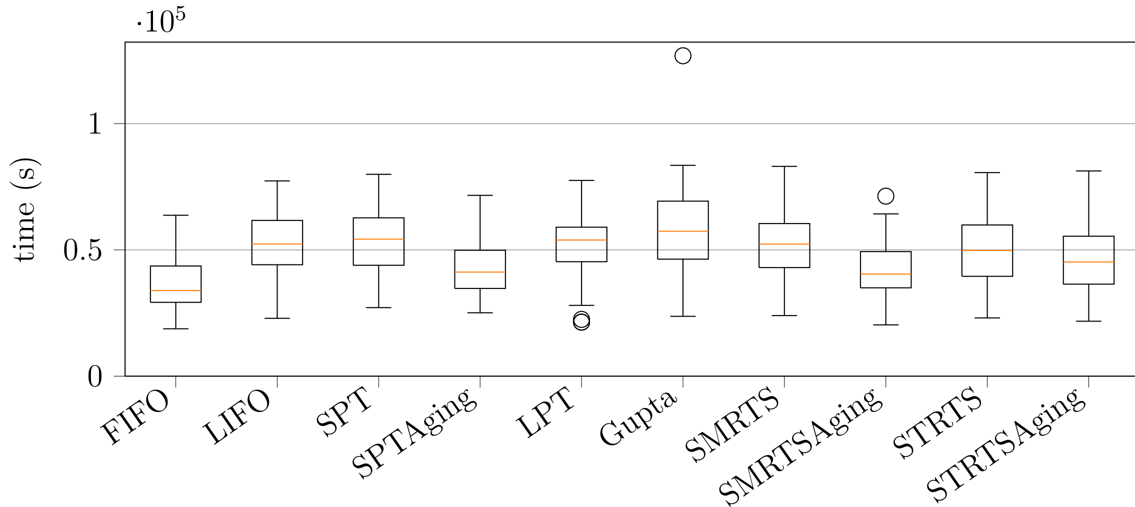


Figure 4.3: Maximum lifetime of a pipeline when scheduling based on different algorithms. The lowest values are achieved by the FIFO scheduler and the Gupta algorithm has the highest values. Adding an aging factor reduces the maximum lifetime for all algorithms where it is applied. Values more than $1.5 \cdot \mathrm{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

Just like the average pipeline lifetime, the discounted pipeline completion cost, shown in figure 4.4, is lowest for the SMRTS and STRTS algorithms. In the plot, generated with a discount factor $d = 0.001$, SMRTS has a median cost of 173 and STRTS a median of 171. In contrast to the linear cost model for the pipeline lifetime, the LIFO scheduler performs relatively good with a median cost of 175 and the LPT scheduler displays performance on similar levels with the other algorithms. More discussion about this can be found in section 5.1.1.

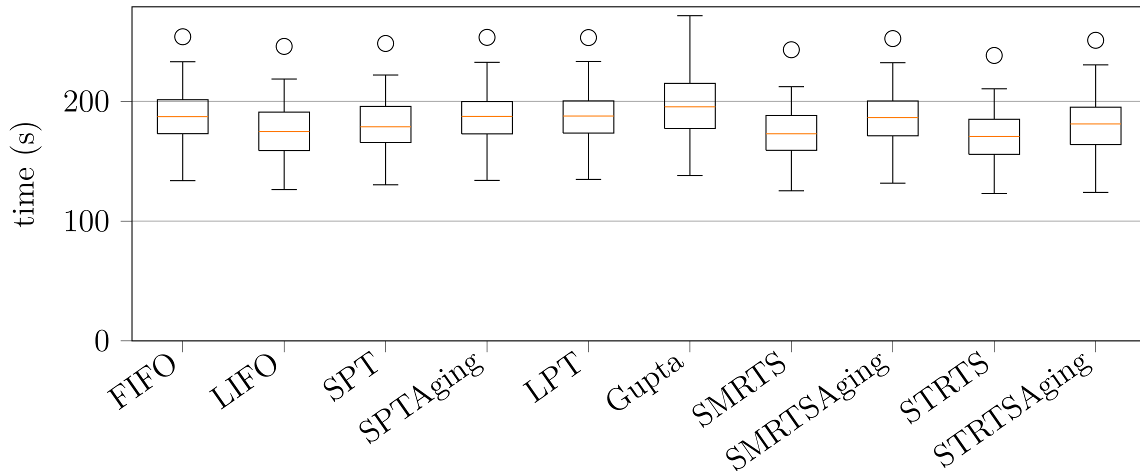


Figure 4.4: Discounted pipeline cost for pipelines scheduled using different algorithms. The lowest values are achieved by the SMRTS and STRTS algorithms. Values more than $1.5 \cdot \mathrm{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

4.1.2 Completion times for individual jobs

Measuring the total weighted completion time, with results shown in figure 4.5, gives relatively small differences between the algorithms. The LPT algorithm has the highest values with a median of 8.35×10^7 s while all other medians are between 7.39×10^7 s (SPT) and 7.72×10^7 s (the greedy algorithm by Gupta et al.) During simulations, the weight of all jobs were set to $w_j = 1$.

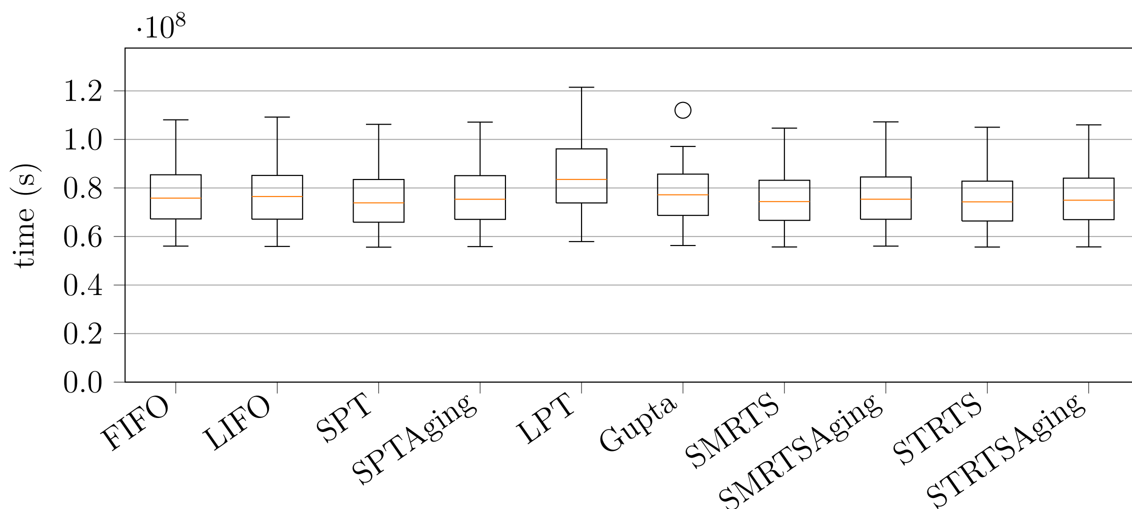


Figure 4.5: Total weighted completion time for individual jobs. The lowest value is achieved by SPT while LPT has the highest values. Values more than $1.5 \cdot \mathrm{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

4.1.3 Queue times for individual jobs

Figures 4.6 and 4.7 show the average and maximum time spent in the queue by jobs waiting for a free executor. The lowest average queuing time, shown in figure 4.6 is generally achieved by the SPT scheduler with a median value of 2118 s (35 min, 18 s) while LPT has the longest queuing times with a median of 7230 s (2 h, 0 min, 30 s).

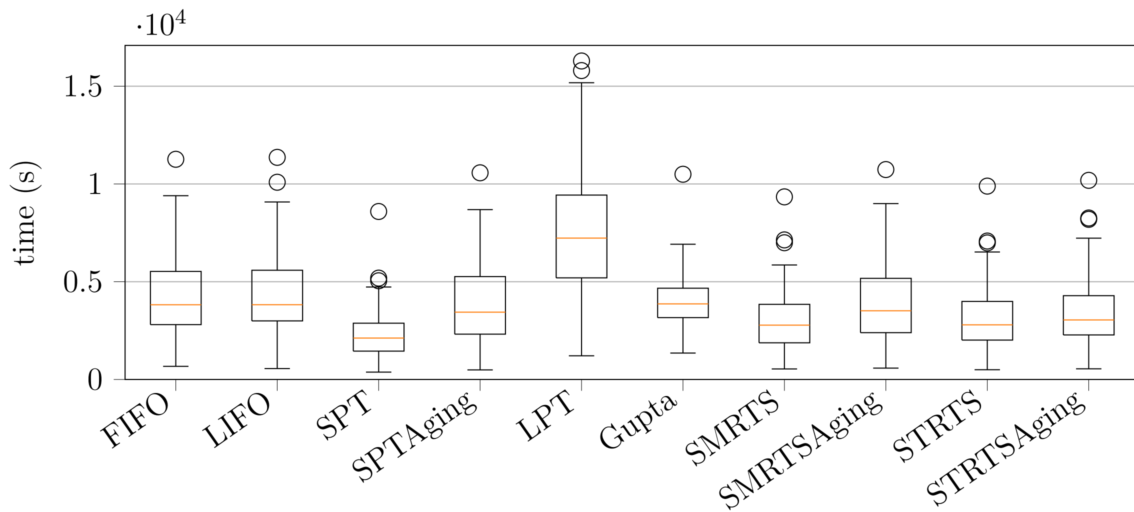


Figure 4.6: Average time spent in the queue by individual jobs. SPT achieves the lowest values while LPT gives the highest values. Values more than $1.5 \cdot \text{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

The maximum time spent in the queue by any job is shown in figure 4.7. The plot clearly shows how algorithms based on the processing time suffer from starvation risk with high values for the maximum queuing time. The lowest values are achieved by the FIFO algorithm with a median value of 13 800 s (3 h, 50 min). LPT has the highest median value at 47 646 s (13 h, 14 min, 6 s) while the Gupta algorithm has slightly higher upper quartile than LPT and a single outlier high above the other values. Adding an aging factor can significantly reduce the maximum queuing time, as can be seen by looking at the SPT algorithm with and without aging.

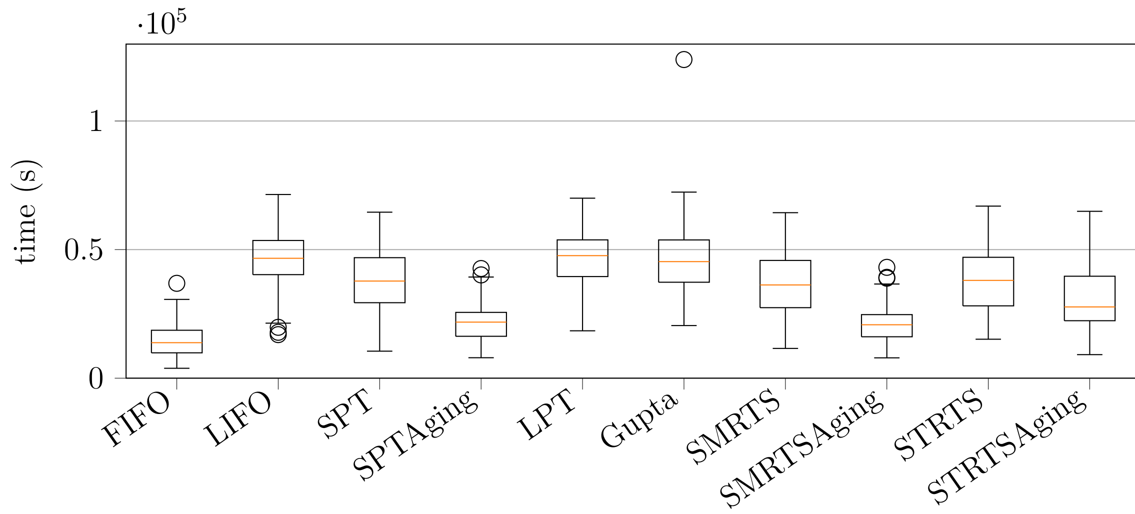


Figure 4.7: Maximum time spent in the queue by individual jobs. FIFO generally gives the lowest values. Adding aging factors reduce the values for algorithms where there is a risk for starvation. Values more than $1.5 \cdot \text{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

4.1.4 Summary of algorithm performance

Table 4.1 shows a summary of the performance of the evaluated algorithms under different metrics with a categorical scale of low, medium, high to indicate the values measured.

Table 4.1: Summary of the performance of different algorithms under different metrics.

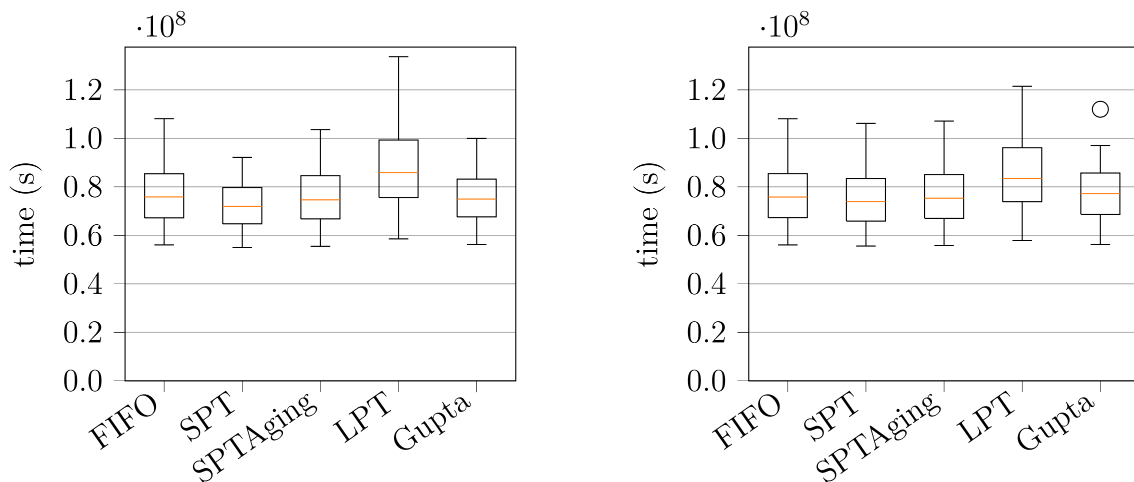
	Pipeline lifetime	Discounted pipeline cost	Total weighted completion time	Queue time
FIFO	Avg: medium Max: low	high	medium	Avg: medium Max: low
LIFO	Avg: medium Max: high	low	medium	Avg: medium Max: high
SPT	Avg: medium Max: high	medium	medium	Avg: low Max: high
SPT Aging	Avg: medium Max: medium	medium	medium	Avg: medium Max: low
LPT	Avg: high Max: high	medium	high	Avg: high Max: high
Gupta	Avg: high Max: high	high	medium	Avg: medium Max: high
SMRTS	Avg: low Max: high	low	medium	Avg: low Max: medium
SMRTS Aging	Avg: low Max: medium	medium	medium	Avg: medium Max: low
STRTS	Avg: low Max: high	low	medium	Avg: medium Max: medium
STRTS Aging	Avg: low Max: medium	medium	medium	Avg: low Max: medium

4.1.5 Expected vs. actual processing times

Figure 4.8 shows the performance differences between a scheduler with access to the actual processing time of each job (4.8a) and the expected value (4.8b), measured through the total weighted completion time.

The results indicate that the effect of each algorithm is amplified when the scheduler has access to the exact processing time of each job instead of using an expected value. Looking at the SPT algorithm, there is an increase in performance when using expected values for the job processing times (figure 4.8b) and this difference increases when the scheduler has access to the actual processing time of each job (figure 4.8a). This can be compared with the LPT algorithm, which does not optimize on this metric. When using expected values (figure 4.8b, the performance is slightly worse

than FIFO and this difference increases when giving the scheduler access to the actual processing time of each job (figure 4.8a).



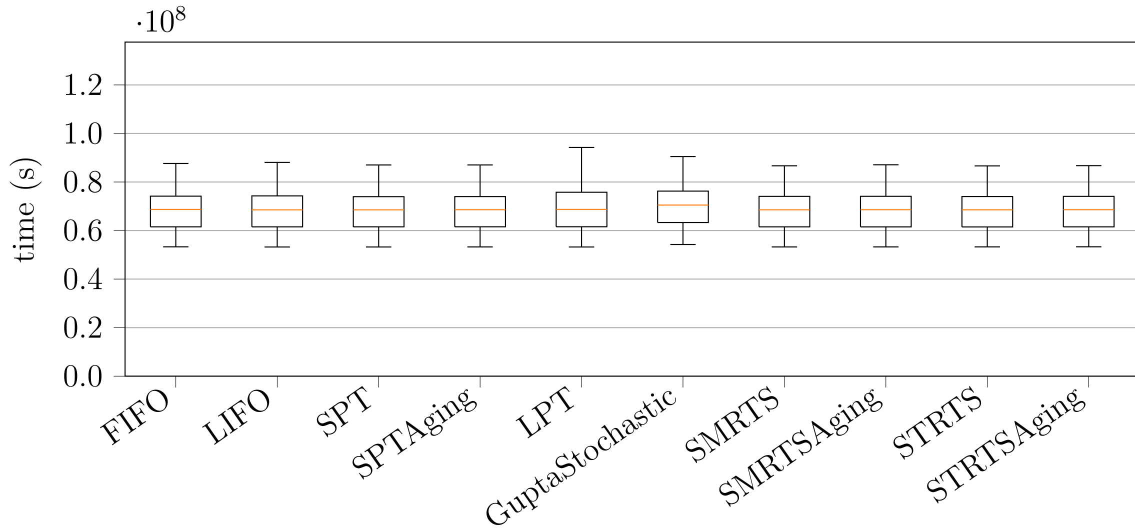
(a) Scheduler with access to the actual processing time.

(b) Scheduler with access to an expected value.

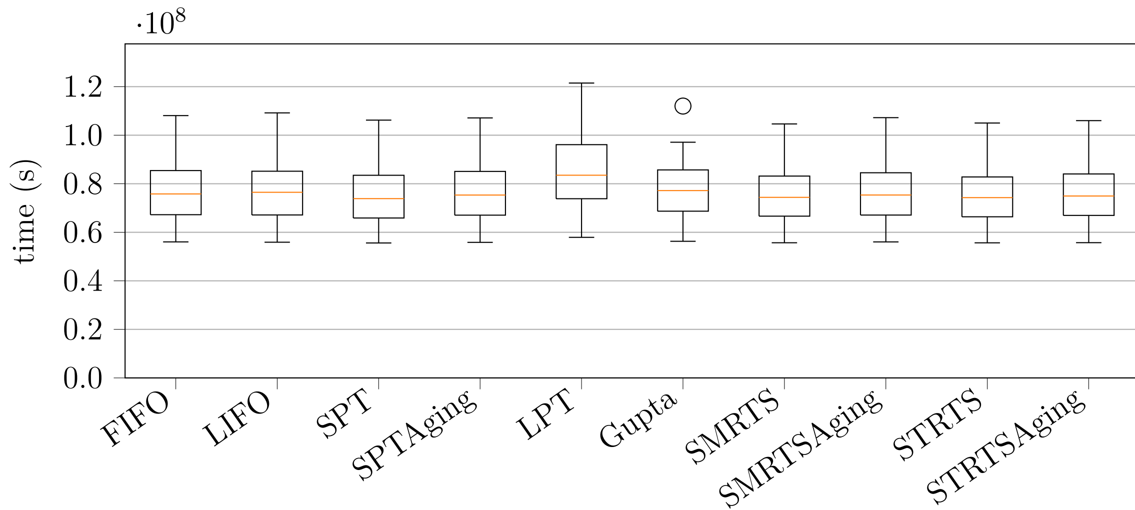
Figure 4.8: Total weighted completion times for schedulers with access to (a) the actual processing time of each job or (b) an expected value for the processing time.

4.1.6 Adding more agents

Section 4.1.6 clearly shows how adding more executors removes the performance differences between different scheduling algorithms. The two plots show the same simulated pipeline sets with 50 executors (4.9a) and 30 executors (4.9b). In addition to all values being lower, the spread of the values is lower as well. Median values with 50 executors range between 68 532 351 s (SPT) and 70 502 362 s (greedy algorithm from Gupta et al.) while the values with 30 executors range between 73 872 610 s (SPT) and 83 510 670 s (LPT).



(a) Total weighted completion time with 50 execution slots.



(b) Total weighted completion time with 30 execution slots (same as figure 4.5). Values more than $1.5 \cdot \text{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

Figure 4.9: Total weighted completion time for simulation environments with 50 executors (a) and 30 executors (b).

4.2 Simulating individual jobs

Figures 4.10 to 4.12 summarize data from initial simulations using 10 randomized sets of jobs.

4.2.1 Makespan

Figure 4.10 shows the spread of the makespan for the simulations described in section 4.2. Among the evaluated algorithms, LPT has the lowest median makespan as well as the lowest maximum and minimum recorded value while FIFO and SPT show very similar results at slightly higher values.

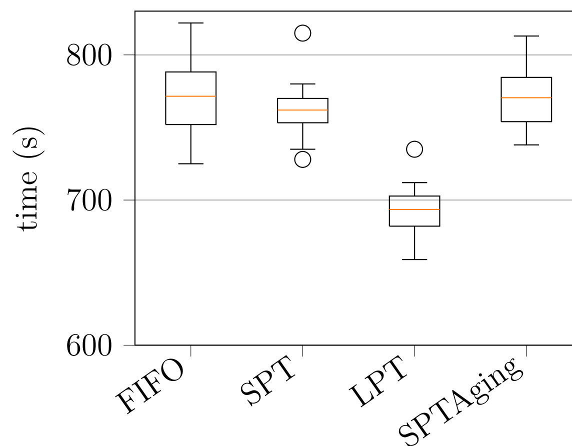


Figure 4.10: Box plot summarizing the makespan of 10 randomized sets of jobs scheduled using FIFO, SPT and LPT. Values more than $1.5 \cdot \mathrm{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

4.2.2 Waiting times

Figures 4.11 to 4.12 summarize the average and maximum amount of time a job spends waiting in the queue after being submitted to the controller for execution. The average waiting time, displayed in figure 4.11, is generally lowest for SPT, while LPT produces the highest values with FIFO in between.

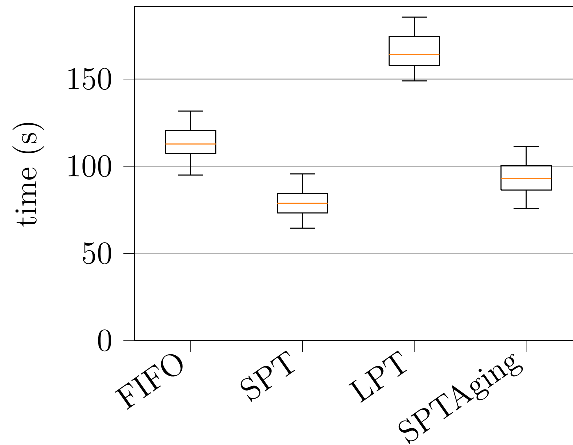


Figure 4.11: Box plot summarizing the average time a job spends in the queue for 10 randomized sets of jobs scheduled using FIFO, SPT and LPT.

Figure 4.12 shows that FIFO gives the lowest values for maximum time spent in the queue by a job with LPT showing the highest values, slightly worse than SPT.

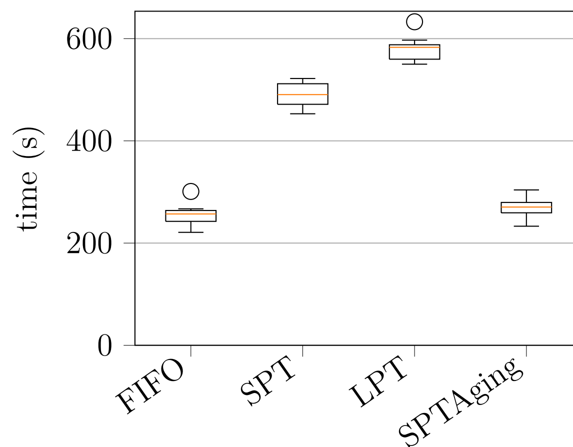


Figure 4.12: Box plot summarizing the maximum time any job spent waiting in the queue for 10 randomized sets of jobs scheduled using FIFO, SPT and LPT. Values more than $1.5 \cdot \text{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

4.3 Jenkins results

The following data was gathered from the actual Jenkins server. The metrics in this section is makespan, maximum waiting time and the average waiting time of the jobs.

4.3.1 Makespan

Figure 4.13 shows the results when comparing the makespan between the algorithms.

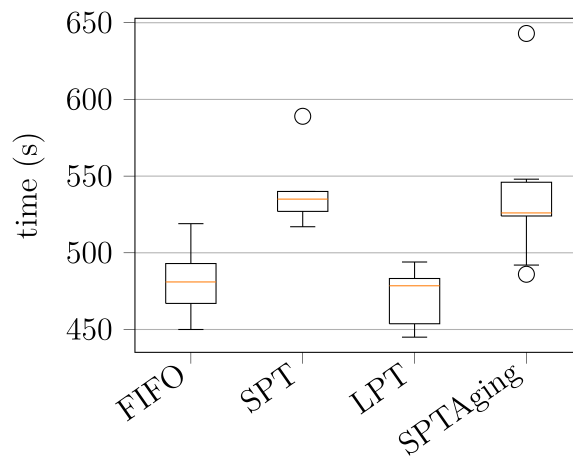


Figure 4.13: Box plot summarizing the makespan of 10 randomized sets of jobs scheduled using FIFO, SPT, LPT, SPT with aging and LPT with aging on the Jenkins server. The figure shows that LPT and FIFO generally gives the lowest values for the makespan. Values more than $1.5 \cdot \text{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

4.3.2 Average waiting times

Figure 4.14 and figure 4.15 shows average waiting time and maximum waiting time for four different algorithms respectively.

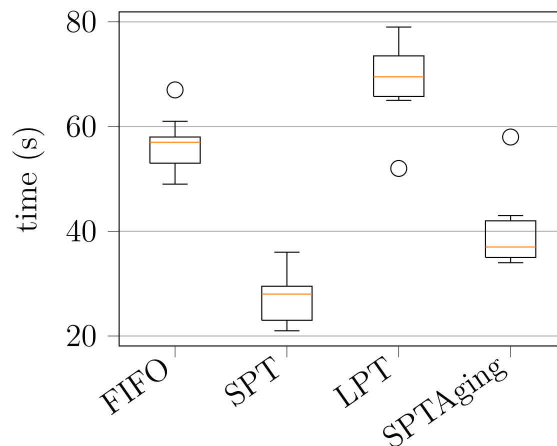


Figure 4.14: Box plot summarizing the Average waiting time of 10 randomized sets of jobs scheduled using FIFO, SPT, LPT and SPT with aging on the Jenkins server. It is clear that SPT has the lowest of these, which is expected. Values more than $1.5 \cdot \text{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

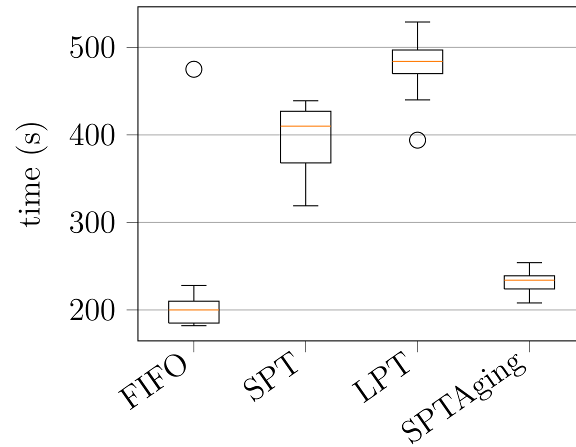


Figure 4.15: Box plot summarizing the Maximum waiting time of 10 randomized sets of jobs scheduled using FIFO, SPT, LPT and SPT with aging. It is clear that LPT has the highest of these, which is expected since all the short jobs that enters the queue early will be in queue for the entire process. Values more than $1.5 \cdot \mathrm{IQR}$ above the upper quartile or below the lower quartile are marked with a circle ($\circ$).

5
Conclusion

In this chapter, we discuss the results and conclude our research. First we discuss answers to the research questions in section 1.2 and give some justification for the metrics used in the evaluation. We then move on to discuss observations made during the project and suggestions for future research before wrapping up with a conclusion of the most important takeaways from the project.

5.1 Discussion

Looking at research question 1, "*Can the performance of the CI pipeline be improved by using scheduling algorithms utilizing existing knowledge of the jobs and machines available?*", the results presented in chapter 4 indicate that the answer is **yes**. There are several cost models where the default FIFO algorithm used by Jenkins is not optimal. However, these improvements in performance are reduced if there are enough agents available to reduce the queuing time altogether, as shown in section 4.1.6.

Question 2, "*How do different scheduling algorithms perform compared to each other?*" depends heavily on the choice of performance metric, and it is therefore hard to give a short answer to this question. Section 5.1.3 gives a more detailed discussion about the differences between the algorithms.

Finally, the results presented in section 4.1.5 give an answer to research question 3, "*How do differences in data available to the scheduler affect its performance?*". The plots in figure 4.8 show that while most algorithms offer a performance increase, under some metric, when provided with an expected value $\mathbb{E}[P_j]$ for jobs, the effect of switching algorithms is amplified when deterministic processing times are used. The SPT scheduler, which produces the lowest values for the total weighted completion times in figure 4.8b, has even better performance in the deterministic case.

5.1.1 Relevance of the performance metrics

The results from section 4.1 show how different performance metrics will give very different results when deciding which scheduling algorithms is the best. In this section, we will give a short discussion about the relevance for each of them when evaluating schedulers in the given Jenkins environment.

The most intuitive metrics for the given environment are the linear models for aver-

age and maximum pipeline lifetime. Measuring the time it takes between submission of a pipeline to the time when a developer can receive feedback, they should not need any further justification. Looking instead at the discounted pipeline completion time, this model includes a decrease in cost per time unit during the processing of a pipeline. The main argument for using this model would be that a longer waiting time would increase the likelihood that the developer can work on something else while waiting for the pipeline to finish, thereby reducing the cost of keeping the pipeline in the system.

Looking at the total weighted completion time and waiting times for individual jobs, the relevance of these metrics depends heavily on other mechanisms in the system. A common philosophy in software testing is to *fail fast*. Part of this philosophy is the practice to give up on a test suite as soon as a build or a test fails in order to provide feedback to the developer as quickly as possible [21]. Unless there is a known correlation between processing time and the probability of finding an error during a job, this would motivate using metrics based on individual jobs when evaluating performance, as finishing more jobs early will increase the probability of finding bugs early. If the pipeline is interrupted, and any jobs still in the queue are aborted when a build fails, this could mean quicker feedback for the developers.

Looking finally at the makespan, this metric may not be very useful in the given setting. Although the simulations are performed with job sets based on 24 h intervals, the real Jenkins server acts on a continuous stream of incoming jobs that makes it difficult to define a relevant makespan. Additionally, figure 4.1 shows that most algorithms achieve a similar makespan. This can be explained by the fact that the submission period (24 h) is significantly longer than the average lifetime of a pipeline since the makespan has a lower bound at the release time of the last job submitted. Together, these arguments suggest that the makespan is the least relevant of the performance metrics used in this project.

5.1.2 Correlation between metrics

Figures 4.5 and 4.6 show that the total weighted completion time and the average time a job spends in the queue give similar results when evaluating algorithms. For a job j, released at r_j, with processing time p_j, which spend q_j seconds in the queue, the completion time c_j is given by

$$c_j = r_j + q_j + p_j$$

and the total weighted completion time for the job set J is given by

$$\sum_{j \in J} w_j c_j = \sum_{j \in J} w_j r_j + \sum_{j \in J} w_j q_j + \sum_{j \in J} w_j p_j \tag{5.1}$$

In the case of individual jobs, r_j is independent of the scheduling algorithm and equation (5.1) can be rewritten as

$$\sum_{j \in J} w_j c_j = k' + \sum_{j \in J} w_j q_j + \sum_{j \in J} w_j p_j \tag{5.2}$$

In the special case of identical machines, p_j is also independent of the scheduling algorithm, meaning equation (5.2) can be simplified to

$$\sum_{j \in J} w_j c_j = k + \sum_{j \in J} w_j q_j$$

where

$$k = \sum_{j \in J} w_j r_j + \sum_{j \in J} w_j p_j$$

As mentioned in section 3.1.4, $w_j = 1$ during all simulations in this project. The average queue time, $\frac{1}{|J|} \sum_{j \in J} q_j$, is then a linear function of the total completion time meaning both metrics will yield similar performance for each algorithm.

Despite the Jenkins agents in the simulated environment having different processing speeds and the jobs being organized in pipelines, the results still follow the same pattern as could be expected from the described special case. An interesting extension would be to simulate the general unrelated machine setting where processing times are given by the machine-specific values p_{ij}.

5.1.3 Algorithm performance

As seen in figure 4.7, FIFO has the best performance for maximum time spent in queue. This is also true for the maximum lifetime of a pipeline, as seen in figure 4.3. Since FIFO sorts the queue by waiting time, the queuing time of a job j is not affected by jobs entering the queue after j meaning the time j must spend in the queue can not increase from other jobs passing j in the queue.

LIFO performs well when looking at the discounted pipeline cost in figure 4.4. This can be explained by looking at figure 3.1. As can be seen in the figure, the cost per time unit is higher when a pipeline has just been created. Since LIFO chooses the last job that entered the queue, pipelines which have a high cost per time unit will be picked over pipelines where the cost is already close to the upper limit.

The SPT performs the best for the average queue time, as seen in figure 4.6. Since all the quickest jobs get put in the front of the queue, most jobs in the queue have a short waiting time. However, the jobs with the longest processing times may have to wait until the queue is empty, causing them to be starved. When an aging factor is added, SPT still perform well for this metric but also adds a bit of fairness to the queuing, reducing the starvation problem.

The LPT algorithm performs the worst on most metrics, except for makespan minimization. This can be seen both on whole pipelines in figure 4.1 and with individual jobs in figure 4.10. The result follows the same pattern for the Jenkins server, as seen in figure 4.13. Given the discussion about the makespan in section 5.1.1, LPT is arguably the worst algorithm in the given environment.

The SMRTS and STRTS algorithms both perform well when it comes to the average lifetime of a pipeline, as can be seen in figure 4.2. However, figure 4.3 shows that both algorithms cause some starvation with high maximum lifetimes. Adding an

aging factor gives a good trade-off with some performance increase over FIFO in the average case but with a limited maximum lifetime.

Finally, the greedy algorithm introduced by Gupta et al. in [15] and [16] is the only evaluated algorithm that makes adjustments for the unrelated machine setting and the stochastic nature of the processing times. While other algorithms produce lower medians for the total weighted completion time, the algorithm shows some interesting characteristics. Figure 4.5 show that, apart from a single outlier, the worst case performance is lower than for other algorithms.

These results highlight the fact that the competitive ratio gives an upper bound for the value of the cost function rather than guaranteeing that the algorithm will have the best average performance. The handling of stochastic processing times is displayed in figure 4.8 and show that the algorithm suffers less performance penalty from working with expected values instead of actual values for processing times.

5.1.4 Results from the Jenkins server

Comparing the results from the Jenkins server in figures 4.13 to 4.15 with the corresponding simulation results in figures 4.10 to 4.12, we see some clear similarities. Figures 4.11 and 4.14 both show that SPT has the best performance when it comes to minimizing the average queuing time while LPT results in longer average queuing times than the FIFO baseline. Figures 4.12 and 4.15 show that SPT and LPT both suffer from starvation with LPT resulting in the longest maximum waiting times both in the simulator and on the real Jenkins server. Adding an aging factor to the SPT algorithm results in similar performance trade-offs in both cases with figures 4.12 and 4.15 both showing drastically reduced maximum waiting times with only small increases in the average waiting time, as seen in figures 4.11 and 4.14.

Because of the strong correlation between the patterns in the graphs, we conclude that the simulator appears to give an accurate model of the real Jenkins server.

There are small differences between the results from the Jenkins server compared to the results from the simulator. While on a real Jenkins server, the network plays a role in execution of the process. This could result in disruption or delays of the job dispatcher. Also, while sending jobs into the Jenkins pipeline, we noticed that the script would sometimes pause for a couple of seconds and then continue execution. The explanation of this is that when sending HTTP POST request, the client waits for a confirmation that the POST request has been sent correctly. These waiting times are not modeled in the simulator.

5.1.5 Ethical considerations

The core of the project, optimizing the use of computational resources, can hardly be considered controversial. The project is not looking to automate tasks currently handled by humans meaning well known ethical discussions, such as whether automation leads to unemployment or not, are not applicable.

5.2 Future work

This section introduces future ideas related to this thesis work.

5.2.1 Job prediction module

The main challenge in an online scheduling problem, compared to an offline equivalent is the lack of knowledge about what tasks will enter the queue. There are some methods to handle this, such as *forced idleness*, to prevent longer jobs submitted to an empty queue from blocking the resources in front of shorter jobs but these methods come with a performance cost for the average case.

An interesting addition to a scheduler would be the ability to predict what jobs will be submitted to the scheduler in a near future. Reliable predictions would allow the scheduler to make more informed choices about how to schedule the jobs in the queue.

5.2.2 Machine learning approaches

All of the algorithms evaluated during this project are based on static policies, using data from the jobs in the queue. A possible approach for future research would be to evaluate machine learning based scheduling algorithms that can handle situations in a more dynamic way.

Reinforcement learning is a machine learning technique that can be summarized as"...learning what to do - how to map situations to actions - so as to maximize a numerical reward signal." [22, p. 1]. Several approaches exist for applying reinforcement learning in scheduling [23] and it would be interesting to evaluate such algorithms in the given environment.

5.2.3 Discounted total weighted completion time algorithm

As discussed above, this metric fits our current environment. An interesting approach is to implement an algorithm based on this metric. This would consider waiting time more valuable in the beginning of queuing compared to later. Which would in theory deliver feedback quicker to the developer.

5.2.4 More algorithms for Jenkins server

During the project, only a subset of the algorithms evaluated in the simulator were tested on the Jenkins server. A natural extension would be to implement and evaluate the remaining algorithms on the Jenkins server as well.

Additionally, the tests on the Jenkins server were only performed using individual jobs. In order to strengthen the validity of the simulator results it would be beneficial to extend these tests to include complete pipelines as well.

5.3 Conclusion and summary

This thesis investigates if the performance of a CI pipeline can be improved by using scheduling algorithms that utilizes existing knowledge of jobs and machines available. The results indicate that it is possible to improve the performance under some metrics but that these improvements come at a trade-off with performance under other metrics.

The results also indicate that some performance improvements are possible with access to expected values for stochastic variables but that access to more accurate values will give better results.

We also conclude that the simulator built in this thesis is a good way of testing algorithms without having to deploy them on a real system. We make this conclusion since the results from the simulator and the Jenkins server show similar characteristics for several performance metrics.

Bibliography

[1] Jenkins Project Contributors. "Glossary," Jenkins Project. (), [Online]. Available: https://www.jenkins.io/doc/book/glossary/ (visited on 02/09/2023).

[2] K. Beck, *Extreme programming explained: embrace change* (The XP Series). Reading, MA: Addison-Wesley Professional, 2000, ISBN: 978-0-201-61641-5.

[3] M. Fowler and M. Foemmel. "Continuous integration." (May 1, 2006), [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html (visited on 01/30/2023).

[4] M. Meyer, "Continuous integration and its tools," *IEEE software*, vol. 31, no. 3, pp. 14–16, 2014.

[5] Wikipedia contributors. "Jenkins (software) — Wikipedia, the free encyclopedia." (2022), [Online]. Available: https://en.wikipedia.org/w/index.php?title=Jenkins_(software)&oldid=1127280095 (visited on 02/13/2023).

[6] J. F. Smart, *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly Media, Inc., 2011, ISBN: 978-1-449-30535-2.

[7] A. Earl. "Jenkins terminology changes," CD Foundation. (Aug. 25, 2020), [Online]. Available: https://cd.foundation/blog/2020/08/25/jenkins-terminology-changes/ (visited on 02/09/2023).

[8] J. Andersson and P. Andersson, "Increasing the performance of a continuous integration server," Master Thesis, Chalmers University of Technology, 2016. [Online]. Available: https://odr.chalmers.se/items/2131b535-8f53-414f-b818-19d9599d53c5.

[9] V. Berglund and I. Eriksson, "Improving the scheduling policy for a continuous integration server," Master Thesis, Chalmers University of Technology, 2020. [Online]. Available: https://odr.chalmers.se/items/18e58e77-2f0c-41a2-993f-fa8bb17f37b4.

[10] K. Kawaguchi. "Hudson." (), [Online]. Available: https://web.archive.org/web/20140701020639/https://www.java.net//blog/kohsuke/archive/20070514/Hudson%5C%20J1.pdf (visited on 04/20/2023).

[11] CloudBees. "Managing agents," CloudBees, Inc. (), [Online]. Available: https://docs.cloudbees.com/docs/cloudbees-ci/latest/cloud-admin-guide/agents (visited on 02/01/2023).

[12] M. L. Pinedo, *Scheduling, Theory, Algorithms, and Systems*, 5th ed. Springer Cham, 2016, ISBN: 978-3-319-26580-3.

[13] A. Tanenbaum and H. Bos, *Modern Operating Systems, Global Edition*. Pearson Education, 2015, ISBN: 9781292061955.

[14] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[15] V. Gupta, B. Moseley, M. Uetz, and Q. Xie, "Greed worksonline algorithms for unrelated machine stochastic scheduling," *Mathematics of operations research*, vol. 45, no. 2, pp. 497–516, 2020. [Online]. Available: `https://doi.org/10.1287/moor.2019.0999`.

[16] V. Gupta, B. Moseley, M. Uetz, and Q. Xie, "Corrigendum: Greed worksonline algorithms for unrelated machine stochastic scheduling," *Mathematics of operations research*, vol. 46, no. 3, pp. 1230–1234, 2021. [Online]. Available: `https://doi.org/10.1287/moor.2021.1149`.

[17] Wikipedia contributors. "Aging (scheduling) — Wikipedia, the free encyclopedia." (2022), [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Aging_(scheduling)&oldid=1089597043` (visited on 12/14/2022).

[18] Python Software Foundation. "random - Generate pseudo-random numbers." (Feb. 9, 2023), [Online]. Available: `https://docs.python.org/3.8/library/random.html#notes-on-reproducibility` (visited on 06/05/2023).

[19] Jenkins Project Contributors. "Priority sorter," Jenkins Project. (2022), [Online]. Available: `https://plugins.jenkins.io/PrioritySorter/` (visited on 04/20/2023).

[20] J. A. Rice, *Mathematical statistics and data analysis*, 3rd ed. Brooks/Cole, 2007, ISBN: 978-0-495-11868-8.

[21] S. Gibson and G. Lee. "Coninuous integration: Fail fast and fail first," Software Sustainability Institute. (Feb. 27, 2020), [Online]. Available: `https://software.ac.uk/blog/2020-02-27-continuous-integration-fail-fast-and-fail-first` (visited on 02/03/2023).

[22] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[23] C. Shyalika, T. Silva, and A. Karunananda, "Reinforcement learning in dynamic task scheduling: A review," *SN Computer Science*, vol. 1, pp. 1–17, 2020.

A

Box plot reading guide

All boxplots used in this report are generated using the same method. In the plot, the *median* value of a data is marked with an orange line. A box is drawn around the median from the *lower quartile* to the *upper quartile*, meaning 25 % of the values are in the part of the box below the median and 25 % of the values are in the part of the box above the median. The distance between the quartiles (height of the box) is called the *interquartile range* (IQR). Outside of the box, whiskers are drawn from the upper quartile to the maximum value and from the lower quartile to the minimum value. If any values are more than a given limit above the upper quartile or below the lower quartile, the whisker is limited at the maximum (or minimum) value within that limit. Values outside the limit are marked as outliers using a circle ($\circ$). All plots in this report use a standard limit [20, p. 403] of $1.5 \cdot \text{IQR}$ for the whisker length.

Figure A.1 shows a box plot with median 50. The lower quartile is 42.5 and the upper quartile 57.5, which gives $IQR = 15$. Outliers at the top and bottom are marked with circles ($\circ$).

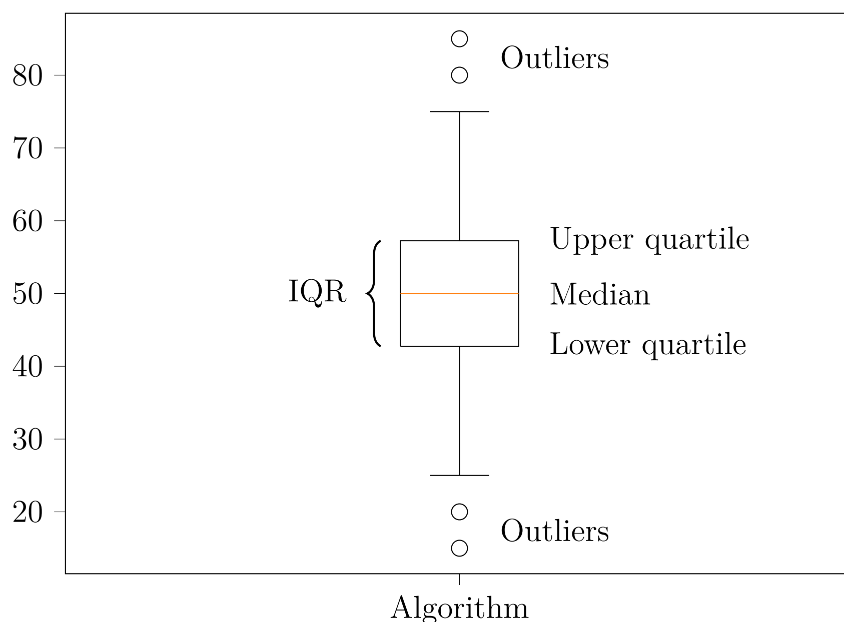


Figure A.1: Sample boxplot showing the median, quartiles and outliers for a set of values.