

# Increasing the Performance of a Continuous Integration Server

Optimisations by Load Balancing and Node Configurations

Master's thesis in  
Computer System and Network  
Computer Science – algorithms, languages and logic

Joacim Andersson, Pontus Andersson



MASTER'S THESIS 2016

# Increasing the Performance of a Continuous Integration Server

Optimisations by Load Balancing and Node Configurations

JOACIM ANDERSSON  
PONTUS ANDERSSON



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2016

Increasing the Performance of a Continuous Integration Server  
Optimisations by Load Balancing and Node Configurations  
JOACIM ANDERSSON  
PONTUS ANDERSSON

© JOACIM ANDERSSON, June 2016.

© PONTUS ANDERSSON, June 2016.

Supervisors: Birgit Grohe, Department of Computer Science and Engineering.  
K. V. S. Prasad, Department of Computer Science and Engineering.  
Richard Norling, Ericsson  
Examiner: Alejandro Russo, Department of Computer Science and Engineering

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

Cover: "Increasing the Performance of a Continuous Integration Server". A modification of the official Jenkins logo. The official Jenkins logo is created by Charles Lowell and licensed under CC BY-SA 3.0.

Gothenburg, Sweden 2016

Increasing the Performance of a Continuous Integration Server  
Optimisations by Load Balancing and Node Configurations  
Joacim Andersson, Pontus Andersson  
Department of Computer Science and Engineering  
Chalmers University of Technology and University of Gothenburg

## Abstract

Continuous Integration (CI) is a software engineering practise where developers integrate, test, and build the system many times a day. The main objective with CI is to provide immediate feedback to developers after code changes, which requires an optimised CI server.

This thesis aims to improve the time between the start and finish of a sequence of jobs in three ways: statistically analysing the best node configurations regarding the number of executors, dynamically assigning executors, and developing an improved load balancer. The load balancing problem is a more general problem that may be applicable for servers other than CI.

The most suitable number of executors was found to be around 1.0-1.4 times the number of cores on the node, with the most significant improvement when the number of executors was equal to or above the number of cores.

Dynamically assigning executors turned out to be an interesting substitute to static allocation; it reduced the load of the nodes and compensated better for changing workloads, at the cost of a increased execution time by approximately 5%.

The improved load balancer was superior to the default load balancer in both execution time and load balancing according to our experiments.

The results of this thesis can have a great effect on CI servers by improving the feedback time for developers and reducing the risk of overloading the nodes.

Keywords: Continuous Integration, Node Configurations, Load Balancing, Optimisation, Load Indices, Jenkins, Executors.



# Acknowledgements

This thesis was conducted for and made possible by Ericsson AB. We would like to thank team Watson at Ericsson who provided insight and expertise when it came to the Continuous Integration server.

Furthermore we would like to thank and express our gratitude to our supervisors Birgit Grohe & K V S Prasad, at Chalmers University of Technology, and our company supervisor Richard Norling, for the useful information, comments, and remarks. This thesis would not be possible without their help.

Joacim Andersson, Pontus Andersson, Gothenburg, June 2016





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Purpose . . . . .	2
1.2	Related Work . . . . .	3
1.2.1	Continuous Integration Node Configurations . . . . .	3
1.2.2	Load Balancing . . . . .	3
1.3	Limitations . . . . .	4
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Unix Load Average . . . . .	5
2.2	PID Regulator . . . . .	5
2.3	Load Balancing . . . . .	6
2.4	Default Jenkins Load Balancer . . . . .	8
<b>3</b>	<b>Methods</b>	<b>11</b>
3.1	Distributed System Specifications . . . . .	11
3.2	Job Configurations . . . . .	11
3.3	Software . . . . .	12
3.4	General Test Specifications . . . . .	13
3.5	Static Executor Allocation . . . . .	13
3.6	Dynamic Executor Regulator . . . . .	14
3.7	Load Balancer . . . . .	15
3.7.1	Load Balancer Test Specifications . . . . .	15
3.7.2	Computational Power . . . . .	16
3.7.3	Load Indices Design . . . . .	17
<b>4</b>	<b>Results</b>	<b>19</b>
4.1	Static Executor Allocation . . . . .	19
4.1.1	Individual Job Performance . . . . .	21
4.2	Dynamic Executor Regulator . . . . .	22
4.3	Load Balancer . . . . .	23
<b>5</b>	<b>Discussion</b>	<b>25</b>
5.1	Experimental Uncertainty . . . . .	25
5.2	Node Overload Problems . . . . .	25
5.3	Static Executor Allocation . . . . .	26
5.4	Dynamic Executor Regulator . . . . .	26
5.4.1	Decentralised Load Balancing Effect . . . . .	27

## Contents

---

5.5	Load Balancer . . . . .	28
5.5.1	Job Distribution for Larger Systems . . . . .	28
5.5.2	Instantaneous and Smoothed Load Indices . . . . .	29
5.5.3	Overhead and Time Complexity . . . . .	31
5.5.4	Computational Power Score Indices . . . . .	31
5.6	Ethical Aspects . . . . .	32
5.7	Future Development . . . . .	32
<b>6</b>	<b>Conclusion</b>	<b>34</b>
	<b>Bibliography</b>	<b>35</b>

# 1

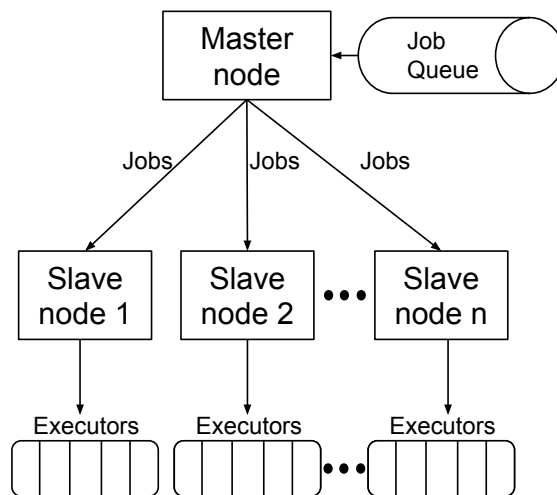
## Introduction

Continuous Integration (CI) is a software engineering practise where developers integrate their code changes to the system several times a day. The code changes are automatically built and tested to ensure that the system behaves as expected. CI has been part of previous research and is state-of-the-art when it comes to development practises, and is an important part in making companies able to compete in the marketplace. CI was first proposed in 1991 by Grady Booch [1] and is part of a software development methodology called Extreme Programming [2].

The CI server used in this thesis is called Jenkins, and it is the most widely used CI server today [3] [4]. Jenkins uses the “master/slave” model of communication. The master node consists of a basic Jenkins installation which distributes the incoming workload to multiple slave nodes (simply referred to as "nodes"). Each node consists of a lightweight program called a slave-agent [5]. While it is possible for a physical machine to have multiple nodes on it, the system in this thesis has one node per physical machine. Each node has a number of executors. One job instance can be executed on each executor; it is a way to limit the number of jobs that can run simultaneously on a node. A job is a runnable task, it could for instance be a test or compile program. See Figure 1.1 for an overview of the Jenkins master/slave structure. For certain jobs, a workspace needs to be created and necessary data needs to be downloaded the first time it is run, otherwise it cannot successfully execute.

The main objective when trying to improve a CI server is to decrease the time it takes for developers to get feedback on their code changes. This is primarily done by speeding up the build process. A CI server usually follows a “ten minute rule”, which means that a build should aim to complete within ten minutes [6].

This is where the main focus of our project comes in, optimising the node configurations, as well as how the CI server load balances the jobs, so the build time can be improved. Load balancing has been part of previous research and includes a collection of engineering challenges. Load balancing describes how to distribute a finite number of jobs to a finite number of nodes in order to lessen the workload of individual nodes, increase throughput, and minimise execution times [7].



**Figure 1.1:** Jenkins master/slave mode visualisation

## 1.1 Purpose

The main purpose of this thesis is to increase the performance of a CI server and decrease the time between the start and finish of a job. This will be done by improving the node configurations and the load balancer of the system. The project is divided into three parts:

- Improve the Jenkins node configurations by statistically finding a suitable number of static executors for a node, depending on the node specifications.
- Assign the number of executors dynamically during runtime, depending on the current load of the node. This will be done by regulating the number of executors to keep a chosen load index at a desired set point.
- Research and develop a better load balancer. The problem consists of non-preemptive load balancing in a heterogeneous system.

The project was proposed by Ericsson AB, a company focused on telecommunication equipment, software, and services [8]. Ericsson has for some time focused on transforming their production strategy from a plan driven organisation to a more agile one [9]. Today, a lot of teams within Ericsson use a lean and agile software development strategy with CI as part of their production practise. It was discussed with Ericsson that improvements could be made when it comes to the node configurations, as well as the load balancer on the CI server.

## 1.2 Related Work

This section surveys related work in the field of CI node configurations and load balancing.

### 1.2.1 Continuous Integration Node Configurations

Several recent books and papers have been published that study various aspects of a CI server. While they contain valuable information, they do not go into much detail when it comes to the node configurations.

In a paper written by Seppälä [10], the number of executors was set to one executor per processor core. Having the number of executors close to the number of cores seems like an intuitive configuration.

### 1.2.2 Load Balancing

Bosque et al. [11] proposed a load balancing algorithm for a non-preemptive system with heterogeneous nodes, similar to the system in this thesis. Heterogeneous nodes means that the nodes have different specifications and computing power. Non-preemptive means that once a job has started, it cannot be moved to another node. The paper describes a load index based on the current workload of a node, in relation to the total computing power of the node, in comparison to the other nodes.

More specifically, the index is defined as

$$Load_{Index} = \frac{P_i}{P_{max}} \cdot \frac{\#Cores}{\#Tasks + 1}$$

where  $P_i$  is the computational power of node  $i$ , and  $P_{max}$  is the computation power of the most powerful node in the cluster.  $\#Tasks$  and  $\#Cores$  are the current number of tasks and cores on the node. Bosque et al. [11] presents experimental results and draws the conclusion that a large improvement is possible with the algorithm.

Another problem specification of the load balancing problem is that the jobs are unpredictable when it comes to the workload required. This situation has been a part of previous research by for instance Suguna & Barani [12] and Daraghmi & Yuan [13].

When it comes to load balancers specifically on Jenkins, there is related work in the form of load balancer plugins [14] [15]. The plugins are relatively basic in comparison to the available theory, but can function as a good starting point when developing a load balancer.

### 1.3 Limitations

There are numerous ways of improving the performance of a CI server. Covering all the possible areas of improvement would not be feasible in the given time span. Instead, we will focus on the node configurations and the load balancing, since this was brought up in interviews with Ericsson prior to the project start.

One limitation is the existing Jenkins setup used in the project. This setup comes with a lot of installed plugins that might affect the results in comparison to a clean install of the CI server. Because of time limitations, we will not try our implementations without the pre-installed plugins.

An important factor for the build speed in addition to the node configurations is the threading settings for the individual jobs. This can for instance be the flags related to parallelism for the build tool *make* [16]. Including these would increase the complexity and is not possible in the given time span.

There exists other CI servers beside Jenkins, for instance Travis CI, Strider, Go, and TeamCity. Since the project is conducted specifically on Jenkins, it may not be possible to apply all the results on other CI servers. Still, while other CI servers do not work in the exact same way as Jenkins, they have similar characteristics. For instance, the equivalent of a Jenkins executor in TeamCity and Go is called a Build Agent[17] and a Go-Agent[18] respectively.

# 2

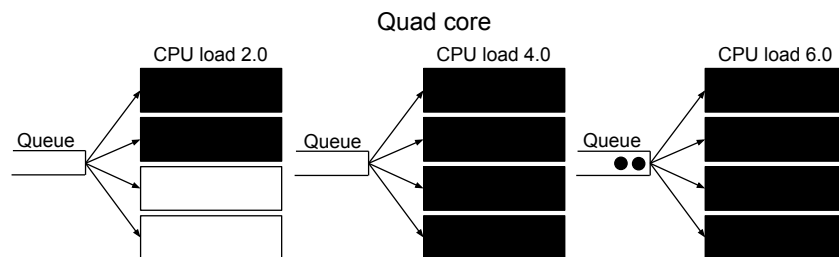
## Background

This chapter presents background information for Unix load average, regulators, load balancing and the default load balancer.

### 2.1 Unix Load Average

The load average is a metric that displays the fraction of processes using a slice of the CPU time. A value of 1.0 does not necessarily mean that 100% of the processor is utilised. For instance, processes may wait for I/O data but still use the CPU time. The load average is presented over the last one, five, and fifteen minutes.

A rule of thumb is to have the load average value below the number of cores in the system to avoid overloading the system [19]. See Figure 2.1 for a visual demonstration on how the load average value behaves on a quad core processor.



**Figure 2.1:** An illustration on how the load average value behaves on a quad core processor. Each dot in the queue represents a process waiting for CPU time. The rectangular blocks represent the current load of one core. When the load average is above the number of cores, jobs will be waiting in the queue for CPU time.

### 2.2 PID Regulator

A proportional–integral–derivative (PID) regulator tries to keep a metric at a chosen value (set point) by calculating the difference (error value) between the current value (process value) and the set point [20]. To keep the metric at the set point, the regulator regulates a control variable using three parts: a proportional, an integral and a derivative part. The control variable is what we modify to make the process value go up or down, it can for instance be starting or stopping a heating element to keep a temperature at a set point.

The proportional part accounts for the current error value by multiplying the current error value with a proportional constant. The integral part accounts for the past

error values, the integral contribution grows larger and larger the longer we stay away from the set point. The derivative part accounts for future changes by looking at the process value's current rate of change.

Regulators do not have to use all three parts, what parts to use depends on the system specifications. The most common types of regulators are P, PI, PD and PID regulators.

### 2.3 Load Balancing

A load balancer's job is to distribute a finite number of jobs to a finite number of nodes with the goal of minimising the elapsed time between the start and finish of a set of jobs, as well as avoiding overloading the nodes. The general formulation of the load balancing problem is NP-Complete [21]. A load balancer can be either static or dynamic.

A static load balancer makes decisions based on statistical information about a system. One example of this is a load balancer that distributes jobs depending on the average workload of a node. The important part is that a static load balancer does not take the system's current state into account when distributing jobs.

A dynamic load balancer makes decisions during runtime by reacting to the current system information, for instance the current load of the nodes. A dynamic load balancer is generally more effective, but requires more overhead since it needs to analyse the system every time a job is to be distributed. Dynamic load balancers are also more flexible in the sense that they can adapt to fluctuating workloads [22]. Fluctuating workloads are represented by heavy, medium, and light jobs in our test environment, see Section 3.2 for more details.

Load balancers are either preemptive or non-preemptive [22]. The load balancer in this thesis is non-preemptive. This means that once a job has started on a node, it will not be moved to another node, even if the current load of the node becomes overloaded, or the characteristics of the job changes.

Another characteristic of load balancing is if the system is centralised or decentralised [22]. A centralised system has one head node called the central scheduler that does all the load balancing. The nodes in the network send their load information to the central scheduler, which then distributes the load depending on this information. In a decentralised system, the nodes instead broadcast their load information to other nodes in the network, and can either take jobs from other nodes if the load is too low, or give away jobs to other nodes if the load is too high. A centralised system has less overhead, but also less reliability [22]. If the central scheduler node goes down, the whole load balancing stops working. This single point of failure does not exist in a decentralised system. In general, most of the previous research has been about decentralised load balancing systems. However, our system consists of a centralised system where all the load balancing is done by the head node.



Current load was mentioned as a possible system characteristic to measure, so this needs to be defined a bit further. A good way to define the system load is called a load index. The load index is a non-negative value, taking on zero if a node is idle, and becomes higher when the load increases [23].

Several metrics can be used as a load index, for instance [22]:

- **Most Idle Executors:** Based on the current number of idle executors. A node with only idle executors gets a load index of zero.
- **Least Load Average:** Load Average is the Unix load measurement metric described in Section 2.1. A lower load average means a lower load index. Variations exist which measure the Load Average during different time spans, for instance during 10s, 60s, or 5m.
- **Least CPU Usage:** Similar to Least Load Average, but measures the CPU usage instead.
- **Normalised Response Time:** A comparison between how a run time of a job is affected if the machine is loaded, compared to if it is idle. [23].

A load index can also be defined as a combination of all the above examples, or some other available metric. Different types of jobs can require different load indices. Some jobs may be disk dependant, in which case a load index taking disk usage into account might be more beneficial, and some jobs may be CPU dependant, in which case an index based on the CPU usage might be better. If the characteristics of a job is predictable, more effective indices can be created [22].

According to a comparative study, a load index based on the CPU queue length gives better performance in general than a load index based on CPU utilisation [23]. The reason for this is that the queue length can more precisely measure the current load on the computer. The CPU utilisation will at most reach 100% on a fully loaded computer, no matter how many processes are in the CPU queue. For instance, it will not be possible to distinguish a busy computer with five processes in queue, to one with ten processes in queue. CPU queue on the other hand, will show a difference between the two scenarios; the queue will be larger on the node with ten processes in queue compared to the node with five processes in queue. This distinction is important since a node with more process in the queue is obviously under heavier load than a node with less processes in the queue.

While it is possible to develop more advanced and complex load indices, previous research claims that relatively simple load indices can give a very good performance, and that more complicated indices are unlikely to improve that performance much further [24].

In a CI server, a load balancer can also take things like cache, and previously set up workspaces into account when assigning the jobs. If a workspace has not already been created for a job, it may take extra time to create and download necessary

data. An example of this can be seen in the default Jenkins load balancer described in Section 2.4.

## 2.4 Default Jenkins Load Balancer

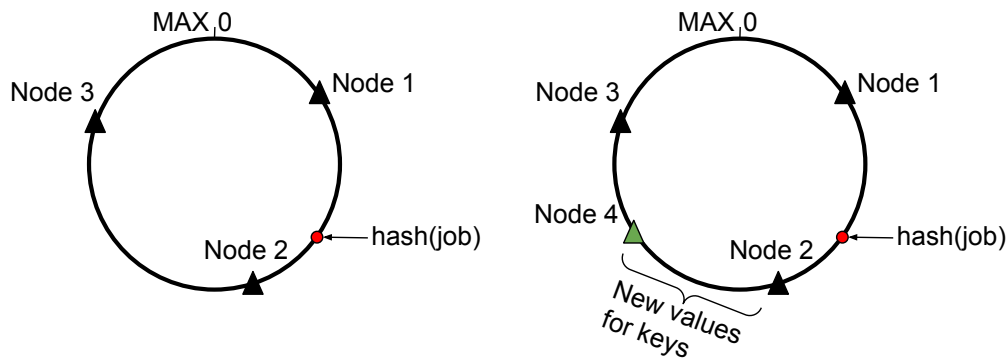
To understand the problem assignment about improving the load balancer, it is important to understand how the default Jenkins load balancer works. The default load balancer uses hashing to distribute the jobs.

In a naive load balancing hashing algorithm, a load balancer can for instance assign a job  $j$  by:  $\text{hash}(j) \% n$ , where  $n$  is the number of nodes. This spreads out the work well, but has a problem when the number of nodes is increased or decreased. A small re-size of the number of nodes will induce a remap of nearly every key. To avoid this, the default load balancer in Jenkins uses a method called Consistent Hashing. Consistent Hashing is an alternative hashing algorithm which performs better on hash tables that re-size often [25].

The consistent hashing algorithm works as follows: each node is hashed by its name and added to the consistent hash table together with a number of replicas corresponding to the available executors on the node. More precisely,  $100 \cdot \#Executors$  entries are added to the table for each node. This means that nodes with more executors have a higher chance of being assigned jobs than nodes with less executors. The hash table can be seen as a virtual circle of a sorted list of integers, corresponding to the possible hash values. The nodes are then distributed across the circle, see Figure 2.2. A good hash function will distribute the nodes in a random fashion uniformly around the circle.

To assign a job to a node, we hash the job and use that number as a key  $k$ . The key is then used as a probe point in the hash table. The resulting node is the closest node more than or equal to  $k$  (the next node going clockwise in the circle). If the node is full, the next node in the circle is used.

When a new node is added to the system, it is hashed like the previous nodes, and added to the circle. As can be seen in Figure 2.2, only the key-value pairs in between the new node and the node next to it counter-clockwise need to be remapped. All other keys keep the same value.



**Figure 2.2:** Simplified representation of a consistent hash table. When assigning a job, the hash of the job is calculated, and the job is assigned to the closest node going clockwise in the circle. In this case, the job will be assigned to Node 2. When a new node is added (Node 4), most keys stay mapped to the same value. The job in this case will still be mapped to Node 2.

The effect of balancing using hashes is that most jobs will keep getting assigned to the same node as they have previously been built on. This method of assigning jobs has both positive and negative aspects.

The positive aspects is that the job already has a workspace set up when it gets assigned to a node it has been built on previously. This is especially useful since a core point of CI is to build and update the code often, which means that the source code usually only has a few small changes at every update. These small changes can then be updated in the workspace. If a job is assigned to a new node which it has not been built on previously, the whole source code needs to be downloaded from scratch. This is a much slower process than updating. Another positive aspect is that the size and the number of workspaces tend to be smaller using this method of job distribution. The reason for this is that since the jobs tend to stick to the same nodes, they can use the workspace which is already set up. In short, the more nodes a job gets assigned to, the more nodes need to have that job's source code in their workspace.

The negative aspect of this sort of load balancing is that the load is not evenly spread out across the nodes. A situation can arise where one node gets the majority of the work assigned to it, while another node sits idle.

For instance, imagine a scenario where five jobs are to be distributed for the first time to two nodes A and B. The nodes have five executors each. If the first job gets assigned to A, A will have four idle executors after the assignment, and B will have five idle executors. Assuming that the hash function is good, and distributes hash values evenly across the available range, the second job will have a 4/9 chance on average of being assigned to A as well, since there are nine idle executors in total, four of them being on A. In the worst case scenario, Jenkins can fill up all the executors on A before distributing any jobs to B. The probability of this is small, but

possible. Since the same input always generates the same result in a hash function, the jobs do not change nodes unless the currently assigned node is full. Therefore, A will continuously be bogged down while B sits idle.

A build will run faster on a node with less load compared to a node that is busy. Therefore, the system in this situation will run slower than if the five jobs were distributed evenly between the two nodes. This is why alternative load balancers may be beneficial when it comes to the build speed.

# 3

## Methods

This chapter contains information about the test specifications, development, how the results were analysed, and load indices for the load balancing.

### 3.1 Distributed System Specifications

Our system consisted of one master node and eight slave nodes. The master node did not have any executors, i.e., the master node did not execute any jobs. Instead, it distributed the jobs to the slave nodes.

All of the slave nodes were virtual machines created from large server computers. The server computers had an Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz processor with twelve cores available. The virtual machines ran a SUSE Linux distribution with 32GB RAM. Two of the virtual machines were configured to use eight cores, and the other five were configured to use four cores. They all used one thread per core. The different configurations were used so machines with different specifications could be tested.

### 3.2 Job Configurations

Different types of job configurations were used when the system was tested.

The highest level of job is called a job matrix. A job matrix is a job which spawns a number of child jobs, where each child job is executed on one executor. Three job matrices were used. One which compiles and link together large components (called Heavy Matrix), one which compiles small components and runs various tests on the components to ensure correct program behaviour (called Medium Matrix), and one which does very fast operations to simulate light jobs (called Light Matrix).

The individual jobs are categorised as either heavy, medium or light. Heavy consists of jobs where the execution time is large (around ten minutes), light consists of jobs where the execution time is small (a few seconds), and medium consists of jobs that have an execution time somewhere between light and heavy.

The different types of jobs were used to get data that closely simulated real world data. In a real world scenario, the jobs may vary between light, medium and heavy, and the node configuration tests should take that into consideration.

In some of the job matrices, three test suites were run on the source code. The test suites consisted of the following:

- **Lint:** A Unix utility that does static code analysis. Lint tries to detect bugs, non-portable code, or wasteful code in C programs [26]. Lint also performs stricter type checking than the regular C compiler.
- **Run:** In the Run suite, various custom software tests written by the developers are run on the software. These tests can for instance be written in CppUnit, a unit testing framework for the C++ language [27].
- **Valgrind:** A programming tool used for debugging and profiling software. Valgrind is used to find memory-management problems (for instance memory leaks), cache problems, as well as other various memory and CPU related faults in software running on Linux [28]. Software profiling is in general used to aid program optimisation, but in this test suite the primary purpose of Valgrind is to find memory related problems.

### 3.3 Software

We developed a node monitor tool that ran on the nodes to monitor the system usage. The tool was made in the Unix shell and command language Bash. The tool collects CPU, RAM, disk, and bandwidth usage, as well as load average (described in Section 2.1) statistics over time. The data is formatted using the file format CSV (Comma-separated values). The CSV files are then used to create graphs so the computer usage statistics can be analysed and interpreted.

We also developed a tool to automate the testing process. The tool starts a job matrix using Jenkins CLI and calculates the elapsed time from start to finish. Then depending on what is being tested, either the number of executors is increased or the load balancer algorithm is changed, and the job matrix is run again. The data is collected in files that are graphed and analysed to find out which configuration results in the lowest execution time. More information about the tests is found in Section 3.4.

The following programs were used in the tools:

- **gnuplot:** Portable command line graphing tool. Used to plot the graphs displaying the collected data. Gnuplot took CSV files as input and created graphs as PNG files as output.
- **Jenkins CLI:** Built in command line interface for Jenkins. Used to start jobs, install plugins, and download console build logs remotely.
- **Secure Shell (SSH):** Unix SSH client used to connect and execute commands on a remote machine. SSH was used to connect to the node machines and launch the node monitor software and the dynamic executor regulator software.

### 3.4 General Test Specifications

When testing the performance of the static and dynamic executor allocation methods, various test specifications were set up.

To get enough adequate data to reach a conclusive result, the matrix jobs were run a certain number of times, i.e., a certain number of iterations. In most cases, seven iterations were run on each configuration. In each iteration we calculated the elapsed time between the start and finish of the matrix jobs, this is called the makespan. After a test, the monitor software calculated the median makespan from all the iterations.

The results were graphed so the data could be analysed. The data was graphed as a box plot displaying the statistical distribution of all the individual runs, as well as the median. When presenting the results in the tables, the Median Absolute Deviation (MAD) was used as a way of showing the variability of the data points. MAD is a robust measurement of the data deviation. It is calculated by taking the median of the differences between the data points and the median [29].

$$\text{MedianAbsoluteDeviation} = \text{median}(|x_1 - m|, |x_2 - m|, \dots, |x_n - m|)$$

where  $x_1, x_2, \dots, x_n$  are the data points, and  $m$  is the median.

The reason MAD was used instead of standard deviation is that MAD is robust, and more resilient to outliers. When MAD is used, occurrences of a small number of outliers is irrelevant. This was beneficial since the results had relatively small deviations, with a few big outliers caused by system noise. See Section 5.1 for a discussion about system noise and experimental uncertainty.

### 3.5 Static Executor Allocation

The static executor allocation problem was solved by running a heavy, medium, and light matrix on every number of executors ranging from two up to 28. The results were analysed to find a suitable ratio between the number of executors and cores.

The maximum number of executors to test was based on three factors. Firstly, the number of jobs in the job matrix being run had to be greater than the number of executors. If it was not, an increase in executors would not make much of a difference in terms of makespan, since all the jobs would run simultaneously. Secondly, increasing the number of executors, and therefore the number of jobs, made the tests take longer to complete. Finally, going above 28 executors did not give any new interesting information in the results. Therefore, a maximum number of 28 executors was decided as a good compromise between the required test time and the result data.

### 3.6 Dynamic Executor Regulator

As a proof of concept, we developed a program to analyse a node's current load and assign the number of executors dynamically depending on the load. The program was run remotely on the node and ran continuously, changing the number of executors dynamically during runtime.

More specifically, the program was designed as a proportional-derivative (PD) regulator (see Section 2.2 for the details). The regulator tried to minimise the error value by regulating the number of executors on the node.

Similarly to load balancers, the regulator had a load index that affected the decision making. We decided to use an index based on the load average. However, while the load balancer tried to keep the index minimised, the regulator instead tried to keep it at a set point. The set point was chosen to be the number of cores plus two. We chose this set point because of previous observations that the makespan was lowest when the load average was slightly greater than the number of cores.

Our system had some unique characteristics that other regulating systems do not usually have. For instance, the system is non-preemptive. Because of this, the jobs are not canceled when lowering the number of executors. After removing an executor, we have to wait until the job currently running on the executor is finished before the executor is removed. This is called dead time, a delay between when the change done by the regulator, and when the change is executed in the system [30]. The dead time is only present when removing executors, not when adding.

Another problem is that the number of executors is an integer with a short span, typically between one and a number equal to twice the number of cores. Because of this, the regulator can not be as accurate and precise as other regulated systems. This is not a big problem however, since we do not need to have an exact load average. Instead, we are satisfied if it is within 1.0 from the set point. The maximum number of allowed executors was chosen with the reasoning that if a higher number was allowed, the node may become overloaded more often since the load average has a one minute delay and the jobs are non-preemptive.

To analyse the results of the regulator software, a job matrix containing all three kinds of jobs (heavy, medium, and light) was created. The matrix was tested and the performance was compared to the best performing executor configuration in the static tests. The percentage of light, medium, and heavy weight jobs was decided to be 65% light, 32% medium, and 3% heavy jobs. This division of jobs represents a real scenario reasonably well. We had observed a similar division on Ericsson's CI server used in this thesis.



## 3.7 Load Balancer

Alternative load balancers were researched and compared to the default load balancer in terms of median execution times and load distribution.

The various load balancers were all launched with a Jenkins plugin. The plugin was developed as an extension to an already existing plugin called "Scoring Load Balancer" [14]. The plugin uses a concept called score, rather than load index. A score is based on the load index, but in contrast to load index, a higher score corresponds to a less loaded node. The resulting scoring list can be seen as a priority list, where a higher score represents a more suitable node to place the job on.

The following pseudo code describes how the load balancers assign a job:

```
1: function ASSIGNJOB
2:   nodeScores = new List of Tuple<Node, Score>
3:   validNodes ← Nodes which the job can run on
4:   for each Node n in validNodes do
5:     score ← calculateScore(n)
6:     nodeScores.add(n, score)
7:   Sort nodeScores by score
8:   while nodeScores is not empty do
9:     node ← nodeScores.Pop
10:    if node has free executors then
11:      Assign job to node
12:      Return true
13:  Return false (Assignment failed, all nodes were full)
```

What varies between the load balancers is the calculateScore function (i.e. what load index the score function is based on). The various scoring functions can be seen in Section 3.7.3.

### 3.7.1 Load Balancer Test Specifications

Two tests were developed when comparing the load balancers.

The first was called a start-up test, performed by running a single heavy job matrix. The reasoning for this test was to see how well the load balancer distributed the jobs when a matrix came in, and all the nodes were idle. The motivation for the test is that a good start up distribution is very important for a good load balancing, especially for heavy jobs. Since the system is non-preemptive, a bad start-up distribution can lead to nodes becoming overloaded and unusable for a long time.

The second test was called a continuous-flow test. The test was performed by running eight medium job matrices after each other with one minute between each start. The run time of one matrix is longer than one minute so that they overlap. This test analysed how well the load balancer distributed the jobs when the nodes are working and have idle executors ready. The motivation was to simulate a real

system where the jobs come in at different times, in a continuous flow.

For both tests, the number of jobs in each matrix was around 25% of the total number of executors counting all the nodes. The reason for this amount is that the more idle executors that are available, the more decisions are possible for the load balancer to make. If we have too many jobs, there will only be a few idle executors available and the different load balancers would not vary much in the decision making. This would also mean that the penalty for a bad load balancer would be smaller.

### 3.7.2 Computational Power

Three load indices in Section 3.7.3 were based on the nodes' computational power. The computational power was calculated similarly as in [11].

The Nasa Advanced Supercomputing (NAS) Parallel Benchmark [31] was used to calculate the computational power of each node. The NAS Parallel Benchmark consists of six different benchmarks evaluating the performance of the computer by testing various components. The benchmark results were measured in total million operations per second (Mop/s) [32]. The average Mop/s of the six tests were gathered for the nodes.

The ratio for a node  $i$  is defined as:

$$PowerRatio_i = \frac{P_i}{P_{max}}$$

where  $P_i$  is the computational power of node  $i$ , and  $P_{max}$  is the computational power of the most powerful node. The  $P_i/P_{max}$  values can be seen in Table 3.1, column four.

**Table 3.1:** The nodes with their computational power and their corresponding ratio. Although the system specifications of the nodes were identical with the exception of the number of cores, there was a difference between them. This was most likely because of the virtual machine configurations, see Section 5.1 for more details.

Node	#Cores	Mop/s	$P_i/P_{max}$
N-41	4	960.96	0.37
N-44	4	1048.24	0.40
N-43	4	1039.47	0.40
N-45	4	1081.10	0.41
N-42	4	1852.12	0.71
N-81	8	2175.07	0.83
N-82	8	2622.19	1.00

### 3.7.3 Load Indices Design

As mentioned in Section 2.3, a load index was used to define a load balancing strategy. We decided to base the load indices on five parts: CPU usage, load average, number of busy executors, number of cores, and the computational power benchmark. When choosing what parts to use, we had to take a few things into consideration. Since it is a heterogeneous system, it is important to take the various node specifications into account in the load index. CPU usage does this by itself, since a better performing node will increase its CPU usage more slowly than a worse performing node. However, on indices based on the load average or the number of busy executors, it is important to combine it with either the number of cores, or the computational power of the node.

By considering these factors, the following load balancers were decided as candidates:

**Cores and Load Average (CoresLoadAvg):** The score is defined as the number of cores divided by the current load average.

$$Score = \frac{\#Cores}{1.0 + LoadAverage}$$

**Idle and Busy Executors (IdleBusyExecs):** The score is based on the current number of idle and busy executors on the node.

$$Score = \#IdleExecutors - \#BusyExecutors$$

**Cores and Busy Executors (CoresBusyExecs):** The score is based on the number of cores divided by the number of busy executors. We add 0.5 to avoid division by zero. The value 0.5 was chosen because it makes the function distribute the jobs well between powerful and weak nodes. With the value 1.0, the powerful nodes took too many jobs, and with the value 0.1, the weak nodes took too many jobs.

$$Score = \frac{\#Cores}{0.5 + \#BusyExecutors}$$

**Cores, Busy Executors, and Load Average (CoresBusyExecsLoadAvg):** The score is based on the number of cores divided by current load average and the number of busy executors.

$$Score = \frac{\#Cores}{0.5 + LoadAverage + \#BusyExecutors}$$

**CPU Usage (CPU):** The score is based on the current CPU usage. A lower CPU usage corresponds to a higher score.

$$Score = 100 - CPUUsage$$

**Exponentially Smoothed CPU Usage (ExpCPU):** The score uses the CPU usage similar to the previous index, but uses the exponentially smoothed average to reduce fluctuations. It was calculated in the following way:

$$s_t = \alpha \cdot x + (1 - \alpha) \cdot s_{t-1}$$

where  $s_t$  is that resulting average CPU usage,  $x$  is the current CPU usage,  $\alpha$  is the exponential factor ( $0 < \alpha < 1$ ), and  $s_{t-1}$  is the previous average CPU usage. After testing different  $\alpha$  values, we decided to use 0.3. The full score is defined as:

$$Score = 100 - s_t$$

**Computational Power, Cores, Load Average, and Busy Executors**

**(CoresCompLoadBusy):** The computational power of each node was calculated and used in conjunction with the load average, number of busy executors, and the number of cores to determine the score.

The main difference of the load index this score is based on, compared to the load index defined in [11] is that we divide by the load average and the number of busy executors, rather than the number of tasks running on the node. This is because in the paper it is assumed that every task uses one thread. In this thesis, the jobs use more than one thread. Therefore, we used the load average value to more precisely specify how much of the computer is busy. Because the load average is the average load over the last minute, it will take a few seconds before the load average is raised. This will lead to all the jobs getting assigned to the most powerful node during start up. To avoid this, the number of busy executors was added to the load index equation as well. A value of one was added instead of 0.5 because it made the index distribute the jobs better together with the computational power.

$$Score = \frac{P_i}{P_{\max}} \cdot \frac{\#Cores}{1.0 + LoadAverage + \#BusyExecutors}$$

**Computational Power and Busy Executors (CompBusyExecs):** The score is based on the computational power and the number of busy executors on the node.

$$Score = \frac{P_i}{P_{\max}} \cdot \frac{1}{1.0 + \#BusyExecutors}$$

**No Load Balancer:** Used to simulate the result of using no load balancer at all. Assigns jobs to the first non-busy node, if the node has no executors available, it assigns to the next node and so on. The nodes were ordered so that the 8-core nodes would be used before the four core nodes.

# 4

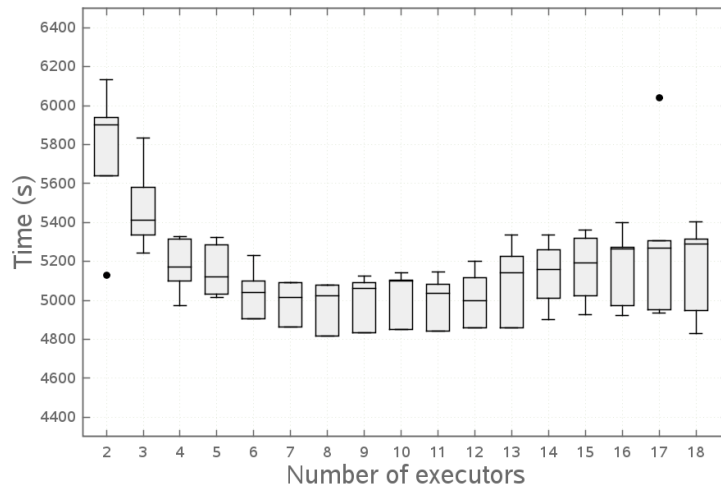
## Results

This section presents the results from the static executor analysis, the dynamic executor regulator, and the load balancers.

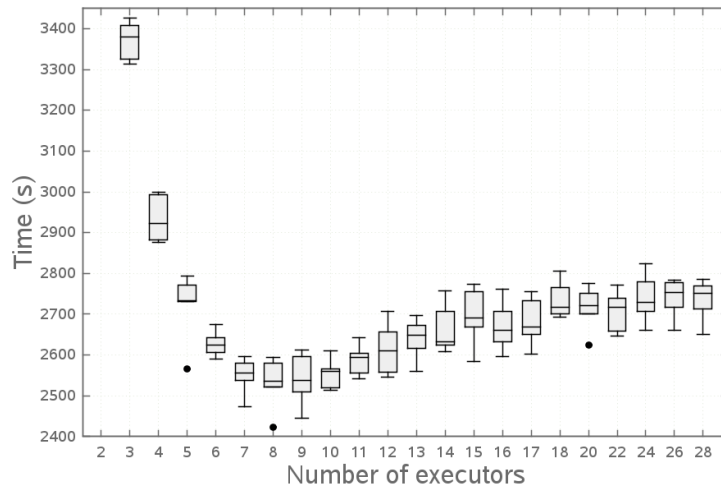
### 4.1 Static Executor Allocation

Figure 4.1 shows the execution times with different number of executors for a heavy, medium, and light matrix on an 8-core node. The lowest makespan for the heavy and medium matrices occurred around 8-11 executors, which results in a ratio of 1.0 to 1.4 executors per core. When the number of executors is increased further, the makespan increases. For the light matrix, the makespan kept decreasing as the number of executors increased until it reached 15 executors. After that, the makespan leveled out. The executor per core ratio was similar for a 4-core node.

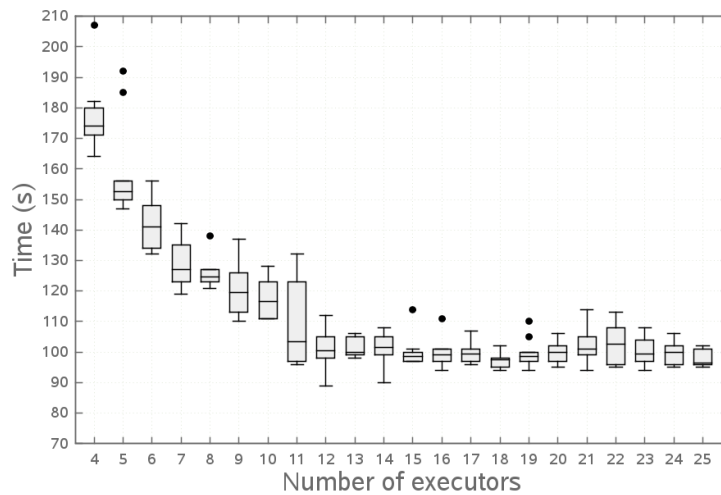
## 4. Results



(a) Heavy matrix



(b) Medium matrix



(c) Light matrix

**Figure 4.1:** The resulting makespans from the static executor allocator tests. The line in the boxes is the median, and the upper and lower boxes are the first and third quartile respectively. Note that the result of two executors on the medium matrix was very high and not included in the graph as to improve visibility.

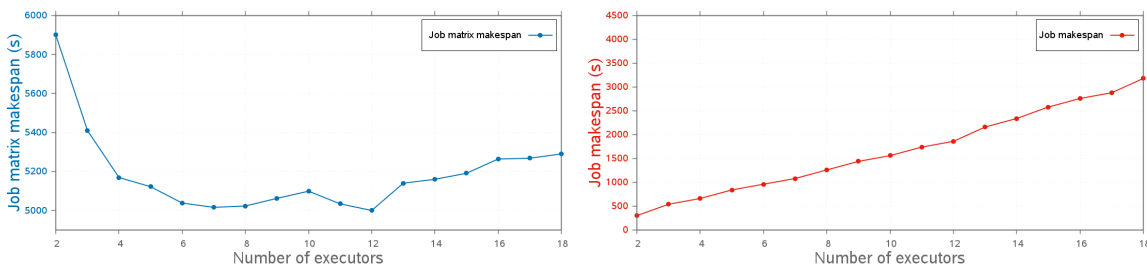
### 4.1.1 Individual Job Performance

A comparison between the makespan of an entire heavy job matrix, and an individual child job's makespan is shown in Figure 4.2. The individual job's makespan is increasing linearly as the number of executors increases. This is because when more executors are added, more jobs are done simultaneously. Since the jobs have to share the resources of the node, the individual makespans of the jobs will increase, although the makespan of the entire matrix may decrease.

The individual jobs' makespans are important if the system is configured to fail fast. Fail fast is when a job matrix is aborted directly after an individual job has failed. This speeds up the system and saves computer resources since less time is spent on building already failing job matrices.

It is also possible to configure the system so it completes the job matrix even if individual jobs fail. The advantage of this is that the developers get more thorough feedback on all the failures. Whether to use fail fast or not depends on personal preference and the amount of resources available in the system.

The best number of executors depends on how the user prioritises between the individual jobs' makespans, and the total makespan of all the jobs in the matrix. Some applications may prefer the total makespan to be minimized, while others find the individual jobs' makespans to be more important. A good compromise between the two seems to be around eight executors on an 8-core node.



(a) Job matrix makespan

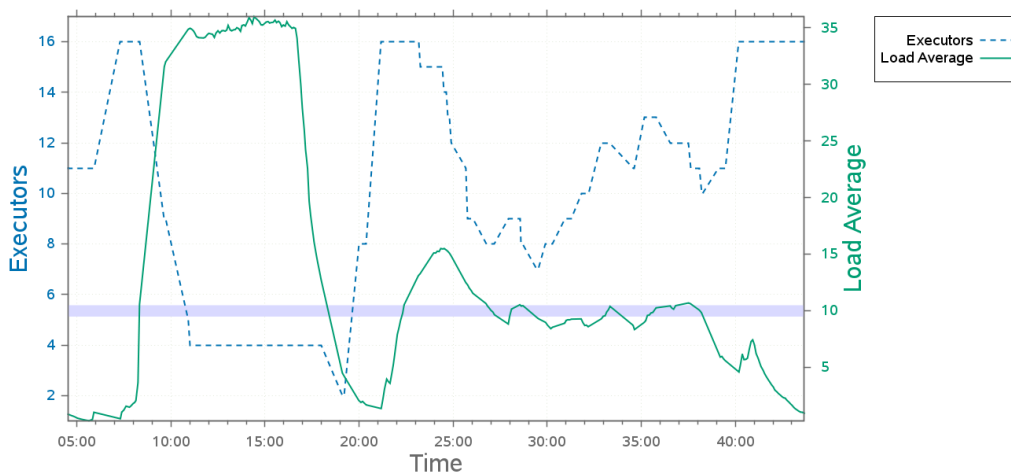
(b) Individual job makespan

**Figure 4.2:** An illustration of the makespan of an entire heavy job matrix in (a), and an individual child job in (b). Even though the makespan for the entire job matrix may decrease as the number of executors increase, the makespan of an individual job will grow linearly.

## 4.2 Dynamic Executor Regulator

With the dynamic executor regulator, the makespan increased with approximately 5% compared to the best static node configuration for an 8-core node. The regulator did bring benefits to the system, like a reduced risk of overloading the nodes, and a sort of load balancing, analysed further in Section 5.4.1. The dynamic executor regulator also handled dynamically changing workloads better than the static configurations.

Figure 4.3 shows how the number of executors is regulated for a matrix run with a target load average of ten. When the load increases above the target range, the number of executors decreases and vice versa. The load average is within target range most of the time. The load average is high at 8:00-19:00 because the jobs that are running on the executors at that time are heavy and utilize many threads. The regulator tries to decrease the high load average around 7:00 by setting a lower number of executors. The regulator actually decreased the number of executors earlier than this, but since it takes a while before jobs complete on the executors that are about to be removed, the actual executor count is decreased a few minutes after the change. Around 39:00, the load average drops because the job matrices are starting to complete. In a static node allocation, the load average would have been higher on average during the run.



**Figure 4.3:** A graph over how the dynamic executor PD-regulator worked for a job matrix on an 8-core node with the load average set-point at ten. When the load increases, the number of executors decreases and vice versa. The solid line represents the load average, and the dashed line represents the current number of executors.



### 4.3 Load Balancer

The results from the load balancing experiments are presented in Table 4.1. Eight new load indices are analysed and compared to the default Jenkins load balancer. The time differences on the continuous-flow test are smaller than the ones on the start-up test; this is because the continuous-flow test primarily contains medium weight jobs while the start-up test contains primarily heavy jobs. The heavier jobs use more resources which results in a larger penalty for bad load balancing decisions.

Since we are comparing to the default Jenkins load balancer described in Section 2.4, we did not get an improvement on every index. However, compared to using no load balancer at all, we saw an improvement on the majority of indices.

**Table 4.1:** Load balancing results for the Start-up and Continuous-flow tests. We present the median makespan times, the improvement in comparison to the default load balancer, and the MAD. For more details about the load indices, see Section 3.7.3.

Load Index	Start-up			Continuous-flow		
	Med.(s)	Imp.(%)	MAD	Med.(s)	Imp.(%)	MAD
Default Load Balancer	733	—	11	439	—	9
CompBusyExecs	555	24.3	9	425	3.2	3
CoresBusyExecs	553	24.6	9	433	1.4	6
CoresBusyExecsLoad	554	24.4	13	438	0.2	6
CPU	607	17.2	8	456	-3.9	16
IdleBusyExecs	706	3.7	11	430	2.1	5
ExpCPU	707	3.6	104	467	-6.4	13
CoresCompLoadBusy	1172	-59.9	250	440	-0.2	12
CoresLoadAvg	2027	-176.5	9	462	-5.2	9
No Load Balancer	2033	-177.4	7	461	-5.0	6

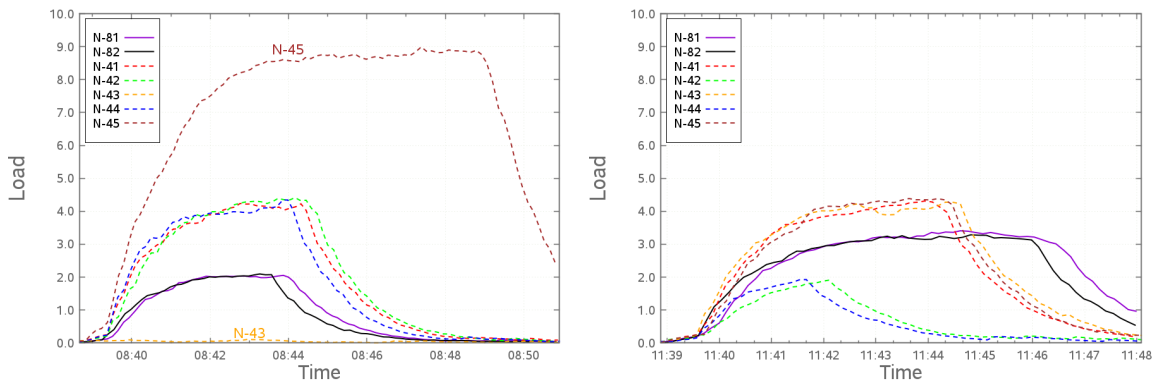
Another way to see how well the load balancers perform is to look at the node loads during a run. Figure 4.4 shows the node loads during a start-up test and a continuous-flow test for the default and CoresBusyExecs load balancers. The load is calculated by taking the Unix Load Average (described in Section 2.1) and normalizing it by the number of cores on the node. A value of 1.0 corresponds to a fully loaded node. The closer the load lines are to each other, the better the load distribution is between the nodes. We compare with CoresBusyExecs because this was one of the best performing indices in both tests.

For the start-up test in Figure 4.4a, we can see that the default balancer distributes the load badly between the nodes. Node N-43 is not used at all, and node N-45 is getting severely overloaded. In 4.4b we see that the loads are closer to each other, which tells us that CoresBusyExecs balancer distributes the load better.

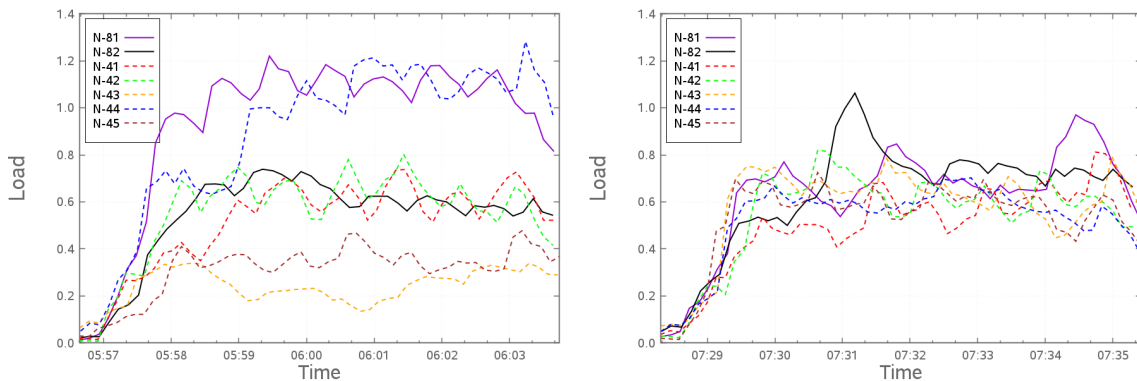
For the continuous-flow test, we can see in 4.4c that the default balancer spreads

## 4. Results

out the load unevenly. Some nodes are barely loaded, while others are heavily loaded. CoresBusyExecs on the other hand, distributes the load evenly between all the nodes.



(a) Start-up test with the default load balancer (b) Start-up test with the CoresBusyExecs load balancer



(c) Continuous-flow test with the default load balancer (d) Continuous-flow test with the CoresBusyExecs load balancer

**Figure 4.4:** Load comparison between the default and the CoresBusyExecs load balancers during a start-up and a continuous-flow test. The dashed lines are 4-core nodes and the solid lines are 8-core nodes. In both tests, the loads are closer to each other for the CoresBusyExecs balancer, which tells us that the jobs are more evenly distributed between the nodes, compared to the default balancer.

# 5

## Discussion

This section interprets the results and addresses interesting findings.

### 5.1 Experimental Uncertainty

The test environment included some noise factors that may have affected the credibility of the test results.

Other users on the system could have had some effect on the results. Our nodes were not used by any other users, but factors like the network was shared and could have had an effect. We tried to minimise this by running the tests at night when the total system load was low. However, some of the tests took a very long time to complete, which meant that they had to be run during the day as well. We do not think this affected the results too much, but it is good to have this in mind when interpreting the results.

The nodes were virtual machines created from a large server computers. Because of this, the virtual machines' performance sometimes differed from day to day, even with identical specifications. We tried to counteract this variability by running the tests that were going to be compared in fast succession after each other. Part of these performance differences can be observed with the computational performance benchmarks presented in Table 3.1.

### 5.2 Node Overload Problems

There are some problems that can occur if a node becomes overloaded. A common Jenkins configuration is to abort a build if it is taking too long. This is in general a good idea so that jobs do not get stuck, but an overloaded node could trigger this timeout and fail the build because of the increase in build time associated with an overloaded node.

Linux kernel errors have also been observed at Ericsson when too many heavy jobs are assigned to the same node, overloading the system.

Since reliability is key for a proper functioning CI server, these overload issues can have a very negative effect on the system's quality of service.

### 5.3 Static Executor Allocation

The heavy and medium matrix tests in Figure 4.1 show that the makespans decrease as the number of executors increases until the executor to core ratio is around 1.0. When the ratio goes above 1.4, the makespan starts to increase.

The reason for the decrease is that as the number of executors increases, more jobs run simultaneously which utilises more of the node's resources. However, after a certain point, the overhead for the jobs starts to take up a larger portion of the processor time, and the makespan starts to increase. This behavior is not present on the light matrix tests since the light jobs only use a tiny amount of resources. Instead, the makespan eventually stagnates as the number of executors increases.

An interesting result from the heavy matrix test is that the improvement when going above two executors is not as large as we expected; certainly not as large as the medium matrix test results. The main reason for this is that the heavy jobs are very parallelisable and utilise many threads. Because of this, two heavy jobs use much more of a node's resources than two medium or light jobs, which makes the difference when raising the number of executors smaller.

### 5.4 Dynamic Executor Regulator

We started the development of the dynamic executor regulator by creating a proportional–integral–derivative regulator (PID regulator).

The main benefits of the integral part of a regulator is that it accelerates the process value towards the set point and removes any steady-state error that might be present. The downside is that it results in a more unstable system [20]. Because of the special characteristics of the system described in Section 3.6, we did not have any steady-state error, or any problem reaching the set point fast enough. Therefore, the integral part of the regulator proved to do more harm than good, and it was decided to remove the integral part and use a PD-regulator instead.

The derivative part of the regulator was mainly used to counteract the big changes in load average when a heavy job started or completed. The rapid changes in the load average increased the derivative contribution and made the response from the regulator less aggressive. This made the regulation more stable and was beneficial to the system.

When run on a single node, the regulator did not improve the makespan. On average, the regulator was around 5% slower than the best static executor configuration. While slower, the regulator provided some other benefits to the system that the static node configuration did not have.

For instance, the regulator might have been faster than the static node configurations when run on multiple nodes under certain conditions. This is because the load would be more balanced between the nodes compared to the static node con-

figurations. This is explained in more detail in Section 5.4.1. We did not test this because we were implementing a load balancer which would test similar load balancing improvements.

The regulator also made the configuration easier for heterogeneous node pools by removing the need of assigning the number of executors manually. Since the executors are regulated according to the load, a node will setup a good number of executors by itself. If a node is configured with the number of executors too low for instance, the regulator will increase the number of executors.

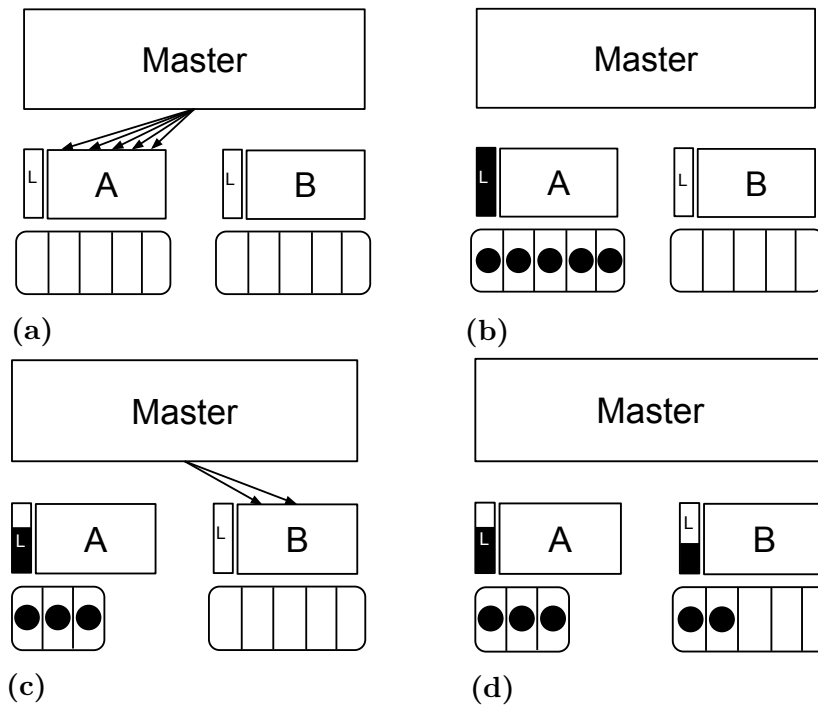
Reducing the number of executors when necessary also decreased the risk of a node becoming overloaded. This reduced the potential overload problems described in Section 5.2.

### 5.4.1 Decentralised Load Balancing Effect

A major benefit of the dynamic executor regulator was that it reduced the risk of a node becoming overloaded because of having too many heavy jobs run on it. If the node monitor saw that the node was becoming overloaded, it reduced the number of executors, and therefore the number of jobs running on the node. The not so obvious implication of this is that there is a possible scenario where the risk of overloading is reduced, while the jobs in the queue still have another node to run on, i.e., the run queue does not become larger, even though the number of executors is reduced on one node.

To see this, imagine a scenario with two nodes A and B, configured with five executors each. We have a continuous stream of heavy jobs that we want to run, but at most five jobs can run simultaneously. Because of the default Jenkins load balancer behavior, there is a risk that all jobs are assigned to node A. With a static node configuration, it's possible that node A would run all five jobs until completion and keep accepting new jobs, while node B would sit idle.

Using the dynamic node configuration proposed in Section 4.2, the regulator would notice that node A was getting overloaded, and lower the number of executors on A, to for instance three executors. Because A now only has three executors and is full, the next two jobs in the queue would have to be assigned to B. The result of this is that the job load is balanced out between the nodes, even with the default Jenkins load balancer. So the dynamic executor regulator can be used as a kind of decentralised load balancing. This situation is illustrated in Figure 5.1.



**Figure 5.1:** An illustration of the decentralised load balancing scenario. In (a) and (b), five jobs are assigned to A by the master. In (c), the regulator notices that A is overloaded and reduces A's executors to three. Since A is now full after three jobs, the next two jobs has to be assigned to node B, as can be seen in (d). The system load of the nodes is illustrated by the rectangle with an L in it.

## 5.5 Load Balancer

This section discusses the job distribution, instantaneous and smoothed load indices, overhead and time complexity, and lastly the computational power score index for the load balancer.

### 5.5.1 Job Distribution for Larger Systems

Our test system only contained 4-core and 8-core nodes. Since a real system may have a lot more diverse nodes, we calculated and tested how some of the load indices would react to this scenario. Only some of the indices were possible to predict this way without actually having the physical machines. The default load balancer could be tested since it bases its decisions on the number of executors, rather than the actual specifications of the node. The results can be seen in Table 5.1.

**Table 5.1:** Distribution of 14 jobs at start-up for nodes with cores ranging from four to 40

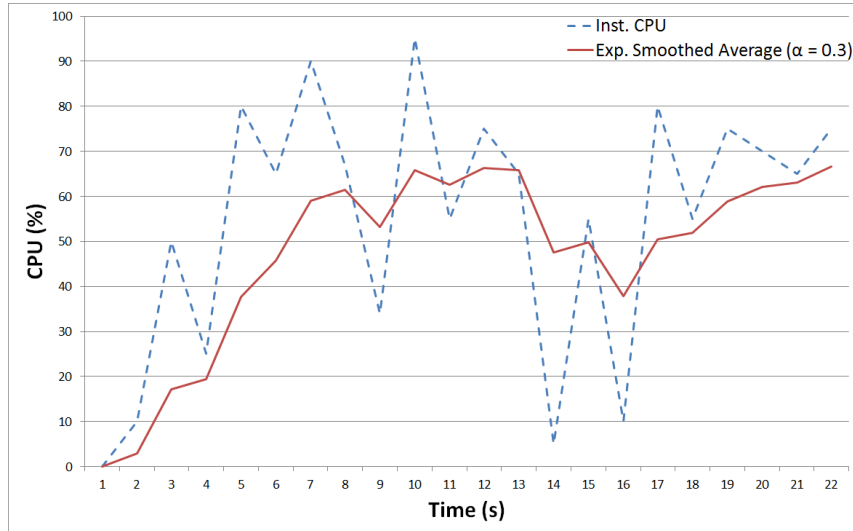
Job Distribution		Cores						
		4	8	12	16	24	32	40
Load Index	CoresBusyExecs	0	1	1	2	3	3	4
	CompBusyExecs	0	1	1	2	3	3	4
	Default	1	2	0	2	3	2	4
	CoresLoadAvg	0	0	0	0	0	0	14
	IdleBusyExecs	0	0	0	0	0	3	11

- **CoresBusyExecs:** Distributes the jobs well because it takes the number of cores and the current number of busy executors into account. As shown in Table 5.1, all nodes except the 4-core node get at least one job and the more powerful nodes get more jobs. This is the main reason why CoresBusyExecs performed well in the start-up test.
- **CompBusyExecs:** Performs well for the same reason as CoresBusyExecs. However, instead of using the number of cores to determine the node’s performance, it uses the computational power.
- **Default:** Distributes the jobs essentially randomly by consistent hashing. In this case it distributed the jobs fairly badly. The 12-core node got no jobs, while the 8-core and 4-core node got three jobs together. This is the reason why default in general performs worse than CoresBusyExecs in the start-up test. The default load balancer has the possibility to distribute the jobs well, but it also has the possibility to distribute them poorly.
- **CoresLoadAvg:** Distributes the jobs badly because the load average has a long reaction delay and does not react fast enough to newly assigned jobs. Even on a system with very similar nodes it may assign all the jobs to one node. This is the reason why load average consistently was the worst performing index in the start-up test.
- **IdleBusyExecs:** Distributes the jobs badly because it only takes the number of idle and busy executors into consideration. If the nodes have vastly different specifications, as in Table 5.1, it assigns most of the jobs to the 40-core node. In a system where the nodes have similar specifications it distributes the jobs better. This is why it performs well in our tests, as shown in Table 4.1.

### 5.5.2 Instantaneous and Smoothed Load Indices

The load indices presented in Table 4.1 can be divided into two categories. Load indices that are averaged over time (CoresLoadAvg and ExpCPU) and load indices that are close to instantaneous (CPU and CoresBusyExecs). Indices that are averaged over a longer period of time have a smoother behavior, and less high frequency fluctuation noise. However, they also have a longer reaction delay, and take a while

to react when jobs are assigned. Instantaneous indices react very quickly when jobs are assigned, but contain some high frequency fluctuation noise that can disrupt the load balancing. An example of instantaneous CPU usage and exponentially smoothed CPU usage can be seen in Figure 5.2.



**Figure 5.2:** An illustration of instantaneous CPU usage represented by the dashed line, and exponentially smoothed CPU usage with  $\alpha = 0.3$  represented by the solid line. The instantaneous CPU usage fluctuates a lot more and can result in the load balancer making bad decisions.

The CoresLoadAvg index is a good example of an index that has a too long average time period. Since a job matrix spawns multiple individual child jobs instantaneously, a long averaging period can lead to all of the child jobs getting assigned to a single node because of the large delay. This is the main reason that the CoresLoadAvg index was one of the worst performing indices in our experiments.

An example of an index on the other side of the spectrum is the CPU. The CPU usage is gathered over one second, and therefore reacts quickly when jobs are assigned. However, since the score only takes the last second into account, it fluctuates a lot which can lead to the load balancer making bad decisions. The results show that the index still performed well, which tells us that a fast reaction time is one of the most important components. To reduce the effects of the fluctuations, an exponentially smoothed CPU usage index was tested as well. This index performed well in early experiments, but performed worse in the final tests presented here. As a conclusion, this tells us that a fast reaction time is vital to a good load balancing, but it might be worth smoothing the index to avoid high fluctuations.

The load indices based on the number of busy executors (CoresBusyExecs and CompBusyExecs) react instantly on a job assignment, yet has none of the high frequency noise fluctuation problems. When a job is assigned, BusyExecs instantly raises the score, and will not lower it until the job finishes since the system is non-preemptive. This removes any large fluctuations. The bad part about BusyExecs is



that it does not take the weight of the job into consideration at all, a light job will affect the score the same way as a heavy job. The results show that BusyExecs is part of the best indices according to our experiments, especially on the experiment where all the jobs had the same weight (Table 4.1 Start-up test). But to get the optimal performance, one should combine the BusyExecs information with another index that takes the job weight into consideration, for instance the load average or the CPU usage.

### 5.5.3 Overhead and Time Complexity

All the load balancers proposed in Section 3.7.3 have some overhead in comparison to the default Jenkins load balancer. The default load balancer basically only does a hash table look up on every job assignment, which results in an average time complexity of  $O(1)$ . Our proposed load balancers on the other hand works by querying each node every time a job is assigned. The query asks for information to be able to calculate the current load of all the nodes, and sorts them in a prioritised order. The query part of the algorithm will do constant work on each node, resulting in a time complexity of  $O(n)$  for the queries. However, since the list is then sorted, the total time complexity required to assign one job becomes  $O(n \log n)$ , where  $n$  is the number of nodes that the job can be executed on (not the total number of nodes in the system).

Since  $n$  in general is small, we do not think that this affects the performance in any significant way, but it is good to have in mind if more nodes are added, or if an unusually high load is observed on the master node.

### 5.5.4 Computational Power Score Indices

The most positive aspect of the load indices using the computational power benchmark is that they take the actual nodes' performances into account when load balancing the jobs. While indices like CoresBusyExecs only take the number of cores into consideration, the computational power takes characteristics like processor speed and RAM into account as well. Sadly, all the nodes in our system have the same processor speed, which means that the strengths of the computational power score indices could not be demonstrated properly.

The negative aspect is that the computational power index requires a benchmark to be run on every node. This requires some more work initially, and every time a new node is added. The good thing is that the benchmark only needs to be run once per node, unless the system specifications of the node changes, although it might still be a good idea to run the benchmark periodically to make sure that the results are up to date. Future work can include automating the periodic benchmark runs. There are different sizes of the benchmark tests available, so for periodical runs, a lighter benchmark could be used to reduce the time required.

Alternatively, the benchmark can be run on one node, and the results reused on all the nodes with similar specifications. This will reduce the amount of work required,

but might not be fully accurate for all nodes. We saw an example of this type of inaccuracy in Section 3.7.2, Table 3.1, where the nodes differed a bit even with identical parameters.

Another point worth mentioning about the benchmark is that the results can be used to diagnose performance problems that might exist in the system when it comes to the nodes. For instance, it might be possible to see that a cluster of nodes performs worse than another cluster of nodes. If this happens it might be worth investigating what the cause of it is. Because of this, the benchmark might have other positive effects in addition to the use in load balancing, which might make it worth doing the extra work.

## 5.6 Ethical Aspects

While the increase in production speed brings forth a lot of positives, there are some ethical aspects that should be taken into consideration. According to an empirical study by Vafaie & Arvidsson [33], the nature of CI may increase the stress level for some employees, negatively affecting their health. One developer felt that faults introduced in the code had to be fixed immediately, causing him to feel more stressed. However, the majority of the interviewees felt that CI was a positive change to the work environment and even mentioned that it decreased their stress level.

The improvements presented in this thesis may increase the development and production speed. This might lead to employees feeling pressured to work longer hours to keep up with production, since the developers more and more become the limiting factor.

## 5.7 Future Development

It would be possible to more thoroughly research the best number of static executors by using a larger CI system with more diverse nodes. It would also be advantageous to have a system with a minimal number of installed plugins to avoid potential overhead and problems induced by the plugins. Another possible research aspect is to experiment with different options for the jobs themselves. Depending on the job options, the best number of executors might differ. The job options can for instance be the flags `-j` and `-l` for the build automation tool *make*. These flags control the amount of parallelism used in the building process [16].

The dynamic executor regulator software is currently a Bash program launched remotely on each node. This was created as a proof of concept to see if the dynamic node configuration could be beneficial. Future development could include creating a Jenkins plugin to handle it instead. A Jenkins plugin would greatly reduce the amount of manual work required to initiate the software on the nodes. A plugin could also add the ability to have several configuration options for each node, easily accessed through the Jenkins webpage. These options could for instance include the maximum and minimum number of executors, how aggressive the number of

executors is changed, and the target range for the load average.

Future work for the Jenkins load balancer could be to experiment with more of the available load balancer types. Only a few types of load balancers were tested in this thesis. There exists many other types of load balancers that may perform better than the types that were evaluated here.

Both the load balancer and the dynamic executor regulator could be improved by incorporating self learning mechanisms so that they could learn about how heavy certain jobs are, and adapt to this the next time the same jobs are run. This can make them able to adapt better to the fluctuating workloads present in the system.

# 6

## Conclusion

This thesis presents three ways to improve the performance of a CI server. Finding a suitable static executor node configuration, dynamically changing the number of executors depending on the load, and designing a better load balancer based on appropriate load indices for a heterogeneous and non-preemptive system.

The lowest makespan occurs when the number of executors in relation to the number of cores is between 1.0 and 1.4. The most important discovery is that the performance drops significantly when the number of executors is less than the number of cores. Dynamically assigning the number of executors proved to be an efficient way to reduce the risk of overloading the nodes, as well as a good way of automatically keeping the nodes well configured even during a dynamically changing workload. Using any of the top performing load balancers clearly led to a more balanced load distribution in comparison to the default load balancer and resulted in a decreased makespan of up to one fourth.

As the results show, a good configuration together with a better designed load balancer can have a great impact on the CI server performance.

# Bibliography

- [1] Booch G. OBJECT-ORIENTED ANALYSIS AND DESIGN With applications. Addison-Wesley; 1991.
- [2] Beck K. Extreme programming explained: embrace change. Addison-Wesley; 1999.
- [3] Kawaguchi K. Meet Jenkins; 2015. Accessed: 2016-06-08. <https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins>.
- [4] Jenkins. Jenkins Press Information; 2015. Accessed: 2016-06-08. <https://jenkins.io/press/>.
- [5] Kawaguchi K. Distributed builds; 2016. Accessed: 2016-06-08. <https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds>.
- [6] Fowler M, Foemmel M. Continuous integration; 2015. Accessed: 2016-06-08. <http://www.martinfowler.com/articles/continuousIntegration.html>.
- [7] Pinedo ML. Scheduling: theory, algorithms, and systems. Springer Science & Business Media; 2012.
- [8] Company Facts - Ericsson; 2016. Accessed: 2016-06-08. [http://www.ericsson.com/thecompany/company\\_facts](http://www.ericsson.com/thecompany/company_facts).
- [9] Paasivaara M, Lassenius C, Heikkila VT, Dikert K, Engblom C. Integrating global sites into the lean and agile transformation at ericsson. In: Global Software Engineering (ICGSE), 2013 IEEE 8th International Conference on. IEEE; 2013. p. 134–143.
- [10] Seppälä A. Improving software quality with Continuous Integration. Aalto University; 2010.
- [11] Bosque JL, Toharia P, Robles OD, Pastor L. A load index and load balancing algorithm for heterogeneous clusters. The Journal of Supercomputing. 2013;65(3):1104–1113.
- [12] Suguna S, Barani R. Simulation of Dynamic Load Balancing Algorithms. Bonfring International Journal of Software Engineering and Soft Computing. 2015;5(1):1.
- [13] Daraghmi EY, Yuan SM. A small world based overlay network for improving dynamic load-balancing. Journal of Systems and Software. 2015;107:187–203.

- [14] Yasuyuki I. Scoring Load Balancer plugin; 2013. Accessed: 2016-06-08. <https://wiki.jenkins-ci.org/display/JENKINS/Scoring+Load+Balancer+plugin>.
- [15] Nolan B. Least Load Plugin; 2015. Accessed: 2016-06-08. <https://wiki.jenkins-ci.org/display/JENKINS/Least+Load+Plugin>.
- [16] GNU make: Parallel; 2014. Accessed: 2016-06-08. [https://www.gnu.org/software/make/manual/html\\_node/Parallel.html](https://www.gnu.org/software/make/manual/html_node/Parallel.html).
- [17] Megorskaya I, Alexandrova J. Build Agent - TeamCity 9.x Documentation - Confluence; 2016. Accessed: 2016-06-08. <https://confluence.jetbrains.com/display/TCD9/Build+Agent>.
- [18] Jariwala J, Aravind S. Concepts in Go | Go User Documentation; 2016. Accessed: 2016-06-08. [https://docs.go.cd/current/introduction/concepts\\_in\\_go.html#agent](https://docs.go.cd/current/introduction/concepts_in_go.html#agent).
- [19] Andre. UNDERSTANDING LINUX CPU LOAD - WHEN SHOULD YOU BE WORRIED?; 2009. Accessed: 2016-06-08. <http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages>.
- [20] Lennartson B. Reglerteknikens grunder. Studentlitteratur AB; 2002.
- [21] Correa JM, Melo AC. Using a classifier system to improve dynamic load balancing. In: Parallel Processing Workshops, 2001. International Conference on. IEEE; 2001. p. 411–416.
- [22] Kameda H, Li J, Kim C, Zhang Y. Optimal load balancing in distributed computer systems. Springer Science & Business Media; 2012.
- [23] Ferrari D, Zhou S. An empirical investigation of load indices for load balancing applications. DTIC Document; 1987.
- [24] Zhou S. A trace-driven simulation study of dynamic load balancing. Software Engineering, IEEE Transactions on. 1988;14(9):1327–1341.
- [25] Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM; 1997. p. 654–663.
- [26] Pohl J. FreeBSD 11.0 - man page for lint; 2001. Accessed: 2016-06-08. <http://www.unix.com/man-page/FreeBSD/1/lint>.
- [27] cppunit;. Accessed: 2016-06-08. <https://www.freedesktop.org/wiki/Software/cppunit/>.
- [28] Seward J, Nethercote N, Weidendorfer J. Valgrind 3.3-Advanced Debugging

- and Profiling for GNU/Linux applications. Network Theory Ltd.; 2008.
- [29] Upton G, Cook I. median absolute deviation. Oxford University Press;. Available from: [//www.oxfordreference.com/10.1093/acref/9780199679188.001.0001/acref-9780199679188-e-1029](http://www.oxfordreference.com/10.1093/acref/9780199679188.001.0001/acref-9780199679188-e-1029).
- [30] Atkins T, Escudier M. dead time. Oxford University Press;. Available from: [//www.oxfordreference.com/10.1093/acref/9780199587438.001.0001/acref-9780199587438-e-1331](http://www.oxfordreference.com/10.1093/acref/9780199587438.001.0001/acref-9780199587438-e-1331).
- [31] NAS Parallel Benchmarks; 2016. Accessed: 2016-06-08. <http://www.nas.nasa.gov/publications/npb.html>.
- [32] Min G, Di Martino B, Yang LT, Guo M, Ruenger G. Frontiers of High Performance Computing and Networking–ISPA 2006 Workshops: ISPA 2006 International Workshops FHPCN, XHPC, S-GRACE, GridGIS, HPC-GTP, PDCE, ParDMCom, WOMP, ISDF, and UPWN, Sorrento, Italy, December 4-7, 2006, Proceedings. vol. 4331. Springer; 2006.
- [33] Vafaie N, Arvidsson M. Metrics to measure the impact of continuous integration: An empirical case study. 2015 May;.