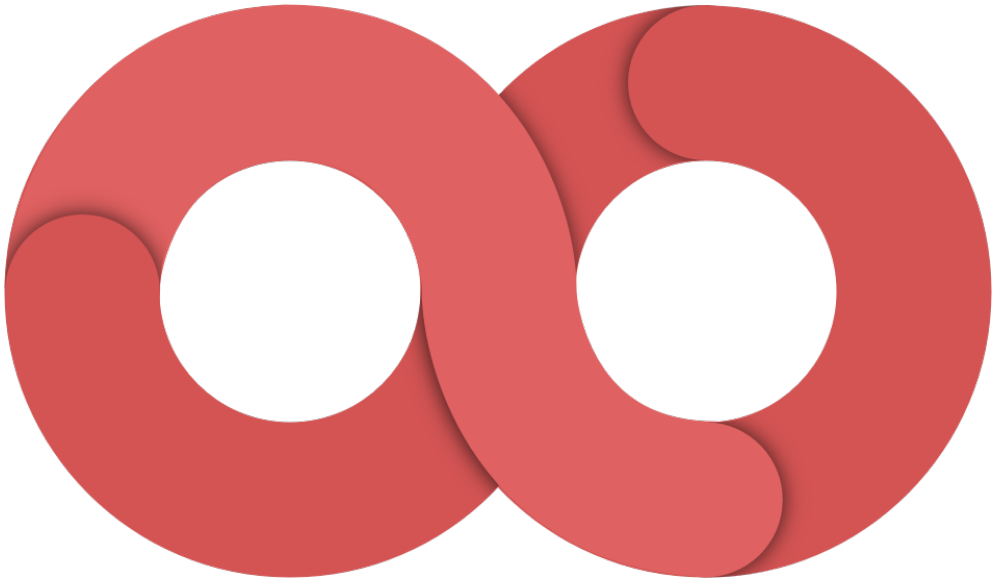

Improving the scheduling policy for a Continuous Integration Server

Master's thesis in Computer Systems and Networks

VIKTOR BERGLUND
ISAK ERIKSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2018

Improving the scheduling policy for a Continuous Integration Server

VIKTOR BERGLUND
ISAK ERIKSSON

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Improving the scheduling policy for a Continuous Integration Server
VIKTOR BERGLUND
ISAK ERIKSSON

Supervisor: Marina Papatriantafilou, Department of Computer Science and Engineering
Dimitrios Palyvos-Giannas, Department of Computer Science and Engineering
Advisor: Pontus Andersson, Ericsson AB
Examiner: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering

Master's Thesis 2020
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 2020

Improving the scheduling policy for a Continuous Integration Server
VIKTOR BERGLUND
ISAK ERIKSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Continuous integration (CI) is a method used in software development to make it easier for developers to handle integration and testing of their code. By automating this process the development process can be streamlined and sped up. One possible way of achieving this could be to improve the scheduler in the CI system to handle scheduling of integration jobs in a more efficient way.
Six scheduling algorithms are empirically evaluated: Longest Job First (LJF), Shortest Job First (SJF), First In First Out (FIFO), Random priority and LJF/SJF with an added aging factor. Tests were done on a Jenkins instance as well as a simulation model developed for the thesis. The execution time of the jobs in the job sets are assumed to follow the Pareto power-law probability distribution, which can be summarized as "20 percent of the jobs account for 80 percent of the execution time". Makespan and average response time are known measures in the scheduling literature and we measure the algorithms against these. We also introduce the measure average matrix response time, which we consider to be the most important for the objective of this thesis.
Tests were carried out on a real Jenkins instance, a server used for building software common in a continuous integration setting, and on a simulation model we developed. The model can simulate system slowdown when running more threads than available cores.
Based on our tests we conclude that LJF with an added aging factor is the best performing algorithm against the measure average matrix response time.

Keywords: continuous integration, scheduling, Jenkins

Acknowledgements

This thesis was conducted for Ericsson AB. We would like to thank our company supervisor Pontus Andersson, as well as the whole Watson team for providing support and knowledge during this whole process. We also want to thank our supervisors Marina Papatriantafilou and Dimitrios Palyvos-Giannas for all the the good input and their help with this thesis.

Viktor Berglund & Isak Eriksson, Gothenburg, May 2020

Contents

1

Introduction

As software development projects grow larger and involve many developers it has become common to use some kind of continuous integration (CI) solution [2]. The main idea of CI is to make all developers cooperating on a software project check in their code into a central repository daily. The code is automatically built and tested to ensure it is working. By applying this technique, errors can be detected and fixed faster and more easily. Getting that feedback fast should mean that no time has to be spent on understanding the code again if the tests fail and the code needs to be updated. Development may also be halted since other code may be dependent on the code that is tested.

A CI server works together with a version control system such as Git or Subversion, used to manage software projects involving several developers. The latest agreed-upon versions of these software projects are called the *baselines* and are stored in a central repository. The way of working dictates that developers first make changes to their respective local copies of the software. When they feel confident about a change they have made they send an update of this code to the central repository, we refer to this as a *commit*. Before the baseline is updated with this commit, a large part of the code has to be compiled and tested.

By using a CI server to compile and test new code instead of having the developers do it manually, the efficiency of software development projects can be increased. This is especially true for many of today's enormous projects employing hundreds of people. With this many people working on a project, even very small delays in the workflow add up to many man-hours worth of waiting. This means that a great deal of time can be saved by even further streamlining the integration process.

A possible way to improve this process is to improve the scheduling policy of the CI server. Generating an optimal schedule for a set of jobs to run on parallel machines is a well studied NP-hard problem that can come in many different variations. We want to investigate if it is possible to improve the scheduling policy for a CI server where the overarching goal is to minimize the amount of time spent on integration.

1.1 Purpose

The purpose of this project is to reduce the amount of time it takes between a commit occurring and the developer getting feedback from the testing system. By doing this, considerable amounts of time could be saved, especially for larger companies that employ thousands of software engineers [3]. We investigate what scheduling objectives are relevant for this purpose when scheduling for a CI server and what algorithms best serve these objectives.

This thesis is done in collaboration with Ericsson Lindholmen. Ericsson uses the CI server Jenkins which is one of the most commonly used [4]. As a use case, we will test the different scheduling policies on this server.

We also develop a simulation model to be able to carry out more extensive tests. The model is developed using data gathered from measurements from our use case.

2

Preliminaries

This chapter introduces scheduling and CI concepts, the system model and the algorithms to be evaluated.

2.1 System model

Our system model is illustrated in figure 2.1. The integration process starts when a developer makes a code commit to a version control system. The CI server is alerted by this commit and generates a set of jobs based on it. Such a job can do a number of things, but in a CI setting a job generally compiles or tests code, or sets up an environment for doing this. We refer to such a set of jobs that are triggered by the same commit with the same release time as a *job matrix*.[1] The jobs in a job matrix then enter the central queue held by the CI server one-by-one in random order.

Jobs then run on machines, referred to as *slaves*, that have a specified number of slots that determine how many jobs the slaves can run at the same time. The slaves do not have any queues and only execute the jobs they get assigned. Which job gets assigned to which slave is decided by the load balancer that assigns scores to slaves. The slave with the highest score gets assigned the job at the front of the queue. In the event of a tie, the slave with the lowest index is chosen.

In our system model, we assume that any job can run on any machine. This is also true for a vast majority of the jobs in the system where we run our tests, with only a small minority having a designated slave. Once a job is finished, feedback is sent back to the developer.

[1] This terminology is derived from a common Jenkins plug-in called "Matrix Project Plugin".

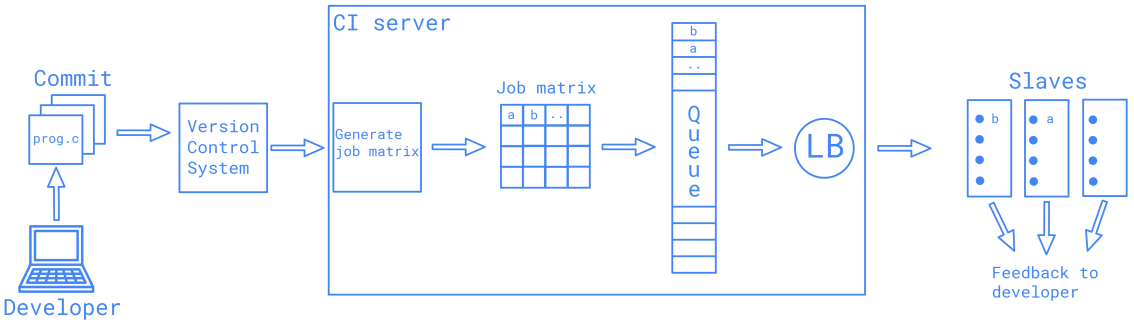

Figure 2.1: Overview of the developer environment, showing all the steps in the integration process.

Scheduling problems can be specified in a number of ways. The following subsections describe the general constraints scheduling policies must respect in our system model.

2.1.1 Parallel machines

Since there are multiple slaves on which the jobs can run, the system under consideration is a system of parallel machines. Scheduling jobs on parallel machines - as opposed to a single machine - with the objective of minimizing the finishing time is a well studied NP-hard problem [5]. This means that no algorithm can be guaranteed to find the optimal solution to this problem in polynomial time. This motivates a heuristic approach.

2.1.2 Non-preemptive scheduling

Once a job has left the queue and started executing on a slave it cannot be interrupted or moved to another machine, making the scheduling non-preemptive. A motivation for this is that in a CI setting, a job may need to download a workspace before execution. To move this job would then require the workspace to be downloaded again. Thus, in our system model, the scheduling is non-preemptive, which is also the case for the Jenkins server. Non-preemptive scheduling decreases the number of possible schedules and also limits the possible scheduling algorithms.

2.1.3 Clairvoyant scheduler

A scheduler that knows the characteristics of incoming jobs, such as execution time, prior to running them is called clairvoyant, while a scheduler that does not is called non-clairvoyant [6]. In a CI setting some code change is made by a developer. A number of jobs are then generated. A job can, for example, be the compilation of a number of files. During the development process, these files must be compiled many times, often with very small changes. This means that the job compiling these files will run a number of times. Thus, data from past runs of a job can be gathered. By using this information it is possible to approximate the execution time of a job. By using this approximation of how long it will take to execute a job the scheduler can be seen as clairvoyant. Without knowing any characteristics of incoming jobs there would be no information to base the scheduling decisions on, which would make the scheduler operate blindly. This problem could be handled if preemptions were allowed since decisions could then be made based on information revealed when the jobs start running.

2.1.4 Online scheduling

Since jobs are triggered by code commits made by developers and not known in advance by the server, jobs will enter the queue continuously, thus the scheduling has to be performed online. Online schedulers are different from offline schedulers. They perform the scheduling on a job set that is continuously changing, with no way of knowing what it is going to look like in the future, whereas offline schedulers have complete knowledge about the set of jobs beforehand [7]. For an online scheduler to be reasonable it needs to be clairvoyant and/or preemptive. Otherwise, the scheduler would be useless since it has no knowledge prior to scheduling a job and the job cannot be moved after scheduling.

2.2 Specific use case

While continuous integration is a widespread concept, the way it is implemented and used can differ from project to project. In this section we explain the main characteristics of the system currently in use at Ericsson Lindholmen.

2.2.1 Environment

We are using the continuous integration server Jenkins [8]. This server is based on the master-slave architecture, where the master holds a single scheduling queue and

distributes jobs across several slaves. The jobs are distributed using a load balancing algorithm that makes decisions based on the current load of the slaves, a high score means a low load. The scores are calculated according to equation 2.1.

$$\text{Score} = \frac{\#\text{TotalSlots}}{0.5 + \#\text{BusySlots}} \tag{2.1}$$

The Jenkins slaves are lightweight client programs running on virtual Linux machines hosted on large server computers. The slaves may have a different amount of threads available, however they are all configured to be able to run as many jobs in parallel as they have threads.

3

Problem Analysis

This chapter analyses the problem at hand and defines the objectives a schedule for a CI-server should be evaluated.

3.1 Sequential builds

Jobs are viewed as being part of a *build pipeline*, which is a general continuous integration concept, where jobs have to be executed sequentially. The executions consist mainly of building code and are therefore referred to as sequential builds. The jobs are assigned different static priorities depending on how far along they are in the pipeline. The further into the pipeline a job is, the higher its priority will be. These static priorities trump the weight assigned to a job by the scheduling algorithm. A higher priority job will always be run before a lower priority job, no matter the scheduling policy. Thus, we can view the system as a combination of several smaller queues, one for each priority level, as shown in Figure 3.1.

Every such queue contains jobs of a specific static priority. All the queues use the same scheduling policy for intra-queue scheduling. If a policy can speed up the execution of one queue then it will not affect the other queues negatively since they would have to wait for all the jobs in that queue no matter what, given the different static priorities. Furthermore, if the higher priority queue was executed faster, then the system as a whole has to have saved time in the process. Thus we determine that it is enough to consider only one queue in isolation during testing, i.e. only using one level of static priority. The results for that one queue will be applicable to all the different priority queues when combining them. This means that in our system model we only consider a single queue.

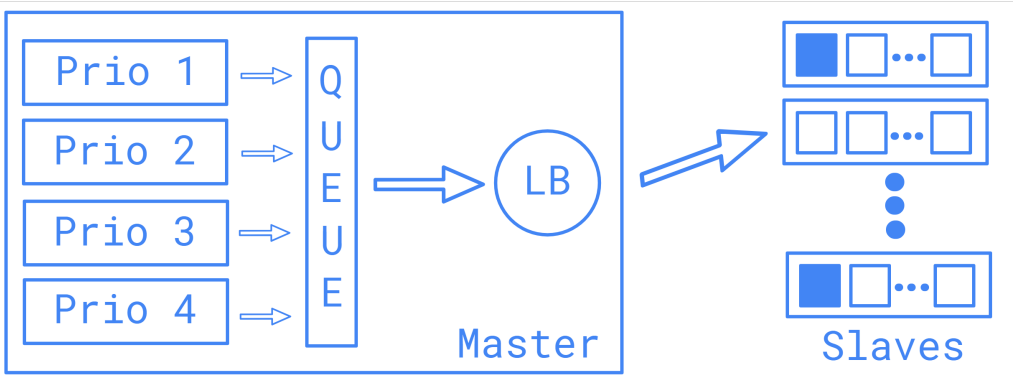

Figure 3.1: Overview of the system, showcasing which parts run on the master. Static priorities, overriding the dynamic priorities, in the same queue are equivalent to having separate queues for every static priority level.

3.2 Defining a good schedule for a CI server

There are many objectives a schedule can be evaluated by. In the literature, concepts such as *throughput, average waiting time, average response time, makespan* and *stretch* are found [9] [5].

An assessment of ours is that scheduling for a CI server is special in that tests may pass or fail and compilations may result in an error or be completed. A common practice in software development is the aim to "fail fast" [10]. In our context, this means that it is more important to know if a test on a piece of software fails than to know if it succeeds since failure will mean that the bug has to be fixed and hence the whole commit has to be redone. With this objective in mind, minimizing the average response time of jobs - also referred to as *flow time* - is relevant. The average response time is minimized by running a large number of jobs early, and thus more tests will be done early. With more tests completed early, there is a higher chance of a failure occurring early and thus a response can be sent faster (meaning that a developer could be more efficient).

Another highly relevant measure for evaluating a schedule is minimizing its *makespan*. The makespan is generally defined as the last completion time of a job from a set of jobs. This relates directly to decreasing the total time spent on integration, and generally the throughput of the system. Minimizing makespan for a set of n machines, n>2, is an NP-hard problem [5]. The problem is modeled as jobs with run times running on machines and the problem is envisioned as a number partition problem, i.e. partition a set of numbers into sets with equal sums, where the sums represent schedules [11]. If every schedule is equally long, as little time as possible will go to waste.

As stated, a developer makes a commit and then, usually, many parallel jobs are carried out based on this commit. One single commit is thus generating a matrix containing a lot of jobs with different execution times, but from the developer's perspective it can be seen as one single, large job. It is often the case that matrices enter the system consecutively. This makes the *average makespan* of matrices a relevant measure. Taking the average makespan of consecutive matrices will equal the average response time when one matrix is viewed as one job. We will, therefore, call this measure *average matrix response time*. A schedule might be good with regards to the total makespan of a number of matrices but at the cost of making a job from an earlier matrix wait. Trying to minimize the average matrix response time will penalize this behavior.

If the job to machine ratio is low, i.e. if the load of the system is low, the scheduling policy hardly makes a difference. Since there are enough resources such that every time a job enters the queue it can be scheduled on an idle machine. We must, therefore, determine a load where different policies can produce different schedules.

The most important measures of interest for scheduling on a CI-server is:

- Makespan
- Average response time
- Average matrix response time

3.3 Avoiding bad co-scheduling

Previous research on balancing the load for a Jenkins continuous integration server has determined that a load balancer that assigns jobs to nodes according to equation 2.1 yields the best load balancing [12]. We use this formula for the load balancer. The objective of load balancing, however, does not only depend on the decision to choose which slave to place a job on but can also be served by assigning priority to jobs. This is true since the above-mentioned formula only makes sure that all jobs are evenly distributed onto the machines. It has no way of knowing what type of jobs a machine has been assigned. For example, it could mean that one machine only gets assigned very heavy jobs, while another only gets assigned lightweight jobs. Such an uneven distribution of jobs could affect the execution times of jobs in a negative way.

For the specific context of computer programs running on machines there are many resources that are shared and influence the performance of the system. For example, running all CPU intensive jobs on the same machine will influence the performance in a negative way since the context switching needed by the CPU easily slows down the execution time [13]. In scheduling for operating systems processes are often

differentiated into CPU-bound and IO-bound based on their respective resource requirements [14]. To best utilize the resources on a computer, CPU-bound and IO-bound processes should be scheduled to run together since they do not compete over resources.

Following de Blanche we call a situation when jobs are scheduled on the same machine such that they influence their execution time negatively *bad co-scheduling* [15]. The influence of co-scheduling complicates the task of scheduling since a policy that produces a good schedule when fixed execution times are assumed may unnecessarily co-schedule tasks with similar resource requirements and thus making the makespan or other relevant measures in effect, worse than another policy. There might be policies that under certain constraints systematically produce schedules with a lot of bad scheduling and others that avoid it. This implies that how a policy co-schedules jobs must be taken into consideration when evaluating what scheduling policy is appropriate for a system.

In a CI context, which is centered around developing new software, it is common with jobs that are set up to perform the compilation of code. Compilation is a computation heavy task. As the code base of a project grows larger the compilation of code for it gets more and more complex. To make sure that the increased complexity of the code does not slow down the compilation, it is common to let the build automation tools use several threads.[1] While this might lead to an increase in compilation speed in controlled environments, it could also lead to too many threads starting doing heavy work which will slow down other processes on the computer.

Another issue related to co-scheduling is the number of slots on the slaves. The number of slots determines how many jobs can be scheduled on a slave at the same time. On the actual hardware, the number of physical cores of the slaves determines how much computations can be done in parallel. While setting the number of slots high means that more jobs can be dispatched from the queue, this may decrease the throughput of the system. If there are more CPU-bound jobs than physical cores scheduled on the same machine the overhead due to context switching will make it more reasonable to execute some of the jobs sequentially. At the same time, too few slots may underutilize the system.

[1] At Ericsson the common build automation tool called GNU Make is used to handle the compilation, where the j-flag can be set to specify the number of threads [16].

4

Methods

This chapter describes the implementation of the scheduling algorithms investigated in this thesis, testing environment and specification of how the tests were carried out.

4.1 Choice of algorithm

Depending on the objective and the specific circumstances different scheduling algorithms are appropriate. In this thesis, we investigate six different algorithms that fulfill the constraints outlined at the end of section 3.2. All these algorithms continually dispatch jobs from the front of the queue as long as there are free slots in the system. When a new job arrives it gets placed in its proper place in the queue based on the weight assigned to it by the current scheduling algorithm. The algorithms are:

Longest Job First (LJF): assigns a weight to jobs based on the execution time of the job, the longer the execution time the greater the weight.

Shortest Job First (SJF): assigns weight based on the execution time of the job, the shorter the job the greater the weight.

First In, First Out (FIFO): assigns weight based on how long a job has stayed in the queue.

Random: assigns a random weight to jobs.

LJF/SJF-aging: assigns weight according to job execution time, but also adds the time spent in the queue scaled with an aging factor.

These algorithms will be measured against the objectives explained in section 3.2. LJF and SJF suffers from the problem of starvation. Starvation happens when low priority jobs are continually denied access to a machine [17]. FIFO and LJF/SJF-

aging do not suffer from this problem since when a job has entered the queue it is guaranteed that it will eventually run if we assume no crashes and finite execution times. For the Random algorithm, all jobs can eventually be expected to run, given enough time. In practice however, a job may have to wait in the queue an unacceptably long time.

In a non-preemptive setting with a non-clairvoyant scheduler, we assess that FIFO is the best policy. It will at least guarantee fairness, which means that all jobs will eventually get a chance to run since no job will ever be allowed to jump the queue. This means that it might still have its merits, despite the simple approach to scheduling, since basic fairness is one of the most important aspects of a scheduling system. While another policy would have no metric to base a schedule on, which would mean it would assign the jobs a weight completely at random. However, our scheduler is clairvoyant, hence other policies than FIFO may produce a better schedule, based on the information known about the jobs arriving.

For a single matrix, we expect FIFO to perform the same as Random since the jobs from the matrix will enter the queue as fast as possible. This will result in random arrival time. When more than one matrix is in the system, however, FIFO will make sure the jobs from each matrix are scheduled together. No jobs from a newer matrix will get scheduled before a job from an older matrix. For a single matrix, the aging factor becomes irrelevant since all jobs enter with the same release time.[1]

The merit of the Random algorithm is that it in most cases achieves a shuffled queue, which could be beneficial in some cases, particularly with regards to load balancing [15].

4.2 Implementation details

This section describes the implementation of our different scheduling algorithms in Jenkins as well as a simulation model we developed to be able to make more extensive testing.

4.2.1 Jenkins priority sorter plugin

Jenkins has core functionality that is supposed to be stable with custom features being added by plugins. The plugins, as well as the Jenkins core code, are written in Java. To implement the different scheduling algorithms we modified an already existing Jenkins plugin called "Priority Sorter" which has support for static priorities

[1] Jenkins jobs from the same matrix have a δ inter-arrival time, but this δ is very small.

[18]. The static priorities allow the administrator of the system to give a job a predetermined priority, to allow for some jobs to always be run before others.

The implementation of the Jenkins queue is defined in the core of Jenkins, it is not desirable to change it directly. The development must, therefore, be done with its characteristics in mind. The Jenkins queue is sorted every five seconds, and when a new job arrives the job gets inserted into its proper place based on its weight. The job at the front of the queue is assigned an execution slot at a slave by the load balancer as soon as there is one available. Rather than directly manipulating the queue one can change how the items are ranked in the queue.

The sorting of the queue depends on the `compareTo` function from the Comparable java interface, which enables instances of classes to be compared. When the queue is sorted, the `compareTo` method is called and the weights of two items are compared. By modifying how the weight is set for items - jobs in our case - one modifies how the queue will be sorted. Low weight means a high priority. The algorithms were implemented in the following way:

LJF: The weight of a job is based on its estimated execution time, which is calculated by the standard Jenkins function `getEstimatedDuration()`. It takes an average of the execution time of the job during its last three runs. The weight assigned by LJF has to be multiplied by -1 to make longer jobs higher prioritized than shorter jobs.

SJF: `getEstimatedDuration()` is used to assign weights for SJF as well. But since SJF looks to prioritize short jobs the result of it does not need to be modified further.

Random: When a job enters the system it is given a pseudo-random weight. The random number is generated using the standard Java function `Math.random()`, which generates a number from 0.0 to 1.0 with an approximately uniform distribution.

FIFO: At the start of this thesis work FIFO was the default scheduling policy used. There was already an implementation of FIFO in the Priority Sorter plugin. When comparing jobs for the scheduling it compares the timestamp of when the two jobs had entered the queue. Information which is readily available in Jenkins.

Aging: To make sure no jobs get starved when using LJF or SJF it is possible to add an age aspect. Every time a comparison is made between two jobs, their respective time in the queue is taken into account. The weights that are used in `compareTo` when aging is enabled are calculated according to equation 4.1. For our experiments, an aging factor of ten was chosen. With the job set we were using this meant that no jobs from a newer matrix would get scheduled before jobs from earlier matrices, i.e. a FIFO policy on the matrix level.

```
weight = weight - timeInQueue() * agingFactor        (4.1)
```

To make it possible for the users to change between the different sorting algorithms and update the aging factor to suit their needs a user interface was implemented that is usable from the Jenkins web interface.

One factor which could cause some noise in our measurements is that when using job matrices, the sub-jobs arrive sequentially with a very small release time apart. This means that the initial distribution of jobs at free slots will be done randomly since there is only one item in the queue when the dispatching is done. Since this noise factor only comes into play when the queue is empty, its impact is proportional to the jobs-to-slots ratio.

4.2.2 Simulation model

The Jenkins plug-in requires testing in real-time. The jobs have a lower limit to their duration since making the jobs too short will make noise factors severe. Furthermore, the server needs to be restarted to change configurations. To be able to make more tests with different kinds of job sets, a model of the system was developed.

The risk of overload, where the system is slowed down due to bad co-scheduling, increases as the number of slots increase. At the same time, more jobs can be carried out in parallel as the number of slots increase. The simulation model takes both these factors into consideration. When several heavy jobs are assigned to a machine the execution time is increased according to measurements from the real system, the tests are described in section 5.2. In this model, we assume that if the execution time of a job exceeds a certain threshold it is also heavy. This assumption is made based on observations from our specific use case. The simulation model enables us to randomly generate a lot of matrices. The pseudo-code is shown on page 16.

A node is represented as a list of slots. A slot is represented as an index in this list. Each index corresponds to a tuple that represents the time left before a new job could be assigned to this slot, information about whether the job is heavy or light and what matrix the job belongs to. Jobs are taken from a queue and fills up all slots which means that the number in the tuple is set to be the duration of the job. If the job is longer than a threshold time, it is set to be heavy.

The model starts by distributing jobs from the queue onto all empty slots. On what node is determined by a load balancer function, defined by the same equation as in our system model. Once all slots are occupied or the queue is empty the system starts stepping forward "in time" with an interval. This interval is set using the `delta` variable. When stepping forward, all jobs at the nodes are decreased with the delta value. The delta value is divided by a factor dependent on how many heavy jobs have been assigned to the node. More heavy jobs mean that it is decreased with a smaller amount to simulate the extra time it will take to execute due to the overload. However, it is still registered as if delta-time has passed as long as there

are positively valued slots on the node. When a job's duration reaches zero it is removed from the slot and another job takes its place if there are jobs left.

To calculate makespan we declare a list filled with zeroes where each index represents a node. If something moves on the node we add delta to it and at the end returns the maximum of this list. To calculate average response time we declare a list where each index represents a node, this index then represents a list filled with empty lists where each index represents a slot. We add delta to each slot for every iteration. When we change job we copy this value and change index. When all jobs have been executed we sum all values and divide by the number of values. To calculate the average matrix response time we mark every job as belonging to a matrix. When a job is finished we check what matrix this job belonged to and update the time on the corresponding matrix.

The queue is implemented as a list, which makes it simple to mimic the different scheduling strategies. To simulate LJF and SJF the queue is sorted and reversed and just sorted, respectively. Since the jobs are taken one by one from the front of the list this makes sure they are selected in the correct order. To add the aging factor the list is split into four equal parts prior to sorting. These smaller lists are then sorted separately before they are all combined into one list, which is used as the final queue. This accurately mimics the scenario where new jobs enter the system when there are already older jobs waiting in the queue, with the aging factor preventing jumping the queue.

Simulation Model: Pseudo code describing our simulation model. The simulation was run for each *slots* value, 1 to 16.[2]

```
nodes = 2
node = [(0,LIGHT/HEAVY,MATRIX)] * slots, makespans = [0] * nodes
avgRespTimes = [] * nodes * slots, avgMakespans = [0] * nodes
delta = 0.1
activeSlot = True
slowdown = [1,1,1.4,1.8,2.2 ...]
while activeSlot or queue not empty:
    activeSlot = False
    if there are free slots and queue not empty:
        activeSlot = True
        node = loadBalancer.getNode()
        job = queue.removeFirst()
        if job > heavyThreshold:
            weightedJob = (job.size,HEAVY,job.matrix)
        else:
            weightedJob = (job.size,LIGHT,job.matrix)
        slot = node.getFirstFreeSlot()
        node[slot] = weightedJob
    else:
        for all nodes
            deltaMoved = False
            heavyJobs = countHeavyJobs(node)
            for all non-empty slots on node:
                deltaMoved = True
                node[slot][0] = node[slot][0]
                                - delta/slowdown[heavyJobs]
                avgRespTime[node][slot].getLast() += delta
                if node[slot][0] <= 0:
                    node[slot].clear()
                    maxVal = avgRespTime[node][slot][-1]
                    if node[slot][2] == matrix0
                    and maxVal > avgMakespan[0]:
                        avgMakespan[0] = maxVal
                    else if node[slot][2] == matrix1
                    and maxVal > avgMakespan[1]:
                        avgMakespan[1] = maxVal
                    avgRespTime[node][slot].add(maxVal)
                else:
                    activeSlot = True
            if deltaMoved:
                makespans[node] += delta
```

[2]Python source code can be found at https://github.com/ondaapotekaren/jenkinsModel

4.3 Risk of overload

In addition to the simulation model, to get an estimation of how the risk of overload caused by bad co-scheduling affects the system, the predicted slowdown can be calculated in a more general way based on the measurements specified in section 5.2. These calculations assume a randomly sorted queue.

We start by assuming a binomial distribution of heavy jobs onto a machine with a specific number of slots. A machine has n slots that can be regarded as n trials. An assignment of a heavy job onto the machine counts as a success and has the probability p [3]. A random variable X is defined as the number of heavy jobs on the machine. Furthermore, we take a probability $P(X = k)$ to mean that $P(X = k)$ of the time the machine has this number of heavy jobs on it. From our measurements, we have a specific overload scalar x_k for each k, such that if k heavy jobs are simultaneously running on a machine we get x_k slowdown. This means that we can define a slowdown function f that gives the slowdown, given the probability of p and the number of slots n on a machine.

$$P(k; n; p) = P(X = k) = \binom{n}{k} p^k (p-1)^{n-k} \tag{4.2}$$

$$f(n, p) = \sum_{k=1}^{n} x_k P(k; n, p) \tag{4.3}$$

The probabilities of the different scenarios occurring for different job sets are showed in figure 4.1. The graph is showing the probability of overload at a varying number of slots and how it is changing given different levels of heavy jobs in the system. The different lines represent different probabilities of assigning a heavy job to a slot, which corresponds to the percentage of heavy jobs in the system. They are calculated according to equation 4.3.

As we can see the risk for overload increases if the amount of slots on the machine is increased. But the machine which is supposed to run the jobs assigned to the slots has a set amount of threads available. Thus the same amount of resources will have to be spread on a larger amount of jobs, which will always lead to a slowdown in the execution of those jobs.

[3] We keep p the same regardless of how many heavy jobs have been dispatched, this simplifies the calculation and enables us to model the sampling as with replacements. We believe that this property becomes more reasonable as the size of the queue grows.

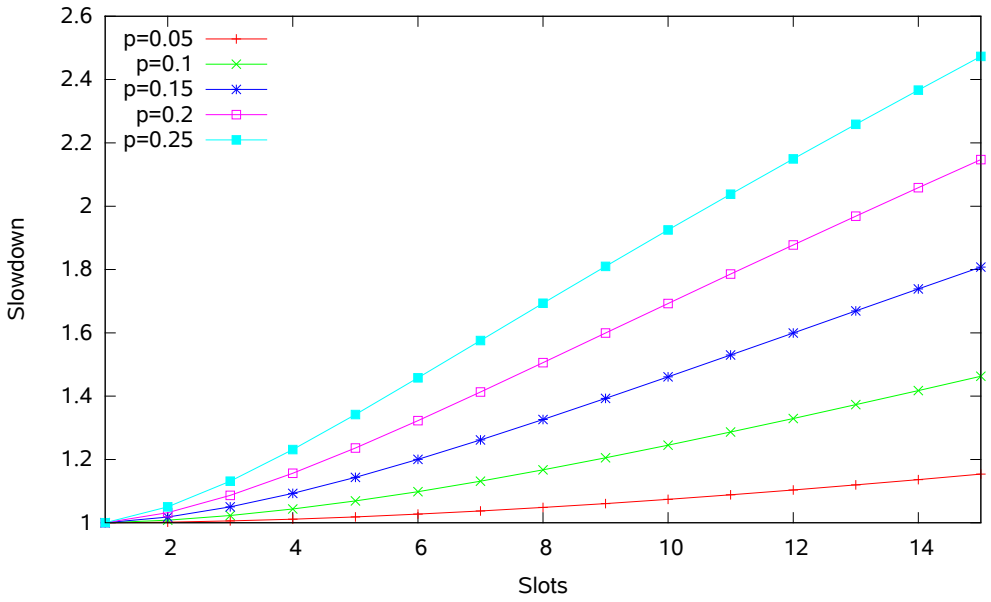

Figure 4.1: Slowdown predicted by the model for different amounts of heavy jobs in the job set when the number of available slots is increased. Each line represents a different probability of heavy jobs in a job set.

5

Evaluation

This chapter presents the results. This is done in three parts. Section 5.1 presents the results from tests made on a local Jenkins instance. Section 5.2 presents the results from load tests. Finally, section 5.3 presents the results from tests performed using the simulation model.

Below is a table summarizing how the algorithms were tested. Matrix size refers to the size of matrices tested, while job set size refers to the total amount of jobs tested in each test.

Test	Algs. tested	Tests	Matrix size	Job set size	Measure	Description
Local Jenkins instance with a single matrix	LJF SJF Random FIFO	30	100	100	Makespan Avg. response time	One matrix for each test, execution time fixed for all jobs.
Local Jenkins instance with consecutive matrices	LJF SJF Random FIFO LJF-aging SJF-aging	30	100	500	Makespan Avg. matrix response time Avg. response time	One matrix every 30 seconds for a total of five matrices, execution time fixed for all jobs.
Simulation with overload	LJF SJF Random LJF-aging SJF-aging	1000	25	100	Makespan Avg. matrix response time Avg. response time	Model of the system, with overload values based on measurements. Four matrices fill up the queue with no release time apart
Simulation no overload	LJF SJF Random LJF-aging SJF-aging	1000	25	100	Makespan Avg. matrix response time Avg. response time	Model of the system, does not account for overload of the system. Four matrices fill up the queue with no release time apart

Summary table: Tests that were run to evaluate the different algorithms.

5.1 Local Jenkins testing

For these tests we assumed that there was no interference between jobs, i.e. if all heavy jobs run on the same machine their execution time will not be affected. To simulate different loads on the system the number of available slots was varied from five up to 100.

The purpose of testing using a real Jenkins instance was to evaluate how well the different algorithms performed with regards to the defined objectives: minimizing response time, makespan and average matrix response time.

5.1.1 Job set

For the job sets, we assumed a Pareto distribution. A Pareto distribution is a general type of distribution popularly described by the 80-20 rule, where 20 percent of the causes account for 80 percent of the effects and vice versa [19]. The Pareto distribution is found in different circumstances, among others: load testing, web services, wealth distribution and bugs in software. We also observed this distribution in our use case where 20% of the jobs accounted for about 80% of the execution time.

For the tests, we generated ten Pareto distributed matrices that were used for all the different policies. To generate these we used the NumPy package, which is a package for scientific computing [20]. It has a built-in function that can generate a Pareto distributed data set. An $\alpha = 1.161$ value was used to achieve an exact 80-20 distribution.

We performed two types of tests: single matrix and consecutive matrices. For the single matrix tests, one matrix was run and completed before the next one was run. When testing with consecutive matrices five matrices were run at the same time but with a 30 second release time between them. This is a good approximation of a real CI system, where the developers are continually adding new jobs to the queue before all earlier jobs are completed. The five matrices were all running a different generated Pareto distributed job set, to further simulate reality. The aging factor described in section 4.2.1 only comes into consideration when running multiple matrices at once, it does not affect the queuing of jobs from the same matrix.

We ran every policy with ten different job matrices, four times each. The number of available slots was varied from five up to 100. The first run after every restart repeatedly produced slower execution times, thus we discarded the results from the first run after every restart. We could not find the root of this behavior, but one

possible reason could be that no values had been cached yet. We computed an average, first for the three tests and then for the ten different matrices.

5.1.2 Software setup

To be able to automate the testing process, Jenkins' built-in command-line interface, Jenkins CLI, was used. Jenkins CLI makes it possible to manage the Jenkins server from the command line, which made it possible for us to fully automate the testing. We created bash scripts that would set up the Jenkins server, start our test matrices and change between the scheduling algorithms. After every run of a job, Jenkins produces and stores a log file with all the information about the build, including build duration and start time. These log files contained all the information we needed to evaluate our policies. After each test, we ran a Python script to scrape all the files created for the data and compile it into the information we wanted, e.g. the makespan of the matrices run. The information was saved as a CSV file which made it easy to use gnuplot to create graphs.

For these tests, the jobs only ran the Unix command `sleep` for a different amount of time. By keeping the jobs that simple we made sure that the tests would not interfere with other tasks running on the machines. This decision was taken in accordance with our supervisor at Ericsson.

To get reliable results the same job set should be used when comparing the different scheduling algorithms. If different job sets are used many more tests must be done to get closely reliable results. Real-world job sets derived from our use case are highly variable between runs. Running just compilation with no network requirements can be highly susceptible to caching, reading from disk and competing resource usage on the underlying hardware. However, using `sleep` meant that the only thing that affected the time it took to complete all the jobs was the scheduling policy used.

5.1.3 Hardware setup

For these tests, both the Jenkins master and the Jenkins slaves were run locally, on the same computer. The computer in question was a virtual machine that was set up using VMware, with access to 8 threads. The threads were mapped to the threads on an Intel(R) Xeon(R) CPU E5-2670 v3 2.30 GHz, with 12 cores and 24 threads, that was used in the server computer. The virtual machine was also assigned 32 GB of RAM.

The Jenkins master and the Jenkins slaves were run on the same computer and use the loopback network interface. This way the behavior of both the slaves and the master is identical to if they were connected by a network with a very small delay.

5.1.4 Local testing results

Two types of tests were carried out on a local Jenkins instance: single matrix and consecutive matrices. These are presented in separate sections below.

5.1.5 Single matrix

One of the potential situations that is interesting to look into is how well the system schedules a single matrix that enters the system, without having other matrices in the queue. Figure 5.1 describes the makespan for four different algorithms when increasing the number of slots. Figure 5.2 describes the average response time.

Figure 5.1 clearly shows LJF to be the best policy for achieving a minimal makespan. After 20 slots the makespan does not decrease, indicating that the longer jobs in the matrices set the bound on the makespan and hence LJF performs optimally. We observe that the difference between strategies decreases when the number of slots increases. Which is expected since we are approaching the point when there are enough slots for every job to execute immediately.

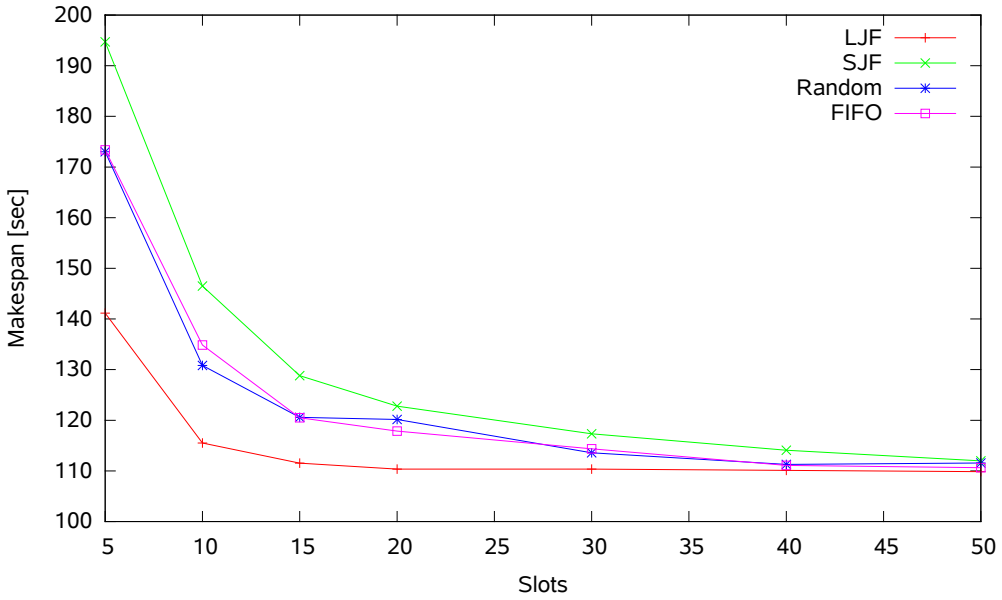

Figure 5.1: Comparison of the makespan achieved by the different algorithms when scheduling a single matrix.

Just as stated in section 3.2 to get a good average response time as many jobs as possible have to be executed as early as possible. Thus it is no surprise that SJF

produced the best scheduling when it comes to improving the response time, which can be seen in figure 5.2. Since LJF has an approach to scheduling that works the complete opposite of SJF, placing all the shortest jobs at the end of the queue, it will produce a very bad schedule when it comes to response time. However, the difference is more apparent the fewer slots there are in the system.

Both figure 5.2 and 5.1 support the statement from section 4.1 that FIFO and Random should perform equally. Since there is no other matrix in the system FIFO cannot make use of its scheduling feature and it performs as another form of Random.

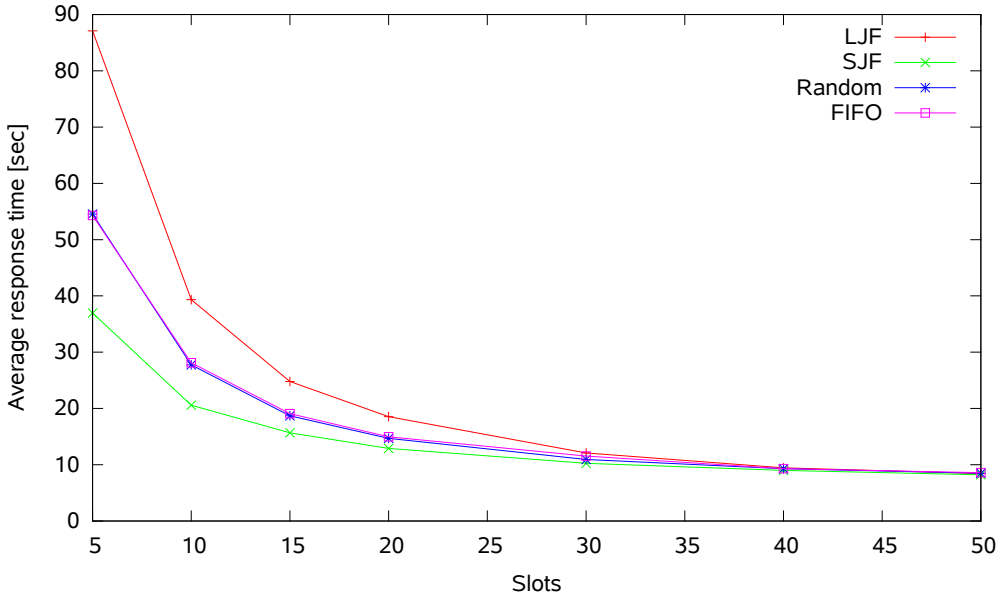

Figure 5.2: Comparison of the average response times between algorithms for a single matrix

5.1.6 Consecutive matrices

The total makespans for the group of matrices for the different strategies are shown in figure 5.3. Once again LJF is the best strategy for optimizing the makespan of a group of matrices. By adding the aging factor the makespan is increased a little.

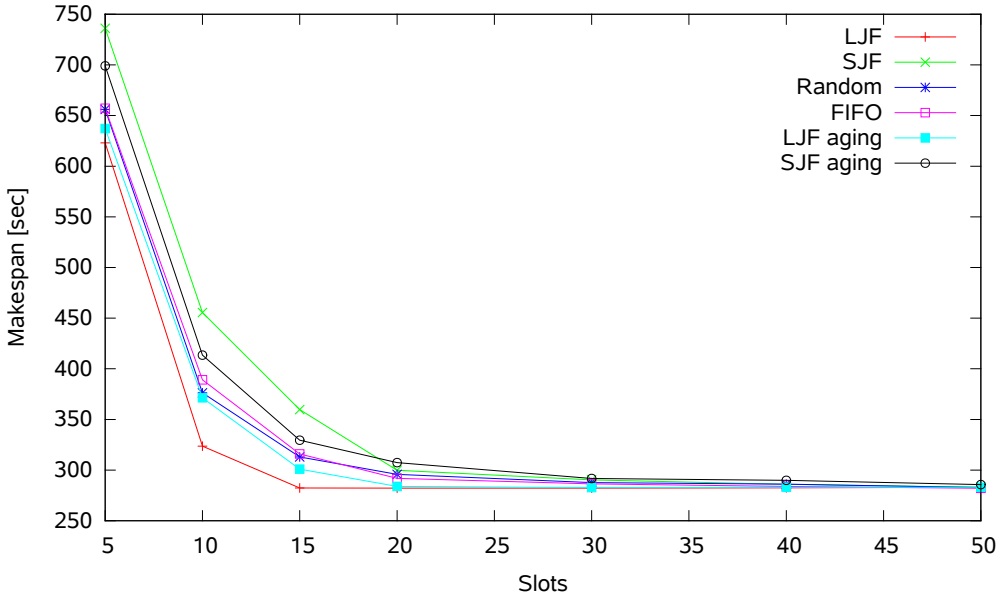

Figure 5.3: Makespans achieved by the different algorithms when five consecutive matrices were run with 30 second release time apart.

Figure 5.4 is showing the results from the measurement that we deem the most important in a CI setting, the average matrix response time in a group of matrices. LJF with an added aging factor is consistently the best strategy. SJF, which has been the worst strategy when it comes to the entire makespan, also performs well when an aging factor is added to it.

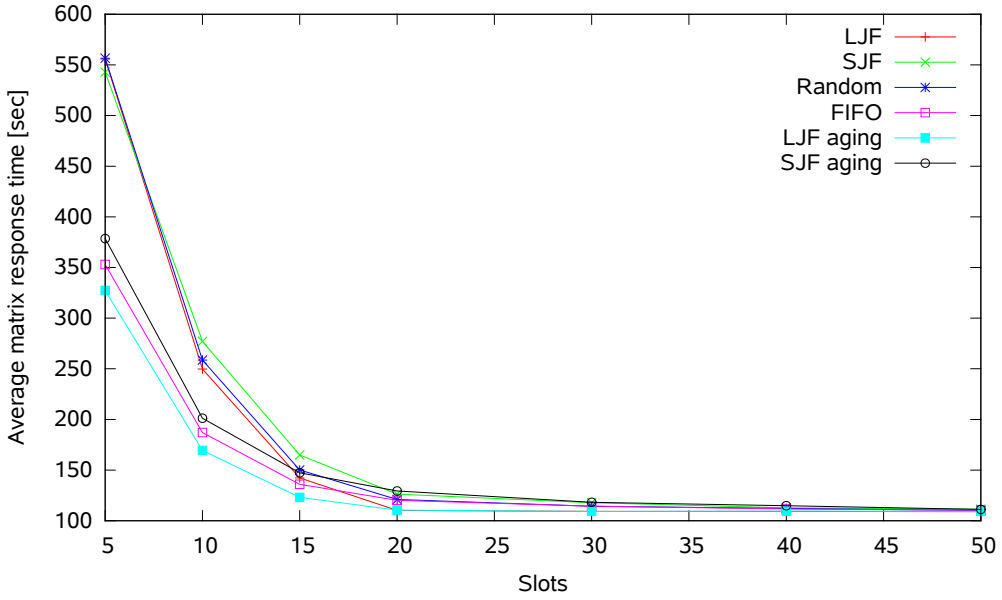

Figure 5.4: Average matrix response time achieved by the different algorithms when five matrices were run with 30 seconds release time apart.

The average response time for all individual jobs in the matrices is shown in figure 5.5. When it comes to the response time there clearly is no better way to schedule than using SJF. While using SJF with aging however, the strategy loses its main advantage since it does not produce a schedule with good response times.

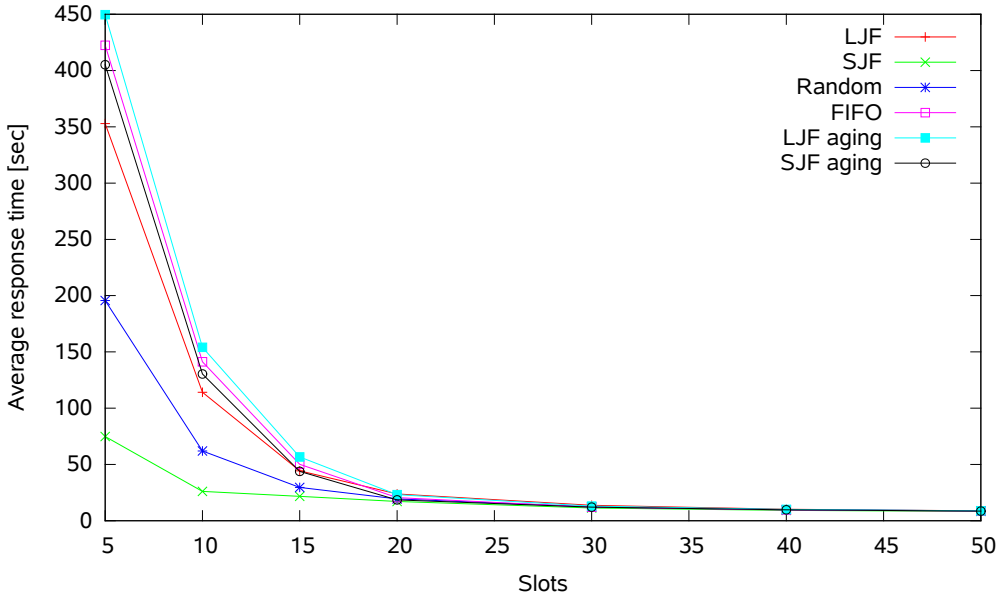

Figure 5.5: Average response time achieved by the different algorithms when five matrices were run with 30 seconds release time apart.

5.2 Load testing

As explained in section 3.3 it is not always positive to schedule too many heavy jobs simultaneously at the same slave. Dispatching as many jobs as possible from the queue will lead to more jobs being executed in parallel but it could also lead to the machine becoming overloaded and actually performing slower. We ~~will~~ investigate this issue by:

1. Finding the threshold when a machine will get overloaded.

2. Finding how much slowdown the overload will cause.

3. Modelling the risk of overload in the system to take this factor into consideration when evaluating scheduling policy.

To examine how bad overloading the system would affect the execution time of jobs on an overloaded machine we created a reference job that would perform a finite task while the system was overloaded. By comparing how long it took to complete the reference job we could see how bad the overload affected the machine.

5.2.1 Software setup

The reference job calculated the first 5000 decimals of π using the Unix command `bc`. It was run in parallel with one to ten heavy jobs, running during the entire duration of the reference job. The heavy jobs were all running `stress` on four threads, to mimic the system in our use case, to consume as much CPU power as possible. `Stress` is a deliberately simple workload generator that has been used in a lot of research projects to generate work [21]. For each amount of heavy jobs, one through ten, ten runs were done to ensure the results were reliable.

A Python script was used to compile all the test data into a CSV file to allow for easy use with the plotting software gnuplot to create graphs of the results.

5.2.2 Hardware setup

These tests were run on a virtual machine with eight threads, virtualized using VMware on a physical machine using an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40 GHz with 14 cores and 28 threads and 16GB of RAM.

5.2.3 Load testing results

Figure 5.6 shows how the execution time of the reference job increases when it has to run in parallel with heavy jobs. The figure clearly indicates that the machines may indeed become overloaded. An increase in execution time is noticeable as soon as the number of heavy job threads exceeds the number of threads on the machine. Given the setup described in 5.2.1 and 5.2.2 it is logical that this increase in execution time starts taking effect when the two heavy jobs are occupying the eight threads available at the machine. For the reference job to be able to execute, the system has to resort to context switching which will always lead to increased execution time. With ten heavy jobs running simultaneously at the same slave a severe slowdown can be observed. It takes about five times as long to finish a job compared to when it was run together with more lightweight jobs.

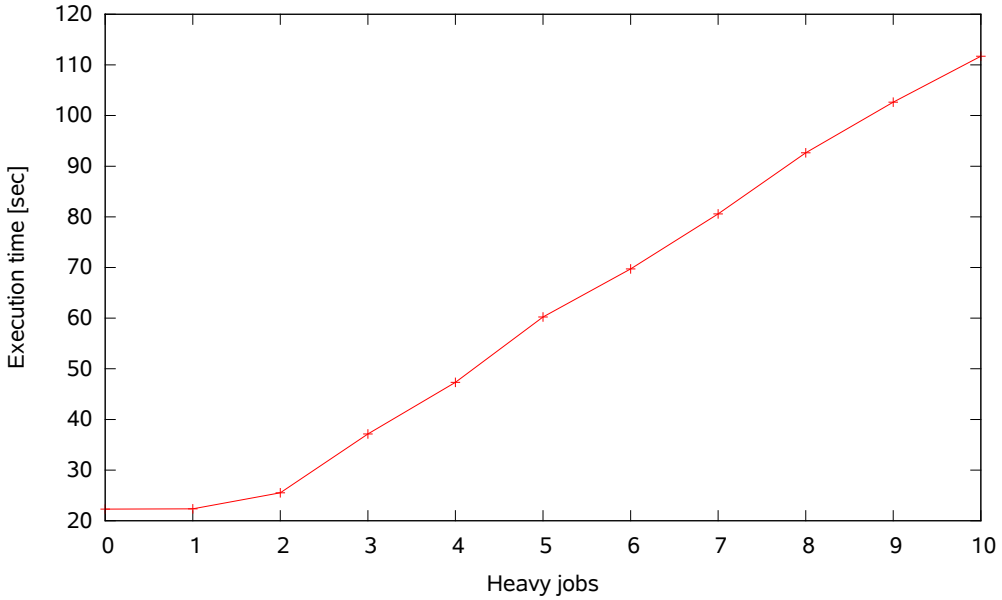

Figure 5.6: The execution time of the reference job when run together with heavy jobs. A considerable slowdown is observed when the amount of heavy jobs is increased.

Given this knowledge, it is clear that another area where it is possible to improve a CI system is to make sure to not schedule too many heavy jobs together. There are many different ways to avoid these situations. In section 7.1 we discuss one way to handle this problem.

From figure 5.6 we can view the slowdown factor as the ratio between the execution time when the reference job is running by it itself and together with a discrete number of heavy jobs. We can then define the set $slowdowns = \{f(x)/f(0)|x \in \mathbb{Z}_+\}$ where $f(x)$ is the function in the figure. Furthermore, this set was used as input to the models described in sections 4.2.2 and 4.3 which are evaluated below in section 5.3.

5.3 Simulation model testing

In addition to the tests described in section 5.2 and 5.1 that were run using a real Jenkins instance, we also ran tests on a simulation model of the system. The model calculates the makespan, average response time and average matrix response time of the schedules produced by the different algorithms.

This model enabled us to both run many more tests than when using a real Jenkins

instance and also simulate what would happen if we adjusted the number of slots on the slaves. The overload factor can also be taken into consideration. We simulated 1000 runs with the number of slots ranging from five to 15 on two slaves. A new set of jobs was generated for every run. A run includes all slot tests. The same job set is used for all different algorithms to properly compare them. This gave us new and more reliable data over the data from the Jenkins instance tests.

5.3.1 Test specification

The simulation model was written in Python 3. The model simulates how a real Jenkins instance would behave, with a load balancer, scheduler and the possibility to change the number of slaves and slots.

For the tests, we generated 1000 Pareto distributed matrices the same way as described in 5.1.1.

In the model, long jobs are also always regarded as heavy. To implement this we set a threshold so that the 20 percent longest jobs in the matrix are also heavy. We believe this is a reasonable assumption since to merit threading, a job can't have too minuscule resource requirements, otherwise, the overhead introduced would exceed the gain.

5.3.2 Simulation testing results

Figures 5.7, 5.8 and 5.9 show the average of the makespan, average response time and average matrix response time without overload of 1000 randomly generated Pareto distributed job matrices of size 100. Figures 5.10, 5.11 and 5.12 show the same but with overload added to the model. A matrix is scheduled onto two nodes with a varying number of slots according to five different algorithms. Since FIFO and Random are equal when all jobs enter with the same release time, they are equal in this model. We calculate the average and standard deviation for all the different policies.

When comparing the overload to the non-overload case we observe an increased execution time and some added noise. However, the general shape of the graphs are not changed, nor are their relations to each other. For all measurements, it is clear what algorithms are the best except in the average matrix response time case. Here the results are varying depending on the amount of parallelism.

A general tendency is that all polices show a diminishing gain when parallelism is increased. The exception to this is LJF when measured against makespan - shown in figure 5.10 - which increases when the number of nodes exceeds six.

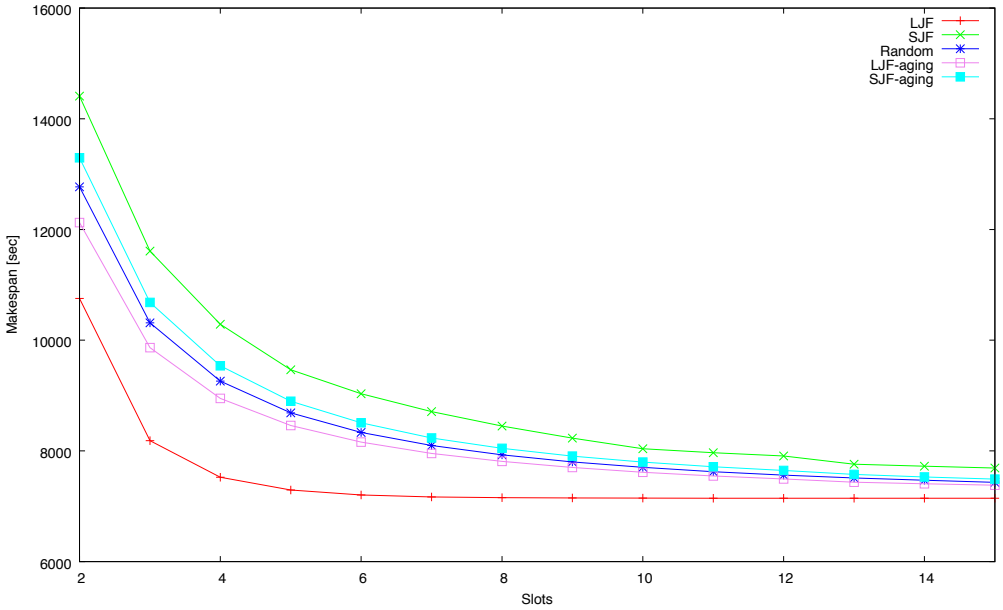

Figure 5.7: The makespan of the different algorithms without slowdown as the number of available slots increases.

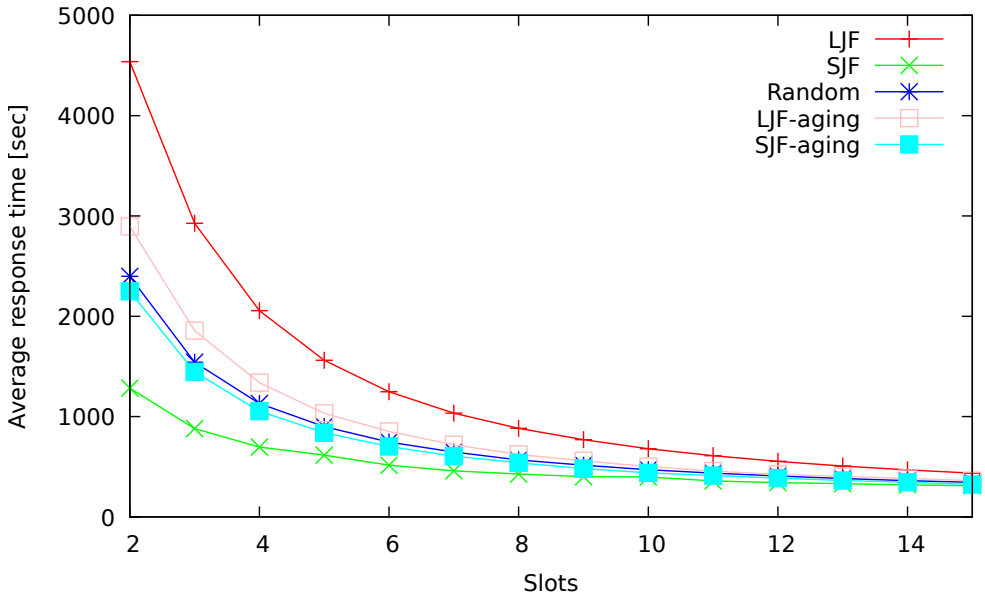

Figure 5.8: Average response time without slowdown

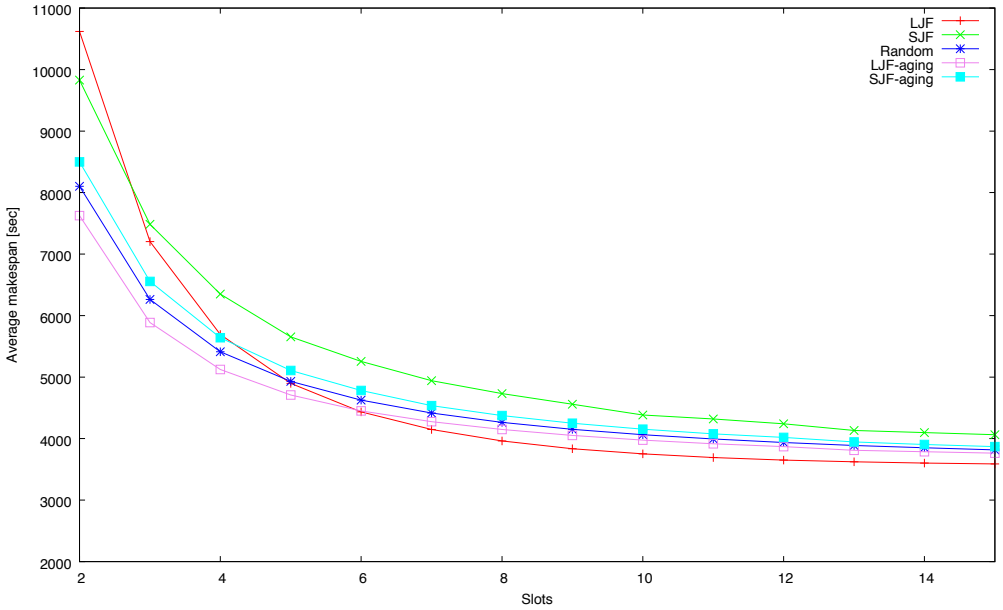

Figure 5.9: Average makespan without slowdown

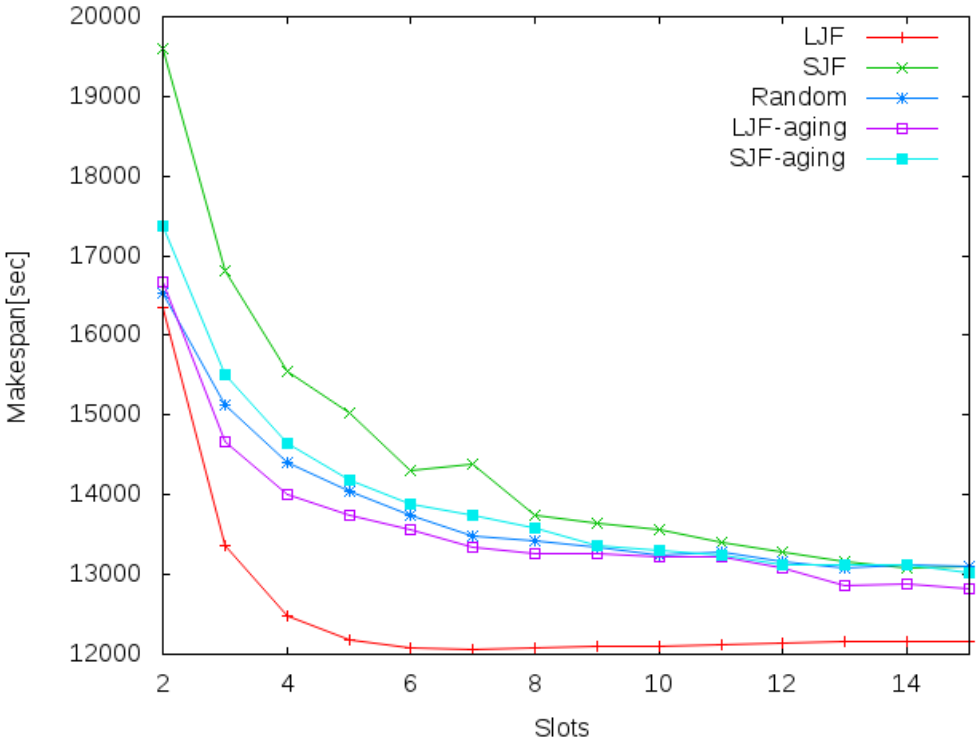

Figure 5.10: The makespan of the different algorithms with slowdown as the number of available slots is increased.

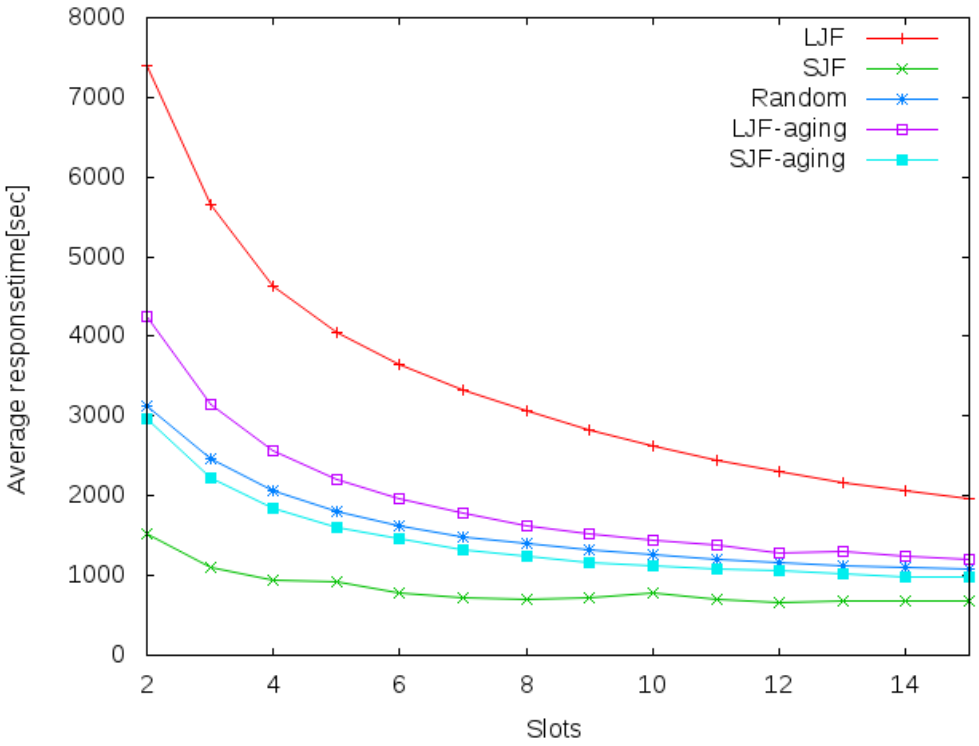

Figure 5.11: The average response time of the different algorithms with slowdown as the number of available slots is increased.

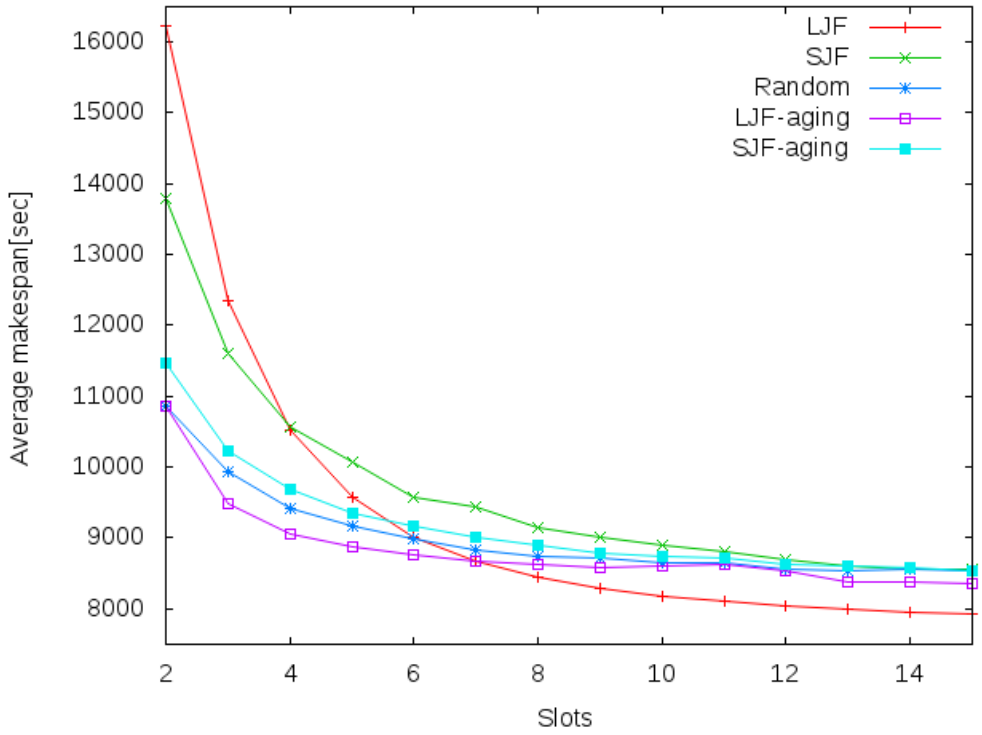

Figure 5.12: The average matrix response time of the different algorithms with slowdown as the number of available slots is increased.

Table 5.1 and table 5.2 show the standard deviation for all the algorithms tested in the simulation. The figures in table 5.1 are for the test cases without slowdown, i.e. the overload factor is set to zero. While the figures in table 5.2 were run with an overload factor to simulate the slowdown. From the figures, we observe that the standard deviation is higher in the slowdown case. We believe that this is explained by the noise in the simulations caused by the added overload.

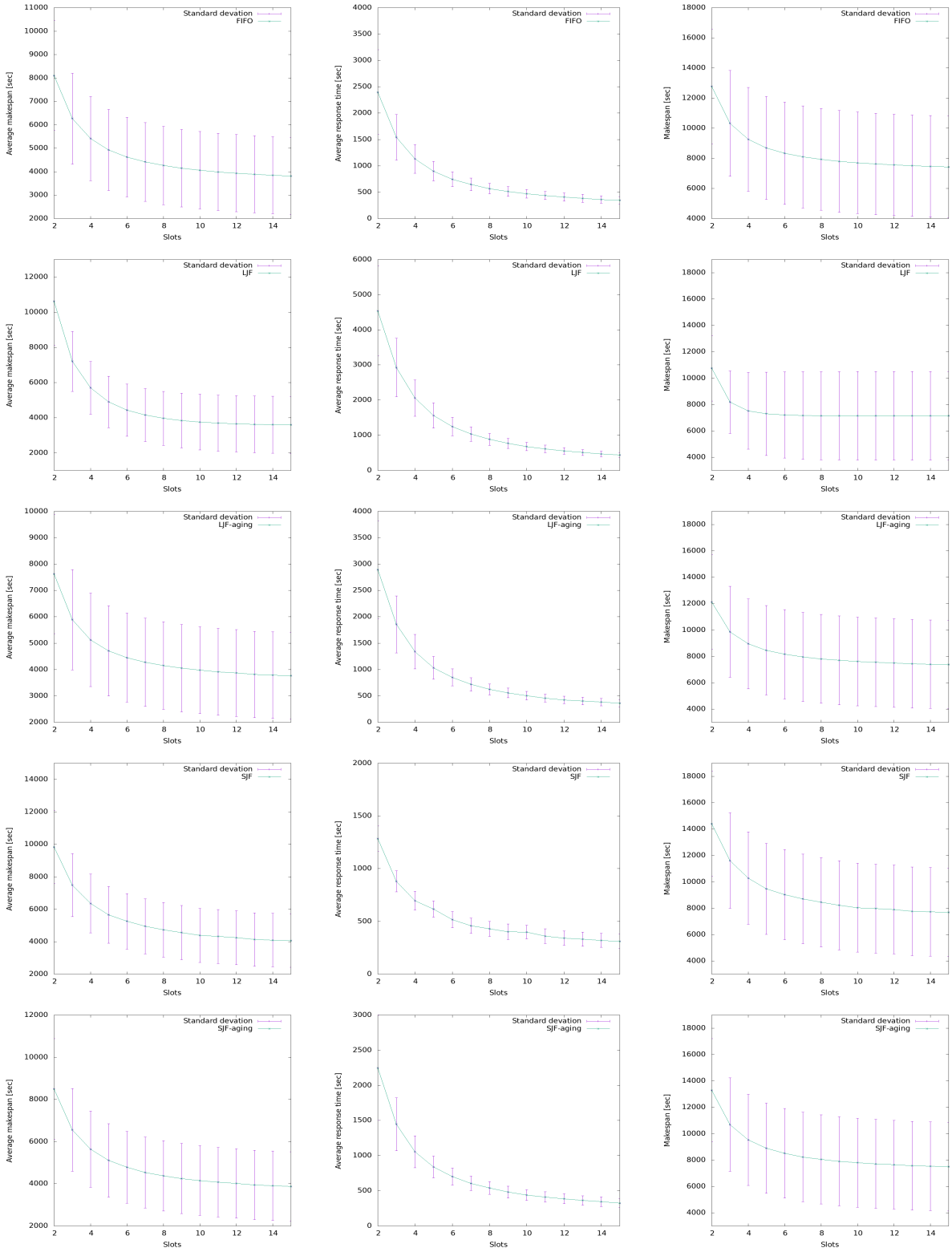

Table 5.1: Standard deviation for all metrics for the different algorithms tested with no slowdown taken into account.

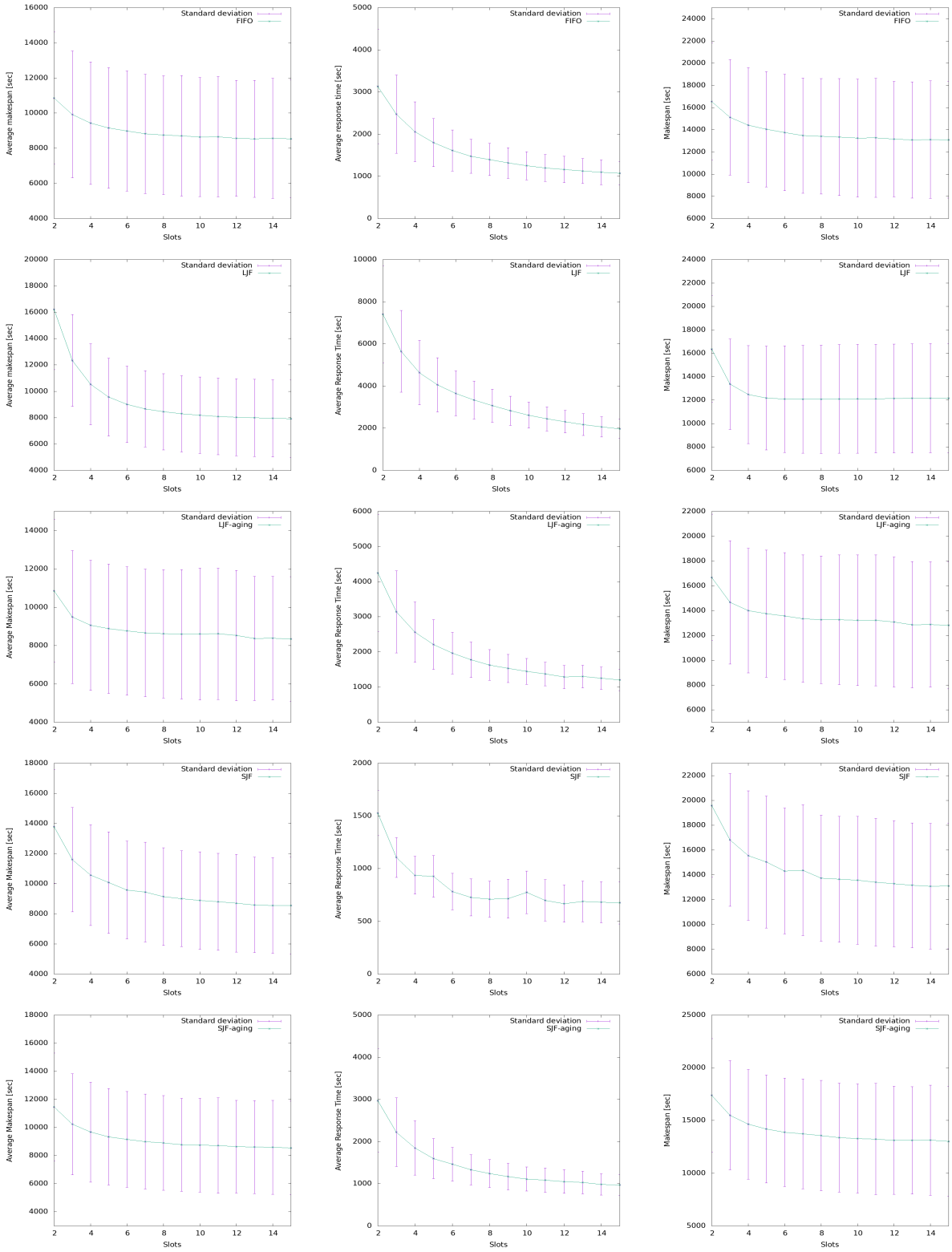

Table 5.2: Standard deviation for all metrics for the different algorithms tested with slowdown factor taken into account.

5.4 Summary of results

We summarize our results in a table below and compare the algorithms against each other with regard to the different measurements. We categorize the algorithms relative to each other as bad, medium, good or best. The overload and non-overload models did not change any of these relations and are therefore not shown as separate.

Algorithm	Avg. response time	Makespan	Avg. matrix response time
LJF	Bad	Best	Good
SJF	Best	Bad	Worst
Random	Medium	Medium	Bad
FIFO	Bad	Medium	Medium
LJF-aging	Bad	Medium	Good
SJF-aging	Bad	Bad	Medium

Result table: Summary of our main results.

6

Discussion

6.1 Local Jenkins Instance

This section contains our discussion about the results observed when testing both a single matrix and consecutive matrices.

6.1.1 Single matrix

As we can see it is not possible to optimize the system with regards to both response time and makespan. Since LJF and SJF work in the complete opposite way and they are the algorithms that produce the best makespan and response time, respectively. So it seems that the user has to make a choice about which measurement is the most important in their use case. Or by using FIFO or Random to achieve a system which is not geared towards minimizing any of the two measurements in particular, but lies somewhere between.

If the failure rate of jobs in the system is independent of their length a shorter average response time for each job makes a lot of sense. Especially if some kind of mechanism for "fail fast" is in place. As discussed in section 3.2 fail fast means that the system should stop all testing as soon as a failure has occurred and reported back to the appropriate developer. By operating in this way the developer will get a faster response when there is something that needs to be fixed, instead of having to wait for all the tests to complete. And since something needs to be changed anyway, which means running all tests again, it's better to abort them as soon as possible when finding an error. One caveat is that this reasoning holds under the assumption that the fail rate is independent of job length. This might not always be a reasonable assumption since longer jobs may be performing more tests, and thus have a higher fail rate.

6.1.2 Consecutive matrices

The consecutive matrices scenario is the most relevant for a larger CI system where there are several commits, that will trigger matrices, that enter the system.

LJF produces the best overall makespan and SJF the best average response time, as in the single matrix case.

We judge that average matrix response time is the most relevant measure since a matrix is generated by a commit and the commit is accepted when all tests have passed. If the overall makespan is prioritized over the average it means that a developer that was almost finished has to wait for an incoming matrix. Since a commit generates a matrix, minimizing the average matrix response time is equal to minimizing the average response time of a commit, from the developer's point of view. This means that the time for the developer is decreased, while also guaranteeing fairness.

As can be seen, LJF-aging is the scheduling algorithm that is best for minimizing the average matrix response time. FIFO is quite good since no individual jobs from a matrix get left behind and are allowed to increase the average. When studying the single matrix case we can observe that LJF performs the best. FIFO on an inter matrix level but sorting the jobs inside a matrix according to LJF, which in effect is LJF-aging, naturally produces the best average matrix response time.

When looking at the average matrix response time both LJF and SJF, without aging, perform as bad as the Random strategy. This further confirms the fact that being able to ensure a fair scheduling policy, with no starvation, is very important to also achieve good execution times.

6.2 Simulation model

Considering the makespan, both with and without slowdown all polices but LJF show a diminishing gain when parallelism is increased. LJF does not change its makespan when the number of slots exceeds six, indicating that this policy already achieves an optimal solution for many of the job matrices tested. SJF produces the worst makespan, which is not surprising since it actively schedules in the opposite way compared to LJF. Random is better and the aging modified policies produce makespans between Random and their non-aging modified counterparts.

Figure 5.10 shows the makespan of the same job matrices as in figure 5.7, this time with slowdown added in the model. The improved makespan gained by increased parallelism is counteracted by the increasing risk of overload when the number of slots increases. For Random the improvement in makespan is 64% in figure 5.7

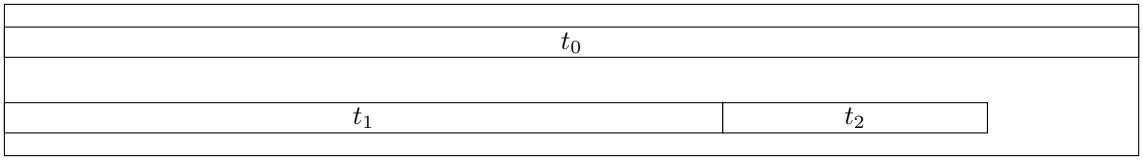

Figure 6.1: Execution of jobs in case 1.

while being only 22% in figure 5.10. LJF, on the other hand, shows an increase in makespan when the number of slots exceeds six. This is not surprising since it was not affected by the increased parallelism but still suffers from overload. If this assumption is correct the same effect will be seen for the other algorithms as well, once they have reached the point where additional slots is not reducing the execution time. The notion that the gain from parallelizing the system to a greater extent is diminished is also supported by the model in section 4.3. In figure 4.1 it is clearly shown that as the number of available slots goes up, the slowdown factor increases with it. And because of this, the improvement to makespan in figure 5.10 is mediocre compared to the improvement achieved in figure 5.7, for LJF it is even slightly negative.

LJF is an interesting case. Although the increased parallelism increases the risk and severity of overload - since more heavy jobs are simultaneously running - it also decreases the time of the overload. Imagine a scenario where there are three heavy jobs with execution times two, one and one. If these jobs are scheduled on two slots, they will cause overload during the entire execution time. If they are scheduled on three slots instead, they will only cause overload during half the execution time, but this overload will be more severe. As long as the overload factor is increasing linearly, proportionally to the number of heavy jobs on a machine, a simple proof can show that these two scenarios will result in the same time consumption. We describe the overload with the help of a constant k that is multiplied with the number of heavy jobs n. We can then integrate this time consumption over their original execution time along the axis t. The two cases are depicted in picture 6.1 and 6.2. We have three jobs 0,1,2 with their respective length: $t_0 > t_1 > t_2$ and $t_0 > t_1 + t_2$. The method of the proof will be to divide the executions into regions and sum their respective time consumption and then show that they are the same for case one and two.

The first case can be divided into two regions. The first region is from 0 to t_1+t_2 In this case we have $n + 1$ heavy job running. The second case is the rest of the duration, from $t_1 + t_2$ to t0. In this case, n jobs are running.

$$\int_0^{t_1+t_2} (n+1)k dt = (n+1)k(t_1+t_2) \tag{6.1}$$

$$\int_{t_1+t_2}^{t_0} nk dt = nk(t_0 - t_1 - t_2) \tag{6.2}$$

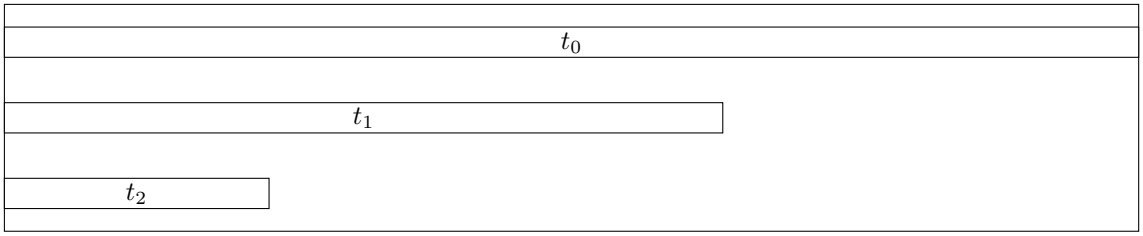

Figure 6.2: Execution of jobs in case 2.

Adding 6.1 to 6.2 results in:

$$k(t_0 n + t_1 + t_2) \tag{6.3}$$

The second case can be divided into three regions. The first region is from 0 to $t2$, when $n + 2$ jobs are running. The second region is from $t2$ to $t1$ when $n + 1$ jobs are running and the third one is from $t1$ to $t0$ when n jobs are running.

$$\int_0^{t_2} (n+2)kdt = (n+2)k(t_2) \tag{6.4}$$

$$\int_{t_2}^{t_1} (n+1)kdt = (n+1)k(t_1 - t_2) \tag{6.5}$$

$$\int_{t1}^{t_0} nkdt = nk(t_0 - t_1) \tag{6.6}$$

Adding 6.4 to 6.5 and 6.6 results in:

$$k(t_0 n + t_1 + t_2) \tag{6.7}$$

6.3 and 6.7 are equal and hence our proof is finished.

That the increase in makespan for LJF is so very slight when slots are increased can be explained by the above effect. It is most prominent for this algorithm since all heavy jobs are clustered in the beginning regardless of the number of slots. For other algorithms, we hypothesize that the risk of overload simply increases when the number of slot increases which would explain their greater slowdown. However, the overload factor is not linear, as there is no overload when scheduling two heavy jobs on the same machine. This means that it is possible to run two heavy jobs on the same machine without it affecting the execution times of the heavy jobs or other non-heavy jobs on the machine. More slots should mean that this overload situation becomes less common. This could explain the small increase in makespan we however still observe for LJF.

For the average response time, the results indicate that SJF is the best policy, as expected.

For the average makespan, LJF-aging is the best policy when a few slots are available but LJF becomes the better policy when the number of slots increases. That LJF-aging is good is not surprising since the measure average makespan punishes non-fair behavior which LJF-aging avoid. However, when parallelism increases LJF gets better. This can be explained by the fact that is is the job that finishes last in the matrix that determines the average makespan. The effect of letting short jobs from earlier matrices wait on longer jobs from later matrices does not affect the average makespan if the short jobs none the less get to finish earlier than the longer ones in their matrix. This situation becomes more common when the amount of parallelism is increased since then there are enough slots such that the short jobs can get to finish earlier than the longer ones. Overload does not change the behavior in any significant way.

6.3 Load testing

Given how much of a slowdown that was observed in section 5.2.3 is obviously very important to try to minimize its number of occurrences. It does not take very many heavy jobs running in parallel to completely erase the improvements achieved by any of the scheduling algorithms. None of the tested algorithms actively tries to avoid this situation. Furthermore, it is hard to know how much overload can be avoided by changing the scheduling policy since we cannot compare it to an optimal schedule. We only compare the schedules produced by the tested algorithms to each other. Nonetheless, overload needs to be taken into consideration when deciding which scheduling policy and load balancer to use, and how they influence each other. In some cases, it might be worth it to reduce the amount of slots available to reduce the risk of overload.

6.4 Future work

Below we propose some future work that could be carried out to further investigate and possibly improve scheduling for a CI server.

6.4.1 Other job set distributions

To further examine how the algorithms perform it could be interesting to analyze them using another job set. Several different job sets have been used to evaluate

them already, but they have all followed a Pareto distribution. There are many other distributions which are commonly seen that could be tested. The jobs could, for example, follow a normal distribution, which is a very common way for data to present itself.

6.4.2 Heterogeneous machines

In all our experiments we assumed that the machines were all homogeneous, i.e. they all used the same hardware. In a realistic scenario, this is not always true. It would be interesting to test the algorithms on a system where the machines were not all built in the same way, since this could greatly influence the results. It would mean that it is very important to not only schedule a long job first but equally important to place it on the machine that can execute it as fast as possible. If that machine is busy, a decision has to be made about waiting for that machine or choosing another, slower machine. These types of decisions can easily become very complex.

6.4.3 Memory bus

There are several resources in a modern computer that can suffer from slow down when they get overloaded, apart from the CPU. One such resource that could be investigated further is the memory bus. Accessing the memory is a very common action in a computer, which will always need to make use of the memory bus. If too many jobs were to use the bus at the same time it is possible that similar behavior could be observed as when too many processes try to use the CPU simultaneously.

6.4.4 Simulation model

The simulation model can be extended to also include failures and a "fail fast" policy such that matrices are aborted if one of the jobs fail. How the algorithms perform under different fail rates could then be tested. Lastly, our slowdown factor is derived from measurements taken in a live system. Instead, the model could be extended so that it could simulate varying slowdown factors. With a different slowdown factor, the algorithms may be affected differently. The best algorithm for a very small slowdown factor might not perform as well when overloading the system means higher overhead.

6.4.5 Real world testing

An interesting research setup would be to test the scheduling algorithms in a real world setup were we gather a large set of actual CI jobs. These jobs could then be scheduled again by the different algorithms and the execution time measured. The technical problem with this approach is to gather representative and good enough live data. If this can be done we believe that the result from this type of testing would be the most reliable way of evaluating the different algorithms.

7

Related Work

The literature on scheduling is vast, and problems can be specified in a number of ways. The book *Scheduling: Theory, Algorithms, and Systems* provides a general theoretical background, description of common algorithms and proofs [5]. It offers an entry point to be able to classify what kind of scheduling problem one is concerned with. Scheduling in this sense is not limited to programs running on a computer but can as well be e.g. runways at airports or crews at a construction site. However the same principles are applicable. The book makes a separation between deterministic and stochastic models. Deterministic models are models where the properties of the system have known values, for example the execution times of the jobs to be scheduled are known. For the stochastic models on the other hand, only the probability distribution of the properties are assumed. The literature for stochastic models is less extensive. A further distinction is then made between parallel- and single machines. The system model at hand for this thesis is a system of parallel machines. One of the measures we investigate in the thesis is the makespan. To find an optimal schedule with regards to makespan on parallel machines, without preemptions and in a deterministic model is an NP-hard problem [5]. The author proves this by reducing the problem to the partition problem which is known to be NP-complete.

In the scheduling literature a classic, often cited paper about this problem is *Bounds on Multiprocessing Timing Anomalies* [22]. The paper is short but dense and presents a mathematical proof for the worst case bounds for the random algorithm and for longest job first. Longest job first is shown to have the better worst case bound.

Scheduling online mixed-parallel workflows of rigid tasks in heterogeneous multi-cluster environments evaluates a number of algorithms for scheduling mixed-parallel workflows - which means a mix of task parallelism and data parallelism with precedence constraints between them [23]. Data parallelism means that the same task can be executed concurrently on different cores since the data can be split into independent sections. Task parallelism means that different tasks can run on different cores. The system model differs compared to ours since we only have to consider task parallelism. The input to be scheduled is defined to be acyclic graphs where edges between nodes represent dependencies between tasks. However, the experimental

approach is related to the one in this thesis. Rather than giving a formal proof for the worst case bounds on their algorithms a simulation model of a scheduling system is developed. Four algorithms are suggested and tested against the measure "average turnaround time". Where turnaround time is defined as the total elapsed time for a collection of interdependent tasks including the waiting time incurred from any precedence constraints. This measure is similar but not identical to our measure "average matrix response time". The difference being that we have no precedence constraints between tasks in a matrix.

Mixed Data-Parallel Scheduling for Distributed Continuous Integration is specifically focusing on scheduling in a CI context [24]. The system model - however - differs from the one defined in this thesis. The model is similar to the above mentioned article, tasks are modeled to have dependencies in between each other and represent the dependencies between builds. The heuristic algorithms tested are based on this property as well as simulations to approximate finishing times. This differs from our system model where no precedence constraints exist between jobs and the execution times of jobs are assumed *a priori*.

The doctoral thesis *A Slowdown Prediction Method to Improve Memory Aware Scheduling* discusses the issue of co-scheduling tasks for memory-intensive applications and ways to combat this issue [15]. This thesis focuses on computation-intensive applications and the problems of co-scheduling these.

7.1 Dynamic load index

One way to try to actively avoid bad co-scheduling would be to have a feedback mechanism where the system will continually keep track of how high of a load there is on every slave and report this back to the central CI server. This information can then be used to schedule the jobs without causing an overload situation on any node. One way of implementing this approach would be to keep two separate queues, one for heavy jobs and one for lighter jobs. Once a slave reports back to the master that it is close to overloading, the system starts scheduling jobs from the lighter queue for that slave. When the slave once again reports that it can handle a heavy job the system goes back primarily scheduling heavy jobs for that slave again.

In [25] Sharifian et al. propose an algorithm with a similar approach. They apply the thought of a feedback system and groupings of different jobs to a web server. Their algorithm produces significant time improvements but introduces additional overhead into the system due to the complex nature of it.

The major drawback of using a feedback mechanism to solve the dynamic load index problem is that it makes the system much more complex. First of all two queues need to be maintained and the state of every slave has to be continually updated. The communication between the master and the slave also needs to become much

more robust. A way to approximate both the resource usage on slaves and resource requirements must also be implemented. On the other hand if, long jobs are also heavy, as we have assumed in our model, this classification becomes easier.

An additional aspect of the complexity is that dependencies between the load balancer and the scheduler are introduced since the load balancer needs to keep track of which of the two queues held by the scheduler to dispatch jobs from. Making these two components that should work separately dependent on each other is another reason why it is hard to justify this approach.

8

Conclusion

The thesis investigates how to improve a CI server by choosing an appropriate scheduling algorithm. We conclude that average matrix response time - a measurement introduced in this thesis - is the most relevant measure for our system model. Furthermore, we believe that our system model corresponds to a common CI architecture. The job sets used in our empirical evaluation were set to follow a Pareto distribution. This is based on observations from our use case.

For the objective of minimizing the average matrix response time LJF with an added aging factor is the best algorithm according to our measurement on a Jenkins instance.

To be able to take machine overload into consideration and make more extensive testing a simulation model was developed. When considering both the overload and non-overload case, LJF produces the best makespan according to our simulation model, LJF and LJF-aging produce the best average matrix response time and SJF produces the best average response time.

An overloaded system diminishes the gain of parallelism. Both our mathematical model and our simulation model indicates this. This is true for all algorithms and measurements. Nearly every algorithm improves for all measurements when increasing the job parallelism but the gain is diminished due to the increased risk of overload. The exception is LJF which early on achieves an optimal makespan and thus the makespan will only increase due to the overload factor caused by the increased parallelism.

Bibliography

[1] [Online, accessed May 29, 2018]. URL: https://pngimg.com/download/41163.

[2] Martin Fowler and Matthew Foemmel. "Continuous integration". In: *ThoughtWorks http://www. thoughtworks. com/Continuous Integration. pdf* 122 (2006), p. 14.

[3] *Ericsson Company Facts*. URL: https://www.ericsson.com/en/about-us/company-facts (visited on 03/13/2018).

[4] Martin Heller. *What is Jenkins?* URL: https://www.infoworld.com/article/3239666/devops/what-is-jenkins-the-ci-server-explained.html (visited on 05/17/2018).

[5] Michael L. Pinedo. *Parallel Machine Models (Deterministic)*. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-26580-3. DOI: 10.1007/978-3-319-26580-3_5. URL: https://doi.org/10.1007/978-3-319-26580-3_5.

[6] Chandra Chekuri et al. "Multi-processor scheduling to minimize flow time with ε resource augmentation". In: *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM. 2004, pp. 363–372.

[7] Susanne Albers. "Better bounds for online scheduling". In: *SIAM Journal on Computing* 29.2 (1999), pp. 459–473. ISSN: 00975397.

[8] *Jenkins Press Information*. URL: https://jenkins.io/press/#about-jenkins (visited on 03/05/2018).

[9] Nikhil Bansal. "Algorithms for flow time scheduling". PhD thesis. School of Computer Science, Carnegie Mellon University, 2003.

[10] Sandeep Sivanandan. "Fail Fast-Fail Often: Enhancing Agile Methodology using Dynamic Regression, Code Bisector and Code Quality in Continuous Integration (CI)". In: *arXiv preprint arXiv:1506.08725* (2015).

[11] Stephan Mertens. "The easiest hard problem: Number partitioning". In: *Computational Complexity and Statistical Physics* 125.2 (2006), pp. 125–139.

[12] Joacim Andersson and Pontus Andersson. "Increasing the Performance of a Continuous Integration Server". MA thesis. Chalmers University of Technology, June 2016.

[13] Chuanpeng Li, Chen Ding, and Kai Shen. "Quantifying the cost of context switch". In: *Proceedings of the 2007 workshop on Experimental computer science*. ACM. 2007, p. 2.

[14] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[15] Andreas De Blanche. "A Slowdown Prediction Method to Improve Memory Aware Scheduling". PhD thesis. Chalmers University of Technology, 2016.

[16] *GNU Make*. URL: https://www.gnu.org/software/make/ (visited on 04/30/2018).

[17] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014. Chap. 6, p. 279.

[18] *Priority Sorter*. URL: https://plugins.jenkins.io/PrioritySorter (visited on 04/19/2018).

[19] Mitsuo Gen and Runwei Cheng. *Genetic algorithms and engineering optimization*. Vol. 7. John Wiley & Sons, 2000.

[20] *NumPy*. URL: http://www.numpy.org/ (visited on 05/14/2018).

[21] *stress project page*. URL: https://people.seas.harvard.edu/~apw/stress/ (visited on 05/14/2018).

[22] CHEN PENG et al. "Timing-Anomaly Free Dynamic Scheduling of Conditional DAG Tasks on Multi-Core Systems." In: *ACM Transactions on Embedded Computing Systems* 18.5s (2019), p. 1. ISSN: 15399087. URL: https://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edb&AN=139108206&site=eds-live&scope=site&custid=s3911979&authtype=sso&group=main&profile=eds.

[23] Yi-Rong Wang, Kuo-Chan Huang, and Feng-Jian Wang. "Scheduling online mixed-parallel workflows of rigid tasks in heterogeneous multi-cluster environments". In: *Future Generation Computer Systems* 60 (2016), pp. 35–47.

[24] O. Beaumont et al. "Mixed Data-Parallel Scheduling for Distributed Continuous Integration". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. May 2012, pp. 91–98. DOI: 10.1109/IPDPSW.2012.7.

[25] Saeed Sharifian, Seyed A Motamedi, and Mohammad K Akbari. "A content-based load balancing algorithm with admission control for cluster web servers". In: *Future Generation Computer Systems* 24.8 (2008), pp. 775–787.