Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

SCHEDULING OPTIMIZATION IN KUBERNETES

Relatore:
Chiar.mo Prof.
Saverio Giallorenzo

Corelatore:
Chiar.mo Prof.
Jacopo Mauro

Presentata da:
Simone Canova

I Sessione
Anno Accademico 2023/2024

Sommario

Con l'avvento del Cloud Computing, sono emersi nuovi prodotti software, con l'obbiettivo di semplificare tutte le fasi di creazione di un prodotto integrato con il Cloud, dal design al deployment. Uno di questi prodotti è Kubernetes, un Container Orchestrator. Kubernetes ha rivoluzionato il processo di creazione e mantenimento di applicazioni in Cloud, creando nuove architetture software, come i Microservizi o l'architettura Serverless. Ci concentriamo sul suo scheduler, un componente critico di Kubernetes, poiché responsabile di allocare le risorse per i microservizi. In questa tesi, commentiamo il funzionamento di Kubernetes, ed in particolar modo il suo scheduler, e proponiamo un possibile miglioramento per esso. In questa tesi, presentiamo un possibile miglioramento allo scheduler, che rende possibile scheduling ottimale quando lo scheduler di default non è in grado di trovare sufficienti risorse per allocare i microservizi richiesti. Per fare ciò, usiamo due solver ottimali, Z3 e OrTools. Mostriamo come solo OrTools sia adatto a questo compito, poiché Z3 risulta troppo inefficiente, e con cluster con al più 16 Nodi, questo approccio fornisce buoni risultati.

Abstract

Since the rise of Cloud Computing new software products emerged, with the goal to simplify all the different stages of the creation of a product integrated with the Cloud, from its design to its deployment. One of these products is Kubernetes, a Container Orchestrator. It has revolutionized the process of creation and management of Cloud applications, generating new software architectures, like the Microservices and Serverless architectures. We focus on its scheduler, a critical component of Kubernetes, because it is responsible for assign resources to microservices. Kubernetes' Scheduler is fast, but the choice it takes are not always optimal. In this thesis, we present a possible improvement to it, that enables optimal scheduling when the default scheduler is unable to find enough resources for the requested microservices. We use two optimal solvers to do this, Z3 and OrTools. We show that only OrTools is a viable option for this task, because Z3 is too inefficient, and for clusters with up to 16 Nodes, this approach gives good results.

Contents

Introduction

Thanks to the continuous evolution of Cloud Computing, the possibilities for developing and maintaining software products have grown exponentially. In particular, the advent of Cloud Computing enabled a service-based architecture for an application, compared to a monolithic one. In the past, the most used architecture was the monolithic one, where the application was a single block of various highly integrated components. On the contrary, a service-based architecture makes it possible to create an application composed of various individual components of various sizes, connected to each other. This has the great advantage of being able to change the configuration of the application more easily, and, thanks to Cloud Computing, it allows you to manage and maintain complex applications with simplicity.

However, a service-based architecture also comes with disadvantages. Developing an application as a set of services makes it much more difficult to manage and maintain it. This is why in recent years there have been new technologies developed, which have made it easier to design, develop, deploy and maintain applications in the Cloud and beyond. The product we focus on is Kubernetes, a container orchestrator originally developed as an internal Google product. We define its components in detail, and also do an in-depth analysis of its functionalities, its problems and its shortcomings, and we present a possible improvement. In particular, we focus mainly on its scheduler.

The scheduler in an operating system is the component that takes care of choosing which process to grant resources to perform its task. For Kubernetes, instead, the scheduler is the component that chooses where to place Pods. This means that a

change to the scheduler is reflected on any Kubernetes cluster, independently of what Kubernetes is used for, making it a critical component.

Kubernetes is open source, and it is developed by thousands of people. In particular, the scheduler works by calling a series of plugins, and you can add your own.

In this thesis, we present a plugin that enables the scheduler to make optimal decisions, avoiding decreasing its performance when not necessary. We use an optimal resolver, in particular, we decided to compare Z3 and OrTools. The first is a SAT solver from Microsoft and the second is a lazy constraint solver from Google.

We defined an optimization problem incrementally, by analyzing Pods by their priorities. By imposing a series of constraints and both minimization and maximization objective functions we were able to provide a solution to the scheduling problem. The goal is to schedule Pods that the default plugin is unable to schedule, while removing (and moving) the least amount of Pods.

Finally, this thesis is based on the work done by Professor Saverio Giallorenzo, Professor Jacopo Mauro and I, during my Erasmus+ for Traineeship in Denmark. After seeing that there were some good results we decided to try to publish a paper about it. In fact, the paper is a slimmer version of this thesis, in particular of the last chapter, where we present our solution.

1 Microservices and Kubernetes

In this chapter we explain all the aspects necessary to understand our work. We start from the background necessary to understand the problem we are trying to solve. Furthermore, we explain some part of Kubernetes that are necessary to understand some design choice we made, based on its internal structure.

1.1 Cloud Computing

In recent years, there has been two phenomena that crossed each other. The first is the continuously improving Internet network infrastructure, and also the fact that access to a fast and reliable Internet connection has become widely available. These two factors make possible new use cases for Internet applications, where users need to be able to access applications and services outside their home/office. The second one is a demand for raw computing power available when needed, without having to manage a real machine or to carry one along.

Cloud computing is becoming the de-facto standard for these use cases. It is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user [3]. Cloud computing is more and more popular today thanks to fast and reliable Internet connections that are now available to everyone. This type of architecture has also other advantages, if you are developing a product, as a developer, you can use someone else's infrastructure and host on their servers your application. Particularly, you do not need to have one or more servers, and employees to manage them, but you

can use the resources of someone else, which is probably better managed compared to a home-made solution. This type of service is called Infrastructure as a Service, or IaaS for short.

Not all applications are suitable to be designed around Cloud Computing, but most applications and services used by most people every day are based on Cloud Computing. So, it is really important that the infrastructure where all these applications and services are based on is secure, fast, and reliable. Furthermore, it is also extremely important that developers understand how to design their programs for this type of architecture. These have been improvement in this direction, with language like Jolie [8]. These languages are designed for developing distributed applications based on microservices (more on this later), and help developers write optimized and secure application for the cloud.

In recent years, there has been an increase in the amount of request for this type of programming languages and technologies, because the request for application based on Cloud Computing is increasing. Developer are asked to design applications around the principles of Cloud Computing, but this is difficult, because you need to account not only for the functional part of your application, but also for the deployment part. This is the reason why in recent years there has been an increase in the usage of technologies like Docker and Kubernetes, that enable developers to write the application for a single, virtual, system, and let Docker handle the difficult part of deploying on different machine.

Reasons like these are the main force that push researchers and companies to try new ways to improve the existing systems, and to create new, hopefully better, ones.

1.1.1 How to Deploy an Application

When you are developing an application, you need to choose between making a native application for the platform you are developing on, or making a non-native one. A native application has the advantage to be, usually, easier to develop, and it is faster on the platform you plan to use it.

On the other hand, if you develop your application in such a way that you can use it on multiple platforms it is more difficult, and the finished application can be slower than a native one. The advantage is, of course, that the application can be used on multiple platforms, without additional work. By doing so, you can distribute your application to more final users. In recent years for this last type of development and deployment of applications, particularly in the cloud, there has been two approaches.

Virtual Machines

The first approach to a more general deployment procedure, which is also the oldest one, is to use a Virtual Machine. A Virtual Machine (or VM) is the virtualization or emulation of a computer system [31]. With a VM, you can emulate a complete Operating System, and run multiple application on them. You do not need to worry about different type of architecture, of different kind of Operating System, since you can run an Operating System on a machine with a different one installed.

A Virtual Machine works by having a software component called HyperVisor, that translates in real time all the instruction from an architecture to another one. By using this component, you can emulate different architecture on your machine.

Their best use case is with legacy application or with software that needs a specific set of dependencies, for example Matita. It is an experimental proof assistant developed by the Computer Science Department of the University of Bologna that is also used by students on their first year in their Bachelor's Degree. The suggested way of using this software is by downloading a Virtual Machine image that contains a Debian Linux system, and all the packages required to run Matita.

Another use case is when you want to test applications, but you do not want to install them on your local machine. It is mainly done for security concerns, but can be also the case for other type of applications.

Using Virtual Machines to deploy applications is not really a good idea. You are giving up a lot of performance, because you need to emulate a whole new machine, and you have a lot of overhead by the HyperVisor. Virtual Machines are extremely useful for some tasks, and are widely used for Kubernetes' cluster.

Containers

Containers are the newest way to deploy applications, particularly in the cloud. Containerization is operating system-level virtualization or application-level virtualization over multiple network resources so that software applications can run in isolated user spaces called containers in any cloud or non-cloud environment, regardless of type or vendor [4].

By using an Operating System feature called namespaces, a containerized application is able to connect to other containerized application if needed, but not to other part of the Operating System. This is extremely useful, because, compared to Virtual Machines, the footprint on the system is much lower. Without the need to virtualize an entire Operating System a containerized application can be lightweight, both for the occupied space and for the speed impact.

In recent years, there has been a major containerization technology that has been used: Docker. Now it is considered the de-facto standard for containers. Docker provides a suite of development tools to make it easier for developer to develop, deploy, and use applications.

One problem with containers is security, without a thick layer of separations some vulnerabilities of the container can transpose to the main Operating System. Another limitation is the storage management. Containers are intended to be used for applications that do not need persistent storage, called "stateless applications" (without a state). You can use them with persistent storage, but it comes with its own set of challenges and problems.

Using a container for your application, however, comes also with advantages. Apart from being able to run application on every computer that supports Docker, you

deploy only the application, not a whole Operating System. The application is faster compared to the same application deployed on a Virtual Machine, because it is more optimized. The last big advantage is the space requirement, containers are extremely lightweight, so they can be downloaded when needed. Containers have revolutionized they way developer design, develop and deploy their applications, conducting to new types of architectures for the cloud.

Microservices and Serverless

Containers are essential for cloud computing, where you can deploy your application as a microservice or a serverless function. Microservices is an architectural pattern that arranges an application as a collection of loosely coupled, fine-grained services, communicating through lightweight protocols [15]. Not all types of applications benefit from this architecture but, with cloud computing, there has been more demand for applications based on this architecture.

There has been also another type of architectural pattern that emerged for simple, lightweight, applications: the Serverless architecture. Serverless computing is a cloud computing execution model in which the cloud provider allocates machine resources on demand, taking care of the servers on behalf of their customers [29]. With this architecture you do not need to deploy your whole application and its every part, but you can call part of it on demand. Kubernetes is essential for this approach, you deploy some parts of your application as serverless functions, with a single, lightweight container. This last step is transparent for the developer, in particular, most of the time, he can design your serverless function through the cloud provider interface.

Containers have revolutionized the way developer think and develop their applications, and, tied with Cloud Computing, changed also the way we use these applications. But, with these new architectural patterns, the system administrators now that need to manage multiple servers with multiple services on it. Even if the services are deployed through containers, the system administrators need new tools to do their work at best.

1.2 Kubernetes

When we take into account the deployment of various applications through containers the biggest problem is managing them. To partially solve this problem, a Container Orchestrator like Kubernetes (or K8s) can be used. In system administration, orchestration is the automated configuration, coordination, and management of computer systems and software [20]. In Cloud Computing, the most used software for managing containers is Kubernetes.

1.2.1 Container Orchestrators

A Container Orchestrator is used to automate the deployment and management of different containers. It works by creating a cluster with different Nodes, where it automatically places Pods on them. The Nodes can be real machines or a series of Virtual Machines. The system administrator that is managing a Kubernetes' cluster only needs to provide some `.yaml` files that describe the wanted behavior. After providing these files, Kubernetes manages the deployment of all the requested Pods.

Developing and managing fast, reliable and secure application with these new architectures, like Microservices and Serverless, is difficult, that is why tools like Kubernetes and Jolie are essentials to do so.

1.2.2 Basic Knowledge

In this last part of this chapter, we provide more in depth information about Kubernetes and some of its part. This information is important to understand the next chapters, particularly to understand why we took some design decision with the creation of your system to improving the Kubernetes Scheduler.

Nodes and Pods

Kubernetes runs your workload by placing containers into Pods to run on Nodes [18]. A Node may be a virtual or physical machine, depending on the cluster. Usually, a physical machine is split up in equally capable Virtual Machines, this is important for how we structured the plugin, as discussed later.

Pods are the smallest deployable units of computation that you can create and manage in Kubernetes. A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers [23]. There are also different abstractions on top of Pods, like Jobs or Replica Sets. A Job represents a single run of multiple instance of the same Pod, while a Replica Set describes how many instance of a Pod you would like to always have in your cluster.

Also, to better manage the cluster, Kubernetes provides the Priority and the Affinity features.

Priority

Kubernetes lets you assign a priority to every Pod you want to schedule [22]. With this option, it is really easy to manage a situation where you have more important Pods and less important ones, but priority assignment can be difficult in some cases because you can cause unwanted behavior, like having Pods in starvation.

To ease managing big clusters, Kubernetes provides also the feature called "Priority Class", used to assign the priority to one or more Pods without a specific value, but with the name of the Priority Class. By doing so, you can change the numeric value of the Priority Class and all the Pods will update their priority.

Priority is extremely useful, also because Kubernetes provides Preemption for Pods. The default scheduler provides only Preemption for a single Node, because it is unable to make a Preemption on multiple Nodes, but there are efforts to make possible also a Preemption on multiple Nodes [26].

Preemption is useful because it allows to remove a Pod from a Node to make space for a more important Pod. However, it can cause some problems; when you make preemption the Pod that needs to remove is "gently" killed. When the Pod receives this signal it has 30 seconds to terminate gracefully, otherwise it is removed with force. These 30 seconds windows is called "grace period", and the length can be increased or reduced.

Affinity

When you define a Node, you can bind to it some labels, and when you define a Pod, you can bind to it some Affinity and Anti-Affinity labels.

There are two type of Affinity:

1. Node Affinity: Kubernetes compares the labels of the Pod with the labels of the Node.
2. Inter-Pod Affinity and Anti-Affinity: Kubernetes compares the labels of the Pod with the labels of other Pods.

In the first case, with Node Affinity, you can define where to place Pods. For example, if you have a Pod that would benefit if it would be scheduled on a Node with an SSD, compared to an HDD, you can label the Nodes with an SSD with `ssd-disk`. And by also labeling the Pod with `ssd-disk` as an Affinity Label, Kubernetes will schedule the Pod only on the respective Nodes. The Anti-Affinity label system works in the opposite way.

In the second case, with Inter-Pod Affinity and Anti-Affinity, you can define where to place Pods. For example if you have a Pod that would benefit to be scheduled on the same Node of another Pod, you can label both of them with the same label. By doing so, you are sure that both Pods, if both of them are schedulable, will be scheduled on the same Node. The Anti-Affinity label system works in the opposite way. Both of the type of Affinity can be set as "soft" or "hard" constraints, so the cluster administrator has a lot of options to optimize the various Pods inside the Nodes of the cluster.

Kubernetes Components

When you deploy a Kubernetes' cluster, there are multiple components that work together without a custom configuration. Every cluster has at least one worker Node but, to really use Kubernetes you need more than one Node.

The worker Nodes host the Pods that are the components of the application workload. The control plane[1] manages the worker Nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple Nodes, providing fault-tolerance and high availability [10].

As represented in Figure 1.1, inside each Kubernetes' cluster there are the Control Plane and one or more Nodes. Each Node has its own `kubelet` and `kube-proxy` components. The first one is an agent that is responsible to actively run each Pods and check if they are still running. The second one is a network proxy that maintains network rules on Nodes and allows network communication to the Pods from network sessions inside and outside the cluster.

The Control Plane is a set of single components with various tasks. It can be placed on a dedicated non-worker Node or inside one of the worker Nodes of the cluster. Its components make global decisions about the cluster, as well as detecting and handling cluster events, like errors. The component inside the Control Plane that we will focus on is the scheduler.

The scheduler is responsible to select where to place Pods, based on a set of rules and the resources of the cluster still available. The default scheduler is called `kube-scheduler`, and we will analyze it more in the next chapter.

[1]The container orchestration layer that exposes the API and interfaces to define, deploy, and manage the life cycle of containers.

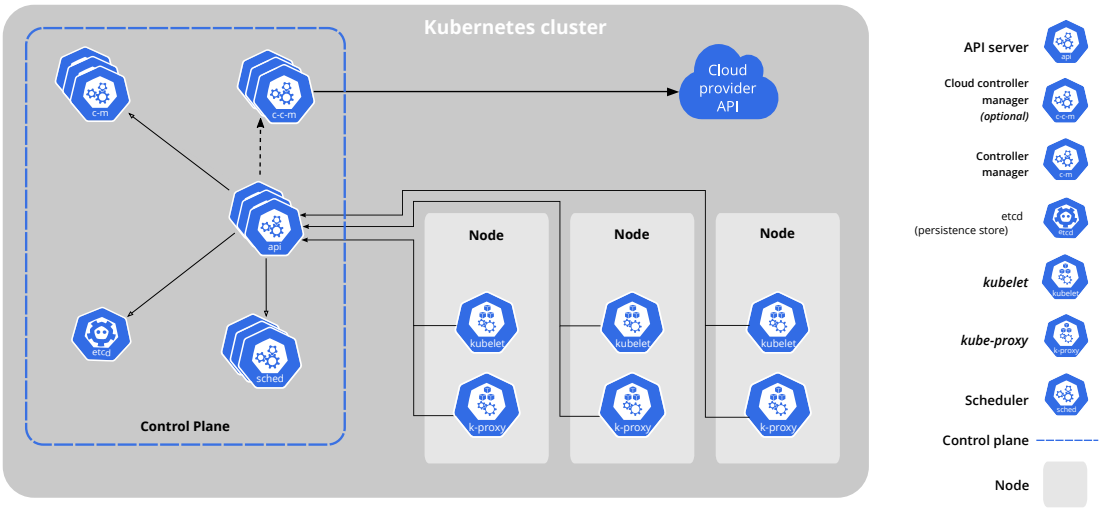

Figure 1.1: Kubernetes Components, from the official documentation on `kubernetes.io`.

2 Kubernetes' Scheduler

In this chapter, we focus on the default scheduler of Kubernetes, some of its feature, and some of its limitations. We also include an overview of how the Scheduling Framework works for `kube-scheduler`, Kubernetes' default scheduler.

2.1 Default Scheduler

In Kubernetes, scheduling refers to making sure that Pods are matched to Nodes so that `kubelet` can run them [11]. As presented briefly in the previous section, `kube-scheduler` is Kubernetes' default scheduler. It is part of the control plane, and its task is to select an optimal Node to run newly created or unscheduled Pods.

In a cluster, Nodes that meet the requirements for a Pod are called "feasible" Nodes. From the list of feasible Nodes, `kube-scheduler` selects the best Node based on a number of factors. So, the Scheduling operation is done in two phases: the Filtering and the Scoring phase.

The Filtering phase is responsible to find every possible Node where the Pod can be scheduled. It does not only check for space and resource requirements, but also for Affinity and Anti-Affinity constraints.

The Scoring phase is responsible to find the best Node where the Pod can be scheduled. Given that the Affinity and Anti-Affinity constraints can be hard or soft constraints, they are important also in this phase. There are two ways to configure the Scoring phase, by defining Scheduling Policies or by creating Scheduling Pro-

files. We focus on the second option, because Scheduling Policies, as explained in the official documentation [28], are not supported anymore since Kubernetes v1.23. Also, we want to discuss the Scheduling Framework, because it is what allows us to create a plugin and not a completely new scheduler.

2.1.1 Scheduling Profiles

A Scheduling Profile allows you to configure each part of the scheduling cycle for `kube-scheduler`. Each stage is exposed by an extension point.

You can configure `kube-scheduler` to run more than one profile. Each profile must have an associated "scheduler name" and can have different behavior. Pods that want to be scheduled according to a specific profile can include the corresponding scheduler name in their `.spec.schedulerName` when defined.

2.1.2 Performance Tuning

You can also modify the behavior of the scheduler, by changing its configuration file. This is easier than defining a Scheduling Profile, but is less versatile. It is used only when you want to use a different scoring strategy for every Pod, or when you need the most general approach. For example, you can set a threshold for how many feasible Nodes the scheduler have to look for before stopping, and raking only the one found. It is useful because if you have a cluster with a lot of Nodes you can avoid to search through all of them to find all feasible ones. After the scheduler have found a certain number of feasible Nodes the Filtering Phase ends, ignoring some other, possible, feasible Nodes.

Or, as described before, you can use a Scheduler Profile. With that, it is really easy to modify how the scheduler will place Pods in the cluster, based on a number of factors. This option is more versatile, and let us configure more precisely the scheduler. There are two options that are described in the official documentation because are the ones that partially solve the Bin Packing problem, making the scheduler (almost) optimal.

Most Allocated Strategy

The simplest way to show how to tune the scheduler is by creating a scheduler configuration with different strategy for the `NodeResourcesFit` plugin. We will describe what a scheduler's plugin is later, for now it is sufficient to know that they exist, and can alter the scheduling cycle. The first option is to use the `MostAllocated` strategy. This strategy scores Nodes based on the utilization of their own resources, favoring the ones with higher allocation. For each resource type, you can set a weight to modify its influence in the Node score.

When the cost is per machine, and not per resource, the `MostAllocated` strategy is extremely useful. This strategy is not the default one, and we found an example of a company that recompiled the default scheduler to make `MostAllocated` the default scoring strategy [2]. They had to do so because their Cloud Provider would not let them configure the scheduler, but only use a different image for it.

Requested to Capacity Ratio Strategy

Another scoring strategy that can be used is the `RequestedToCapacityRatio` strategy. It allows the users to specify the resources along with weights for each resource to score Nodes based on the request to capacity ratio. It favors Nodes according to a configured function of the allocated resources.

2.1.3 Other Features

Furthermore, there are other features that change the way the scheduler works. The first one is Cluster Zones. They provide a way to partially split large clusters, thus enabling better management.

Another one is the use of Multiple Scheduler. They can work in parallel, because you can define for each Pod which scheduler to use. The usage of Multiple Scheduler is possible, but it is advised only to advanced users, because it is really easy to create situations that you are unable to control.

2.2 Scheduler Framework

The scheduling framework is a pluggable architecture for Kubernetes' scheduler. It consists of a set of "plugin" APIs that are compiled directly into the scheduler. These APIs allow most scheduling features to be implemented as plugins, while keeping the scheduling "core" lightweight and maintainable [27].

2.2.1 Overview

The framework's workflow has two cycles, the Scheduling Cycle and the Binding Cycle. During the Scheduling Cycle the scheduler find the most appropriate Node where to place the Pod. After that, the Binding Cycle binds the Pod to that Node, so that, at the next Scheduling Cycle, the scheduler is aware that a portion of the Node chosen is reserved to the previous scheduled Pod.

As already described, the Scheduling Cycle is composed of a Filtering phase and a Scoring phase. Figure 2.1 is the visual representation of how the Scheduler Framework works.

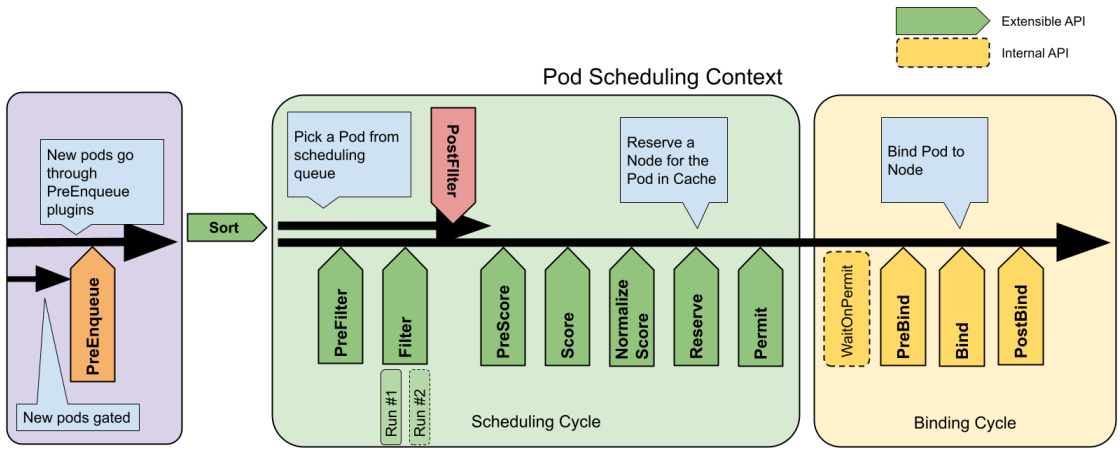

Figure 2.1: Scheduling Framework Extensions Points

The arrows are called Attach Points (or Extension Points), and they define where the scheduler can be extended. There are also two Attach Points that are placed before the Scheduling Cycle. They are responsible for analyzing new Pods and ordering the unscheduled Pod queue.

2.2.2 Attach Points

In this part, we explain some of the Extension Points of the Scheduler Framework. There are also extension points not visible from Figure 2.1, but they can be used to customize even more the behavior of the scheduler.

2.2.3 Before the Scheduling Cycle

These plugins are called prior to adding Pods to the internal active queue, where Pods are marked as ready for scheduling. They can be marked as `Schedulable`, and go into the Scheduling Cycle, or `Unschedulable`, and go at the end of the queue. In some particular cases the Pods can be handled differently, and go into a special queue.

PreEnqueue

This Extension Point is used to make some checks to the Pod before it goes in the queue of Pods ready to be scheduled. When all the plugins that implement this Extension Point return `Success`, the Pod that has been analyzed goes into the queue. Here, there are checks that can be done before entering into the Scheduling Cycle. They are useful because, if one of these checks fails, the Pod is not marked as `Schedulable` nor as `Unschedulable`, and it goes in a special queue.

QueueSort

These plugins are used to sort Pods in the scheduling queue. While for other Extension Points there can be more than one plugin active, for the QueueSort Extension Point there must be only one.

2.2.4 The Scheduling Cycle

If everything up until this point is successful, we start with the Scheduling Cycle. The Scheduling Cycle includes both the Filtering and the Scoring phase.

PreFilter

Plugins that implement this Extension Point are always called, before the "real" start of the Filtering phase. They are usually used to pre-process information about the Pod, or to check some conditions about the cluster. There can be more than one plugin that implement the PreFilter Extension Point, but if one of them returns an error, the Scheduling Cycle is aborted.

Filter

This Extension Point is used to actively do the Filtering phase. The Plugins that implement the Filter Extension Point are responsible to filter our Nodes where the Pod cannot run. There can be more than one Filter Plugin, and the Nodes that the first one marks as infeasible are not checked by the second Filter Plugin, and so on. To speed up this part even more Nodes may be evaluated concurrently.

PostFilter

After the Filter phase, there is the PostFilter Extension Point. Plugins that use this Extension Point are called only if the Filter plugins marked all Nodes of the cluster as infeasible. The typical PostFilter plugin, which is active by default, is a preemption plugin.

Based on the assumption that plugins attached to this Extension Point are called only when a Pod cannot be scheduled for lack of feasible Nodes, we decided to design our plugin based on this feature. We will place our plugin after all PostFilter plugins, so that it gets called only if the default scheduler or other's plugins have failed to find a feasible Node.

Score

Plugins that implements this Extension Point are responsible for actively do the Scoring Phase of the Scheduling Cycle. They rank each feasible Node left by the Filtering Phase. There can be multiple Score Plugins, and each of them can have its own different weight.

NormalizeScore

For each Score plugin, there can be a NormalizeScore plugin connected. It is used to adjust the scores of each Node before completing the final ranking. The scores of each Node are the ones calculated by the corresponding Score plugin. Compared to other "secondary" Extension Points, if there are any error for one of the NormalizeScore plugins, the Scheduling Cycle is aborted.

Reserve

Plugins that implements this Extension Point must define two methods: `Reserve` and `Unreserve`.

The `Reserve` method is called before the scheduler actually binds a Pod to its designed Node. It is used because the scheduler works in parallel, so there can be race conditions. If one `Reserve` method fails, the following plugins are not executed and the Reserve phase is considered failed.

The `Unreserve` method is called only if the Reserve phase, or a following one, fails. If this happens, the `Unreserve` method of the corresponding plugin is invoked. This method should reverse its corresponding `Reserve` method. It is important to define it correctly because it may not fail, otherwise the scheduler can, temporary, crash.

There can be more Reserve plugins, and to be sure that, in case of any error, the `Unreserve` method will not fail, the `Unreserve` methods of different plugins are called in the reverse order of the corresponding `Reserve` methods.

2.2.5 The Binding Cycle

After the best Node has been chosen, the Binding Cycle began. With binding, we mean the active process of instructing the Node that one Pod must run on it. Then, `kubelet`, the component on every Node responsible to run Pods on it, will obtain the images required by the Pod, and start running it.

As showed by Figure 2.1, this cycle is composed of three Extension Points: the PreBind, the Bind, and the PostBind. PreBind plugins are used to prepare the cluster (and the selected Node) before binding the Pod to the Node. If during one of these PreBind plugins there is an error, the Pod is rejected and placed again in the Scheduling Queue. PostBind plugins, instead, are called after the all Binds plugin.

Bind

Bind plugins are used to actively bind a Pod to a Node. These plugins get called after all PreBind plugins have terminated.

The Bind plugins can choose if they want to handle the Pod or not. If one Bind plugin does not bind the Pod, another Bind plugin gets executed, until one plugin get successfully executed, and the Pod bound. If all Bind plugins decide not to handle the Pod the Binding Cycle gets canceled, and the Pod goes back in the Scheduling Queue.

3 State of the Art

In this chapter, we discuss what are the problems with the default scheduler, what is the state of the art research about Kubernetes' Scheduler, and in what direction the research is moving. The starting point for this work are two surveys about possible optimizations for the scheduler, an older one from 2018 [32] and a more recent one from 2023 [1].

3.1 Problems with the Default Kubernetes' Scheduler

Before discussing the state of the art research, we need to understand why we need improvements to the capabilities of the scheduler of Kubernetes. The main reason to find new ways to improve the scheduler is to make it faster, but this is not the only reason.

The first one is make the scheduler more configurable. This is important because there is no way to create a general scheduler that works perfectly with every workload and every cluster, unless you can configure it to work in a certain way. There are already lots of plugins to add more ways to customize the scheduler, but they are not all complete, and, more importantly, not all ready to be used in production.

Another important aspect is to make the scheduler better by allowing it to make better decision. For example, by improving the number of Pods that the scheduler is able to place inside the cluster, or placing them in a different position to get better schedules when new Pods arrive. This is what we are trying to do with our work.

Lastly, the default scheduler is not perfect. There are complaints from the community about design decisions because, to get a really fast scheduler, there have been decisions that generated problems in some situations. The one tied with our work, that we took as an example, is "necessary" starvation. In some cases where you have a big enough Pod, and it does not get placed inside the cluster for a couple of times, the scheduler places it at the end of the queue, generating starvation. Regarding this problem, there is an open issue [30] on GitHub, where the designers of Kubernetes wrote that they decided to accept total starvation for Pods in certain situations, so the cluster does not get blocked by a single, big Pod.

3.2 Areas of Study

Based on the two surveys, authors focus on a single aspect of the scheduler, so there are three main, general areas of study:

1. Optimization based on the architecture of the cluster;
2. Optimization for improving the automatic scalability of Kubernetes;
3. Totally New Scheduling approaches.

Global improvement for the scheduler are rarely investigated in the open-source community. These improvements are not production ready, and aim to try new things that can help in very few scenarios, but these are still really important works, because most innovation is based on this research.

3.2.1 Architectural Optimization

One possible approach is to understand better the architecture of the cluster, and, based on that, change the default scheduler. When talking about Architectural Optimization, the main focus is on the Fog Architecture (or Edge Architecture).

The Fog Architecture is an architecture that uses edge devices to carry out a substantial amount of computation (edge computing), storage, and communication locally and routed over the Internet backbone [6]. This is used a lot for Internet of Things (IoT) applications, when you have a few low-power devices on-site that send all the data retrieved to a server off-site, through Internet.

There are situations where you want to deploy a Kubernetes' cluster on-site, but you need a powerful computer to elaborate all the data, so this architecture is really useful. On this topic, the research is more generic, not only tied to Kubernetes, but it is applicable to more scenarios. The main bottleneck with this type of architecture is the network status, mainly its speed and latency. For example, a group of researchers implemented an extension of the default scheduling mechanism for Kubernetes, enabling it to make resource provisioning decisions based on the current status of the network infrastructure [25].

There is also some research on the scheduling capabilities of the cluster that collects network traffic information in real time and allocates computational resources proportionally to the distribution of network traffic [17]. The main difference with this approach is to dynamically change the cluster dimensions to accommodate for bigger or smaller Pods.

There are also some authors that tried to optimize the scheduling capabilities of Kubernetes based on the power consumption [9] and the usage of green energy utilization [7]. These are interesting extensions but not useful for improving the scheduling capabilities of the scheduler.

3.2.2 Autoscalability Optimization

There are situations where there is the need for or the desire to change frequently the configuration of the cluster, both horizontal and vertical. A cluster is expanding horizontally when the number of Nodes is growing, and vertically when the amount of available resources of one or more Nodes is growing.

In these scenarios, the scheduler needs to be aware of these changes. It needs to be responsive to the addition or the removal of one or more Nodes, and, if desired, reschedule some Jobs to optimize the load balance. The automatic scaling feature is also used with GPU clusters, where the training of neural network models requires frequent changes to the hardware and the size of the cluster.

3.2.3 New Scheduling Approaches

There are three different possible approaches for scheduling:

1. Heuristic and Meta-Heuristic;
2. Machine Learning;
3. Mathematical Modeling.

The default scheduler uses only heuristic techniques, it is fast, and also configurable. But, while good for a general purpose scheduler, there are better alternatives for specific use cases.

There are also Meta Heuristic Algorithms proposed for Kubernetes' clusters, and they are all described in the survey paper [1]. These Meta Heuristic Algorithms can be divided into two main categories:

1. Evolutionary algorithms (EAs), such as Genetic Algorithm (GA);
2. Swarm intelligence algorithms, such as Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), and Whale Optimization.

The default scheduler uses some of these techniques and algorithms, but the researchers that tried to implement more advanced techniques and algorithms have found that the new scheduler was not as fast as the default one, and it was less configurable. Configuration, in fact, is a big advantage of Kubernetes' Scheduler, because it allows the handling of different kinds of Pods based on some parameters.

There is, however, one specific area where there are more efforts to improve the scheduling capabilities of Kubernetes' Scheduler, in the last years in particular: Machine Learning. Machine Learning is not used inside Kubernetes' Scheduler, but Kubernetes is used to do Machine Learning workload. So, particularly in the last years, there has been a big effort in make cluster "learn", in some way, how to have better scheduling capabilities. Here, with "better", we mean faster and more precise, assigning Pods to Nodes which can probably handle and finish their task.

The biggest example is from the DL2 system [21]. DL2 uses the data of the default scheduler to train a new scheduler based on a Deep Neural Network, and use reinforcement learning to improve it based on much information received by the default scheduler. After some time, the default scheduler is replaced by the DL2 scheduler, which should be faster and more precise. The authors of DL2 claim improvements over the default scheduler up to 44.1% in terms of average job completion time.

The problem with this approach is that, if the workload or the cluster structure changes, the scheduling performance of the Deep Learning scheduler will be terrible. The solution is to use Hybrid Learning. It is the idea behind using a Neural Network Model to solve a task, trained by the data obtained by another system, for example a heuristic one. You use the Neural Network Model until performance goes below a certain point, and then you go back to a heuristic system and re-train the Neural Network Model, usually not from scratch.

This solution is really powerful with Kubernetes' clusters, because, for example with GPU clusters, the workload is similar and the cluster identical for the whole task, so the trained model can be reliable for a long period of time. But when the task changes, or the cluster is re-sized, you still have the default, heuristic scheduler to fall back to.

The last possible approach is to use mathematical modeling to make an optimal scheduler. This technique requires using an optimal solver to get an optimal solution to the allocation problem. It is really powerful and, with recent solvers, there have been an increase in performance, reflected also in real world scenario. Unfortunately, this approach has a big drawback: it is not as fast as the default scheduler, and it does scale exponentially the bigger the problem gets. So, there have been only few attempts to use this technique to improve the scheduler, like with the SAGE [14] paper or the BOREAS [13] paper.

While not as fast as heuristics, schedulers based on this technique are essential to validate the precision of a heuristic scheduler. For this reason, it is important to continue improving these solutions.

3.2.4 Difference with our proposal

As already stated, the main fields that are trying to improve Kubernetes' Scheduler are two. The first one is Machine Learning, and the second one is Autoscalability. In these fields, there are a lot more papers that try to improve the scheduler.

We, instead, decide to use the mathematical modeling technique, but not in the usual way. Unlike with SAGE and BOREAS, where they call an optimal solver for each scheduling task, we try to use the optimal solver the least amount of time, given how bad it is compared to a heuristic scheduler in terms of time.

4 Using an optimal solver to improve the scheduler

In this chapter we present the idea behind using an optimal solver to obtain a more precise scheduler. While the work is still not usable in a real Kubernetes' cluster, we made some benchmarks from random generated examples and how this could scale with bigger systems.

In Kubernetes, as we have seen, there is a really fast scheduler, and there are several papers and plugins that try to make it better. The problem is that the scheduler can give you sub-optimal schedules, and in tight-up scenarios this could be a problem. In fact, the default scheduler, as we have seen in the previous section, can even lead to Pod starvation; this is intended behavior as specified in a GitHub issue [30], and there is no intention of changing this. Therefore, while the scheduler is fast and precise in most scenarios there are clusters that would benefit a lot with a slower but more precise scheduler.

There is some research to achieve that, see the Boreas [13] and the Sage [14] papers, however the problem with using always an optimal solver could make the scheduler really slow. Given this issue, we started searching how to improve the scheduler. In the next subsections we present different ideas that we explored.

4.1 What we are trying to solve, and how we intend to do it

We want to bring optimality to the scheduler, but we want to keep the feature rich default one. In fact, other attempts have been made in this direction, but while they can be used in a real Kubernetes' cluster (unlike ours) they are very limited in their customizability. The main problem with other projects similar to this one, like Boreas [13] and Sage [14], are that they try to be optimal with every schedule that the cluster needs. This means that there is a lot of unnecessary latency, for two reasons:

- Because most of the time there is no need to solve a whole optimization problem to find a schedule, the simple and fastest solution found by a heuristic is also the best one;
- Because you can call the optimizer only when you really need it.

The last point is the one that made us think about trying something new. Other works tried to use optimal solvers for every iteration; here, we try instead to call the optimal solver only when it is really needed. This approach results in having all the features of the default scheduler, while having the plugin to fall back when the scheduler is unable to make a Pod allocation. This is our idea more in depth.

Figure 4.1 explains when the plugin would be called. When a Pod needs to be scheduled the default scheduler try to find a feasible Node with enough resources to run the Pod. After the filtering phase if there is at least a Node in the set of Nodes found the plugin is not called. But, if the default scheduler is unable to provide a feasible Node to schedule the Pod, our plugin would be called. After its execution in case of success we move all the necessary Pods to get to the solution found by the plugin, skipping all the following phases of the default scheduler.

We would attach the plugin to the `PostFilter Phase` of the scheduler, so that it gets called only when the scheduler is unable to provide a location to place the Pod.

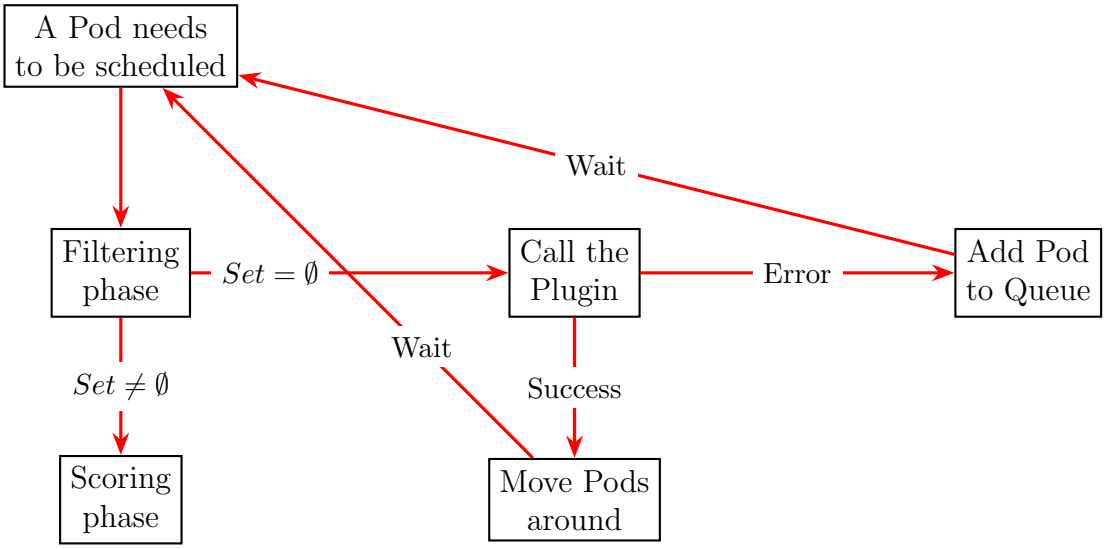

Figure 4.1: Plugin execution diagram

Notice that in case of an error, so when the plugin is unable to provide a solution, we mimic what the default scheduler would do, so we place the Pod in the queue of the Pods that needs to be scheduled. At the next scheduling request the scheduler will sort the queue, without any downtime for the cluster.

Another choice we made was to check for all the Pods available, already scheduled in the cluster and in the queue. In this way we call the solver the least amount of time, without calling it again for the next Pod.

4.1.1 Description of the problem

We start by assuming that:

- We know all the information about each Pod and Node;
- We know how many resources a Pod needs to complete its execution.

Then, we define $B = b_1, \ldots, b_n$ as the set of Bins, and $P = p_1, \ldots, p_m$ as the set of Pods. Finally, we define the variables:

$$x_{i,j} \quad \forall i \in P, j \in B$$

These variables indicate if the Pod i is allocated on the Bin j. These are boolean variables, but, to make the constraints easier to work with, we define them as integer variables with two constraints:

$$x_{i,j} \geq 0 \quad \forall i \in P, j \in B$$
$$x_{i,j} \leq 1 \quad \forall i \in P, j \in B$$

We need also to account for the affinity label of each Pod, so:

$$x_{i,j} \leq 0 \quad \text{if } l \notin b_j.\text{label} \quad \forall i \in P, j \in B, l \in p_i.\text{affinity}$$

And of course also for anti-affinity:

$$x_{i,j} \leq 0 \quad \text{if } l \in b_j.\text{label} \quad \forall i \in P, j \in B, l \in p_i.\text{anti-affinity}$$

We also want that the sum of the requested resources by the Pod in each Bin do not exceed its capacity, so for RAM:

$$\sum_{i \in P} x_{i,j} \cdot p_i.\text{ram} \leq b_j.\text{ram} \quad \forall j \in B$$

And for CPU:

$$\sum_{i \in P} x_{i,j} \cdot p_i.\text{cpu} \leq b_j.\text{cpu} \quad \forall j \in B$$

Lastly, each Pod can be assigned at most to a single Bin:

$$\sum_{j \in B} x_{i,j} \leq 1 \quad \forall i \in P$$

Optional Constraints for Symmetry Breaking

Our problem is similar to the Bin Packing one, and there are a lot of equivalent solutions. We do not need to check solutions that are equivalent. With the additional constraints we found, we check much fewer cases when the cluster respect certain characteristics. In our case when there are equal Nodes and when there are equal Pods. This also result in a faster plugin.

With SMT solvers this process is called *Symmetry Breaking*, and in some cases it helps reduce the research space by finding additional constraints that can reduce the number of possible cases to analyze, while keeping all the possible solutions still valid. We found two additional constraints that we think are really well at reducing the research space. These are constraints that are not necessary, but in some scenarios they break several symmetries and help to reduce the research space.

We started from the assumption that if you work with a container orchestrator like Kubernetes, when you create a cluster you do not rely on real machines, but you create Virtual Machines (VMs) from single real machines. Moreover, except in some specific cases, these VMs are all equal; i.e., they have the same amount of RAM, storage, and CPU available. Lastly, usually, instead of creating single Pods by hand for a service, the users create them through Replica Sets or Jobs [33].

Knowing this information, we add two similar constraints, one for Bins and the other for Pods, they both create an "order" based on their index. In this way we do not allow equivalent solutions, and the research space is greatly reduced.

For example, for Pods, the formula is:

$$\forall p_1 \in P, p_2 \in P \mid p_1.\text{index} < p_2.\text{index} \wedge p_1 = p_2$$
$$\Rightarrow$$
$$x_{p_1,b_1} \leq x_{p_2,b_2} \quad \forall b_1 \in B, b_2 \in B \mid b_1.\text{index} < b_2.\mathit{index}$$

And for Bins:

$$\forall b_1 \in B, b_2 \in B \mid b_1.\text{index} < b_2.\text{index} \wedge b_1 = b_2$$
$$\Rightarrow$$
$$\sum_{p \in P} x_{p,b_1} \cdot p.\text{ram} \leq \sum_{p \in P} x_{p,b_2} \cdot p.\text{ram}$$

The Python code is a little bit more clear:

```
# Redundant Costraint for equivalent pods
# If two pods are equivalent we impose and "order" so that the one
# with the lowest index must be placed first
for p1 in pods["index"]:
    for p2 in pods["index"][p1+1:]:
        if pods["ram"][p1] == pods["ram"][p2] and
            pods["cpu"][p1] == pods["cpu"][p2] and
            pods["priority"][p1] == pods["priority"][p2] and
            pods["affinity"][p1] == pods["affinity"][p2] and
            pods["anti-affinity"][p1] == pods["anti-affinity"][p2]:
            for b1 in bins['index']:
                for b2 in bins['index'][b1:]:
                    o.add(x[(p1, b1)] <= x[(p2, b2)])
# Redundant Costraint for equivalent bins
# If two bins are equivalent both for cpu and ram we impose that
# the cpu and ram allocated to the one with the lowest index is
# less or equal than the one on the other side
for j in bins["index"]:
    for j2 in bins["index"][j:]:
        bin_index = j-1
        bin_index2 = j2-1
        if bins["ram"][bin_index] == bins["ram"][bin_index2] and
            bins["cpu"][bin_index] == bins["cpu"][bin_index2] and
            bins["label"][bin_index] == bins["label"][bin_index2]:
            amount_packed_j =
                sum(x[(i, j)] * pods["ram"][i] for i in pods["index"])
            amount_packed_j2 =
                sum(x[(i, j2)] * pods["ram"][i] for i in pods["index"])
            o.add(amount_packed_j <= amount_packed_j2)
```

Cycling through Priorities

After these initial constraints, we analyze the Pods of each priority separately. First, by maximizing the number of Pods allocated for that priority. So, for each p in the set of priority, from the one with the highest value to the one with the lowest value, we do:

$$max \sum x_{i,j} \quad \forall i \in P, j \in B \mid p_i.\text{priority} = p$$

Then, we fix the number of Pods allocated to be at least the value found, named k, so we add the constraint:

$$\sum x_{i,j} \geq k \quad \forall i \in P, j \in B \mid p_i.\text{priority} = p$$

Then, we minimize the amount of Pods moved:

$$min \sum (1 - x_{i,p_i.where}) \quad \forall i \in P, \mid p_i.\text{priority} = p \wedge p_i.\text{where} \neq 0$$

We fix the number found, named t, and pass onto the next priority:

$$\sum (1 - x_{i,p_i.where}) \leq t \quad \forall i \in P, \mid p_i.\text{priority} = p \wedge p_i.\text{where} \neq 0$$

In both cases, we impose that the value found must be greater/less or equal to k/t, because a value that is not exactly the one we found is still good for our purpose. This is the current implementation, in the section about Z3 there is an explanation of the first idea we proposed.

4.1.2 Implementation

We started by wanting to create a plugin for the Kubernetes' Scheduler Framework, so that it could be used also in real clusters. The first thing was to study how to make one with the information present in the GitHub page dedicated to out-of-the-tree plugins [12].

Following the little documentation about that was really difficult, because the only official source of information is a markdown file with little to no instruction on how to effectively make the plugin work. We found a few articles [2] and blog posts [24] of people that have done it, but they wanted to do a different thing from ours. They referred to a previous version of the project architecture, and we could not get the docker image of the new scheduler working.

Knowing this, we decided to focus on studying the logic of the scheduling and write a Python program to try if the idea works and if the results are good, and then try to port that into the real scheduler. This idea proved to be feasible also because Z3 and OrTools have both libraries for a multitude of languages, we picked Python because it is a good language for prototyping.

4.1.3 Integration with the Plugin Framework

While for now there is not a real plugin, we thought about this system keeping in mind that in the future this could become a real plugin. This is a description of how the system would work.

As already explained previously, we suggest linking the optimizer to the `PostFilter` attach point, so that the plugin would be called only when really needed.

In particular, the system would send the information about the cluster to the solver, and because the plugin needs to be written in Go, and Go does not have a way to work with other languages (apart from C), we would need to send the data through a JSON file, or something similar. After this, the plugin would need to elaborate all the data obtained, and then respond with a new allocation or with an error:

- In case of an error, the schedule cycle would continue, probably placing the Pod that was unable to schedule in the end of the queue of the unscheduled Pods;
- In case of success, the system would handle the new schedule accordingly to what the solver has found.

4.2 Z3

We started working with Z3, an efficient Satisfiability Modulo Theories (SMT) solver from Microsoft Research [34].

We initially chose this solver over others because of its incremental feature. You can push a "tag" inside the stack and then pop all the thing added until that tag. The only problem with this approach is that after a pop operation the model does not retain any information about a possible solution found at that time, so we need to recompute a solution. This means that we can structure our optimization problem incrementally.

4.2.1 Previous formalization

We took the decision to use Z3 because our first idea would have benefited from the incrementality of the solver. We would start with the usual initial constraints, and for each priority we would:

1. Impose that all the Pods of that priority would need to be allocated.
2. If that is the case, we would then minimize the amount of Pods moved from the initial configuration, and add a constraint to make sure that at most that amount of Pods would move.
3. If not, we would pop the latest constraint series placed, and instead we would:
   (a) Maximize the number of Pods of that priority allocated.
   (b) Store the value found and add a constraint to make sure that at least that amount of Pods would always be placed.
   (c) Minimize the number of Pods of that priority allocated.
   (d) Store the value found and add a constraint to make sure that at most that amount of Pods would move.

Ultimately, we decided to move to the new version, for two reasons:

- This way, we need to call the solver fewer times. In fact, if the problem is solvable with all the Pods allocated, the maximization problem should find the maximum solution really fast, or block altogether.
- We decided to try and compare the performance and usability of Z3 with OrTools, and OrTools does not provide a native way of making it incremental. We tried to recreate every time the solver from scratch, saving with lambda function the constraints placed before. That resulted in degraded performance, and decided to improve the optimization problem.

With these elements, we started developing the Python script to verify if the optimization problem is structured correctly.

4.2.2 Problems we encountered

Time

The first big problem we encountered was time, if the instance of the problem we are trying to solve is not easy to solve, nor easy to mark as not solvable, the solver continues to try to find a solution. This is true in general, but especially for a maximization problem, where the solver continues to try to find a better solution because it needs to explore all the research space.

This is a big problem, considering that this software would ideally be part of the scheduler, so we decided to fix a time limit for each call to the solver. Fortunately, the Z3 Python library expose a `set()` method of the solver object, which we used to set a timeout. In Z3, this works by having a time limit on every time the `check()` method is called. This is a parameter that can be set as an argument when the script is called from the command line.

We need a time limit because this solver is included inside a scheduler, and we cannot block the scheduling capabilities of the cluster indefinitely. We decided to include a timeout for every time the `check()` method is called, mainly for two reasons:

1. It is the most reliable way to impose a limit, as the timer is inside the library and not on top of it;
2. We can track how much time the overall schedule take, and can retract other information more easily. This is the reason why we did not want to make an overall timer.

Push and Pop

The second big problem we encountered was the way that Z3 handles the previous solution found with the usage of push and pop. While, on one hand, this feature simplifies a lot the writing of the constraints of our problem, on the other hand, Z3 does not memorize the previously found solution when using the `pop()` method. Therefore, every time that we need to pop of the last portion of the stack, we need to call again the `check()` method, with two implications:

1. We can find a different solution that the one previously found. This is not really a problem because we do not fix the exact solution found, but it can lead to confusion;
2. If the solution is found almost when the time limit is reached, the next time could pass over the time limit, so the program would crash. It is really a big problem, but the workaround found for the next point partially solves also this problem.

Custom Minimize and Maximize

The third big problem was the way that Z3 handles the minimization and maximization process. Z3 outputs a solution only if that is the optimal solution, not a feasible one. This is a problem because, as previously stated, there are instances

of this problem that have really large research space, and to mark a solution as "optimal" the solver has to travel through all the research space. With a timeout, this does not work well.

We are fine also with a feasible solution, we want to extract them even if they are not optimal, but the library does not allow it. So, we wrote a wrapper around the `maximize()` method that simply runs the solver without the maximization objective function, adds the constraint that the value of the objective function must be more that the one just found, and then repeats the process.

However, also this solution comes with some minor problems, because if the time limit hits on the last call we need to go back, and for the problem at the previous point we need to recompute the solution just found.

This is not perfect, and there is an alternative. The library exposes a `lower()` and an `upper()` method on the objective function. These should return the lower and upper bound of the objective function, but in our case this did not work. And there are a lot of other users of the library that are reporting that these methods do not work as expected, they always throw a runtime error. In any case, in addition to that, when a solver like this try to find a solution it does not "travel" only across feasible solutions. This is the reason why Z3 does not expose a partial solution, and so we need these workarounds to use that like we want.

4.3 OrTools

After some results with Z3 we were not satisfied about its performance. We decided to try to use also OrTools [19] for our problem. Besides trying a different solution, we can also compare performance between the two solvers. This would make our results stronger, and hopefully better than Z3.

This is the fastest solver we could think of, and it exposes a Python library that we can use inside the same program we use Z3.

Given that is all inside a Python program, we do not need to expose via JSON or in other formats the results, and we can plot the results of the comparison inside the same program.

4.3.1 Try to use OrTools like Z3

We started by writing the counterpart code for OrTools, following the same optimization problem as in Z3. But OrTools is not an incremental solver, so we need a way to recreate the solver again every time we want to delete the last portion of the stack. We managed to make a working prototype, storing lambda function in a list of lists for each portion of the stack, and re-calling them every time we need to recreate the model. This was tedious, slow, and inefficient, so we tried new possibilities.

4.3.2 New optimization problem

The way we structured our problem was caused by the incrementality feature of Z3. Trying to use OrTools as an incremental solver like Z3 was not a good idea, so we came up with the structure explained in Section 4.1.1, the one we propose. By doing so, we do not need the incrementality of the solver, and it is better also for Z3, because we need to call the solver fewer times.

The solutions that our program outputs are equal, they only require less time to be computed. Doing so, also reduces the problems we had encountered with Z3, so we were satisfied with this solution.

4.3.3 Comparison between OrTools and Z3

In this part I (Simone) will talk about my experience with using an optimal solver for a real world problem. After writing the same program to solve the same optimization problem with two different solvers I can write something about the experience you would get if you pick one or the other for a project.

Performance

Both OrTools and Z3 can solve big problems really fast, but there is no doubt that OrTools is faster. I checked also part of the source code of both of the library when debugging, and for what I have seen, Z3 is more high-level than OrTools. In OrTools everything is non-mutable, and so when you add a variable inside the solver you get an object that references it, but you do not have the ability to modify it. This was the reason that we could not find a good solution for using OrTools like Z3, which is incremental.

On the other hand, this means that OrTools is a lot faster than Z3, by design. It is considered extremely performant compared to the alternatives. In the MiniZinc Challenge 2022, OrTools placed first [16] in 3 out of 4 categories, it is known for being one of the fastest lazy constraint solver available. Also, with the most recent versions you can also change the backend that it uses under the hood and switch to other ones. This is not available with Z3 at all.

Flexibility and ease to use

For straightforward problems, OrTools is faster, and probably better overall, but if you need to write complex problems, that involves backtracking and undoing constraints, Z3 is clearly better. In my experience, writing a complex problem with Z3 was really easy, also because it comes more natural, you can more or less describe the problem with code. On the other hand, writing not-so-simple problems with OrTools is more difficult, if not impossible.

This was my opinion just after the change of the optimization problem structure, after more time working and debugging problems with both of them I think that, while OrTools has undoubtedly fewer features, it is easier to use compared to Z3. OrTools methods and function (at least in the Python library) are much more clear, and follow the same rules for, more or less, all cases. In the Z3 Python library, instead, there are a lot of different approaches, I think that this is caused by the age of Z3 compared to OrTools, but now I am not so sure which solver I would pick for a similar project.

Another thing is debugging, with Z3, it was really easy to find where I made a mistake, and how to solve the problem, because there are a lot of ways to analyze the internal structure of the model during every phase of the process. With OrTools, instead, there is not a real method to analyze the model and all of its constraints, you can analyze the value of the variable after the solver has finished checking if a solution exists. For most of the problems that is fine, but in our case the debugging part was really tedious, and having a nicer interface with the model would have helped.

The last thing is the type hinter, in both of their Python library it is unpleasant to use, partially because Python does not have a complex type system, and partially because the type system of the library, of Z3 in particular, is not well-structured, and gives more errors than it should. This is not a good thing, and I hope that the respective libraries for languages like Java or C++ have better type checking support, but I have not tried those.

In conclusion, I would recommend trying first OrTools, and see if you can do everything you need with it, otherwise trying something like Z3 or other solvers would be a good idea.

4.4 Benchmarking

In this part, we talk about how we have done the benchmarks, and why we took some design decisions.

First, we analyze our system to generate random examples. To do a good suite of benchmark and have reliable results, we need a way to create an arbitrary number of cases with different boundaries. The main problem in doing so is to find a way to create cases where our system would be actually called. Our system would only be called if the default scheduler would fail to schedule the next Pod, so generating completely random examples is not a viable option.

First and foremost, we designed the system in such a way that everything would be configurable, from the amount of verbosity in the printing phase to the ratio between

the total amount of resources needed by the Pods and the amount of resources of the Bins. Having this, we created a script that calls our system for all the necessary cases, using the same random generated example both with Z3 and with OrTools. This is done 50 times per configuration, and with this we have a fair amount of confidence that our test are reproducible with similar results. Also, we test a lot of possible configuration, in particular we decided to test each combination from a series of values for each of these settings:

- Timeout, in particular these values: 1 second and 10 seconds;
- The ratio between the total amount of resources needed by the Pods and the amount of resources of the Bins, in particular these values: 0.8, 0.9, 1 and 1.1;
- The number of Bins, in particular these values: 4, 8, 16, 32;
- The amount of Pods compared to the number of Bins, in particular these values: 4 and 8;
- The amount of different priority possible for the Pods, in particular this value: 2 and 5.

In total, we have $2 \times 4 \times 4 \times 2 \times 1 \times 50 = 6400$ different instances to try with two different solvers. Before showing the results, we describe more in depth how we generate the examples.

4.4.1 Heuristic for random generated examples

To benchmark the plugin, we simulate a heuristic scheduler and, after that, call the plugin. We decided to use a *First Fit Decreasing* heuristic that is simple to implement, but still gives us decent performance compared to the optimal case [5].

Also, we decided to ignore for the random generated examples the affinity and anti-affinity feature of Pods, mainly for two reasons. First, because generating them reliably would require more work, and second, because they would only generate additional constraints.

4.4.2 Choice of values

For the timeout, we want to check not to high values, because the timeout is tied to each solver call, so the overall time required by the plugin would be a higher, in the order of $2 \times$ *number of priorities* $\times$ *timeout*. Because for each priority the plugin call the solver two times, one for with a maximization problem and one with a minimization problem. We wanted also to try with 100 seconds, but the amount of instances that the solver was able to solve with the additional times were marginal.

For the ratio between the total amount of resources needed by the Pods and the amount of resources of the Bins we thought, originally, to check also 0.7 as a value. But we found out that finding a seed that would generate an example where our heuristic would fail to schedule all the Pods was not possible. So, we decided to exclude it from the list, because the heuristic of Kubernetes is more advanced than ours, and probably it would schedule every Pod too.

Also, we did not check examples with more than 32 Bins, because, as we will show, the plugin already struggles with 32 Bins, and big clusters are not the one that would benefit from this plugin.

We decided, also, to check two different values for the ratio Pods/Bins. By doing so, we can observe if the size of Pods influences the timing of the plugin.

Lastly, we wanted to check how the plugin would perform with an ideal scenario: where there are only 2 priorities, one for the cluster's Pods, and one for the user's Pods. But we also wanted to include a more advanced configuration, with 5 different priorities, one for the cluster's Pods, and four for the user's Pods.

4.4.3 Not all random generated examples are good

While creating the generator of random examples, we noticed a problem, there were cases where, if the workload ratio was too little, the heuristic would find a solution for all Pods. These cases are not the one where our plugin would be called, because it would be called only if the default plugin would be unable to schedule a Pod.

We decided then to generate beforehand the seeds that will be used to generate the random examples, by checking if for each configuration there are some Pods that are not scheduled after the heuristic scheduling phase. After this list of 50 seeds is created, we call the plugin for every combination of the different possible spec of the cluster and keep all the results in a CSV file. By using the same seed, we can really compare the performance of the plugin for each different combination of specs.

4.4.4 Results and comments

For the 3D histograms we present, the vertical axis represents the percentage of instances. More precisely, the yellow part of each bar represents the percentage of the instances where the plugin found an optimal solution. Instead, the green part of each bar represents the percentage of the instances where the plugin found only a better solution. All other cases are not represented, because the plugin would discard them.

For the wireframe 3D graph we present, the vertical axis represents the amount of milliseconds the plugin used to find a solution. The value is the average of all the 50 instances, for each configuration. Where the plugin was unable to finish, we kept the biggest value found of the 50 instances, or, if higher, the theoretical maximum value, defined at the start of Section 4.4.2.

OrTools vs Z3

The first analysis that we can do is a direct comparison between Z3 and OrTools. As showed in 4.1, only in 23 out of 6400 test cases Z3 is better than OrTools. Most of them are because Z3 spent some milliseconds less, compared to OrTools, to find the solution.

The comparison is based by, in this order, the amount of Pods added, removed, moved, and, finally, the overall time spent to find a solution. Inside the number of instances where they performed the same, there are also the cases where they were not able to provide a solution, and reached the time limit before a partial solution

was found. For those instances the library of both solvers raise an exception, that is catch by the plugin and an error is returned.

$$\text{Number of Instances where Z3 is better} = 23 \tag{4.1}$$

$$\text{Number of Instances where OrTools is better} = 3395 \tag{4.2}$$

$$\text{Number of Instances where Z3 and OrTools performed the same} = 2982 \tag{4.3}$$

$$\text{Total Number of Instances} = 6400 \tag{4.4}$$

The other comparison that we can do is based on how many times OrTools and Z3 are able to find a better, or optimal solution to the instance. For this comparison we use a 3D histogram, in both these 2 histograms the timeout is 10 seconds, the ratio of Pods/Bins is 8, and the max Priority for Pods is 2. On the left there is OrTools, on the right Z3.

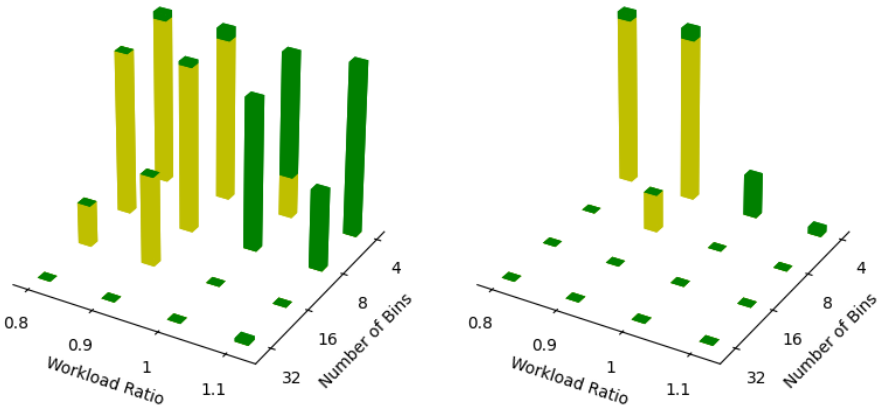

Figure 4.2: Difference between OrTools and Z3

As we can see in Figure 4.2, with a timeout for each solver call of 10000 seconds, Z3, compared to OrTools, performs terribly. The percentage of instances where Z3 is able to provide a solution to the problem are high only for small clusters, while OrTools performs way better.

Given these results, for the version of the plugin we are planning to release and publish, we will propose only OrTools as the solver.

Cluster Characteristics

Another difference we noticed was the difference that the ratio of Pods/Bins made. With the other parameters set with the same value, a ratio of Pods/Bins is 8 instead of 4 means that, on average, the Pods are half the size, while being double the amount. We noticed, as presented in Figure 4.3, that a higher ratio of Pods/Bins resulted, for smaller clusters, in the optimal solution found more often. This happens because it is easier to place more, smaller Pods inside one or more Bins. On the other hand, with more Bins, the number of constraints with more Pods is higher, and probably surpass this advantage, so the total number of solved instances is lower. There is probably a sweet spot, where the size of the Pods is optimal compared to the dimensions of the Bins, but finding this spot is beyond the scope of our work.

For this comparison we use the same 3D histogram, in both these 2 histograms the timeout is 10 seconds, the solver is OrTools, and the max Priority for Pods is 2. On the left with 4 as the Pods/Bins Ratio, on the right with 8.

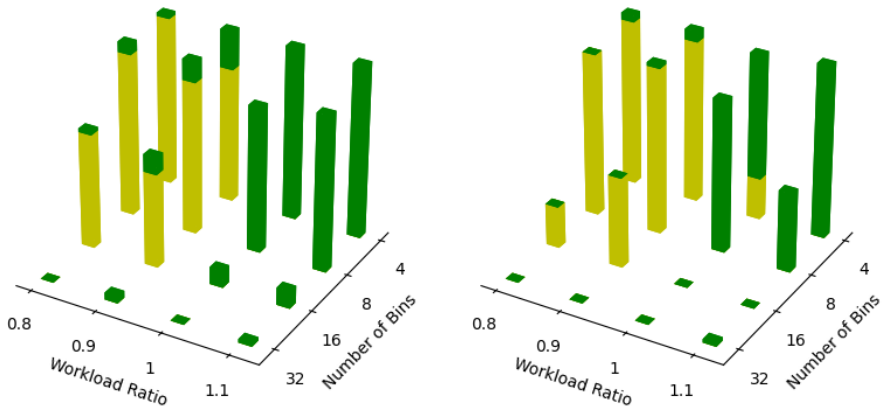

Figure 4.3: Difference with 4 and 8 as the Pods/Bins Ratio

Timing

The first comparison we can make is how much the timing allowed to use by the solver impacts the performance of the plugin. In Figure 4.4, the only difference between the two histograms is the timeout given to the solver. With a higher timeout the plugin is able to find a solution to more instances of the problem, but only by a small portion and only with some combinations of factor. Particularly, with cluster with 16 Bins, 64 Pods, and a Workload Ratio of 0.8 or 0.9.

For this comparison we also use the same 3D histogram, in both these 2 histograms the ratio of Pods/Bins is 4, the solver is OrTools, and the max Priority for Pods is 2. On the left with 1 second as the timeout, on the right with 10 seconds.

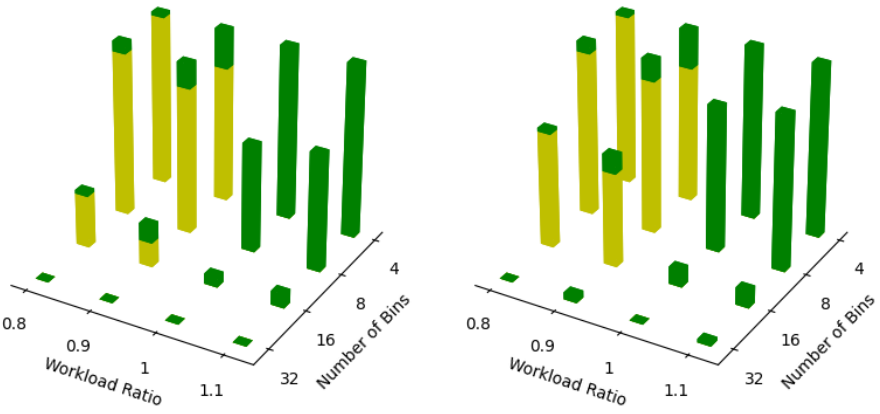

Figure 4.4: Difference in Timeout

Apart from 4 instances out of 6400 combinations, the plugin, when it was able to finish, outputted a solution that is better than the starting situation. This is why we did not represent in the graph the finished percentage, because they are almost always the same as the better one.

Particularly, after seeing that, we decided to place a check inside our plugin, after the solution is calculated. We want to make sure we are always going to provide a solution that is better than the situation before, otherwise we do not provide a solution for the cluster.

Furthermore, as presented in Figure 4.5, as we expected, the time necessary, on average, to find a solution increases with bigger clusters. The curve growth seems linear, but with more test it would probably become evident that it is exponential. Furthermore, with a higher workload, while the number of solved instances is smaller, the amount of time necessary to solve does not grow so quickly.

In particular, Figure 4.5 represent the combination of OrTools, Pods of 2 priorities, a ratio of Pods/Bins of 4 and a timeout of 1 second. The vertical axis indicate the time spent by the plugin in milliseconds.

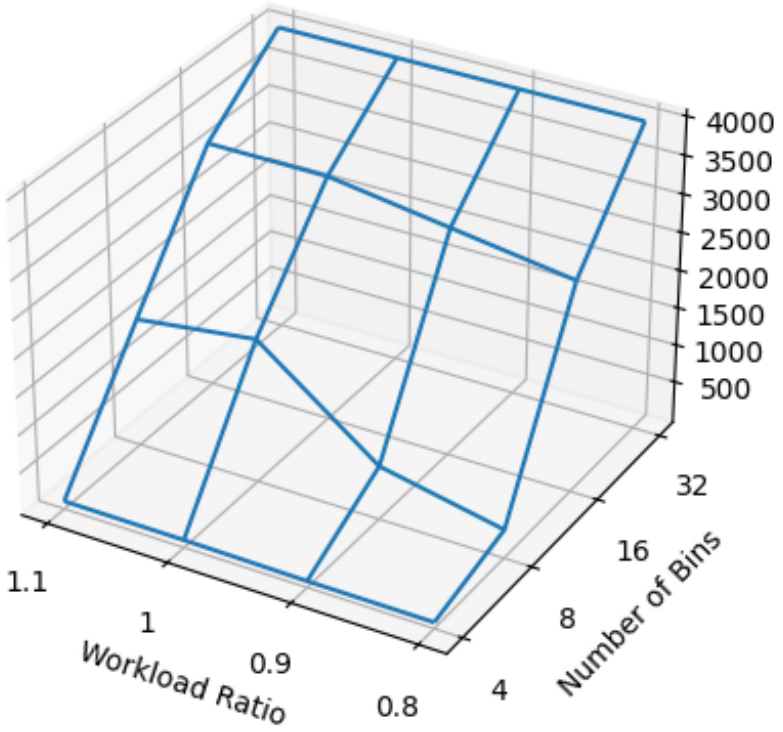

Figure 4.5: Wireframe Representation of Average Time Spent

The system is not perfect, we think it can be still improved and this is only a starting point. By having this results, we think that the main use case for this plugin would be clusters with a low amount of Bins and Pods, but that are really tight on resources. Another type of cluster that would benefit from this system would be a cluster with low dynamism, because results improve with bigger time limits for the solver, and you do not need a really fast scheduler for those clusters.

In any case, these results show that OrTools is better than Z3 for this problem, basically in every situation, and this corresponds to our predictions.

Priority

The last comparison is about Priorities. In particular, we tested how the scheduler would perform with the same settings, but with Pods with 2 and with 5 priorities. One thing to notice is that the highest priority Pods should represent the system's Pods, so they are always scheduled, but they can be moved.

As presented in Figure 4.6, it seems that, assuming we have the same Pods but with more Priorities, more Priorities gets better results. Particularly, for the amount of instances solved with an optimal solution. The graphs show instances with OrTools, a ratio of Pods/Bins of 4 and a timeout of 10 seconds. On the left with 2 priorities, and on the right with 5 priorities.

But, after a deeper analysis, this is not the case. These results are influenced by the current implementation of the timeout. The timeout implemented is not a global timeout, it is a timeout for each solver call. By designing the system like this, the overall average timeout that the plugin has with an instance with Pods of 5 priorities is, approximately, four times as much compared to 2 priorities. It is four times as much because the highest priority does not count, the Pods with it must always be scheduled. In a future version of the plugin the timeout should be global, so comparisons like these can be done correctly.

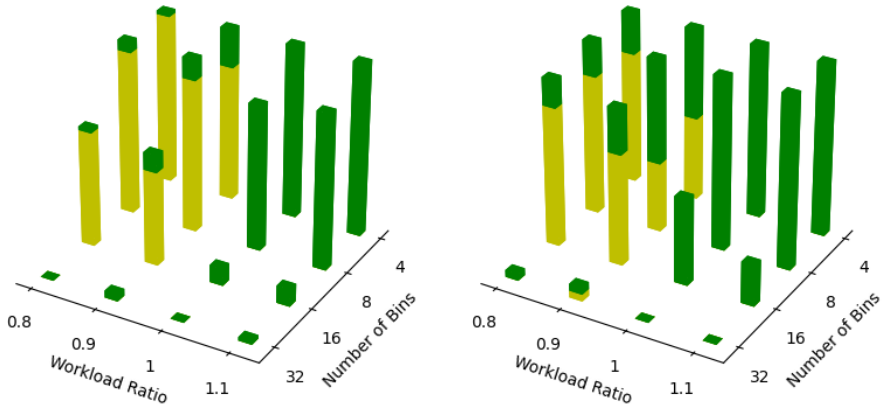

Figure 4.6: Difference in Priorities

4.4.5 Future Work

In this section we propose some ideas and possible improvements related to our work, and what we think are the next steps for it.

The first thing would be to find a way to handle the timeout better. It would be better to have a global timer for the whole plugin execution. This is not difficult, but there are edge cases that we need to address, and it was beyond our goal.

Another thing that is not perfect is the handling in Z3 of partial solutions, we wrote custom wrapper function for the maximize and minimize phase in Z3, but we are not satisfied by it. We think that there is a better solution that can be found, and also here there should be a timer for how long the custom wrapper function was executing, and tying that to the timeout of the solver. However, we are not interested in following this path, because, as we have shown, OrTools is better.

There is also one of the original ideas, that is writing a real plugin for Kubernetes, and connect it to this system by exchanging the information about the cluster. If that works, we can open a pull request to the out-of-tree plugins repository [12] on GitHub, to share it with the very active community of Kubernetes. One of my colleagues, as its thesis, is working on doing a plugin for Kubernetes' Scheduler, based on our work.

As of now, to tune the system, you need to call it with the right flags *and* pass a JSON file as a parameter. In the future, there would be a single config file with all the information needed.

Finally, the last thing, which in our opinion is the most difficult one, is to find a way to move the problem completely to boolean variables, in this way, the solver can act like a boolean SMT solver, and not like a SAT solver. This should increase performance quite a bit, but only if the restructure of the problem is not a bigger bottleneck for its performance. We also are not sure if this is even possible, because there are some situations where knowing only if the solution exists but not having the possibility of extracting real values makes the system useless.

Also, there could be use cases that we have not thought of, given that this work started with the idea of a more precise scheduler for some use cases. Considering additional use cases can help us improve our system.

Finally, OrTools supports hints when working with the same problem. Another possible approach to improve the plugin would be to make the solver able to solve incrementally the problem, while being guided by hints.

4.5 Conclusion

Optimal solvers can be used inside the scheduler, and, for some particular cases, they should be used. But, assuming that Kubernetes is used mainly with cluster that are medium-sized, if not bigger, our particular system would not perform good enough. We still think that, while still being work in progress, a plugin like ours can improve the scheduling capabilities of the scheduler. Particularly, because the plugin would not be called always, but only when the default scheduler would fail. This is a big advantage for its adoption, because, even if turned on by default, in most cases you would not use it.

Furthermore, using an optimal solver like OrTools or Z3 should make possible a correct and functional implementation of the Cross Node Preemption Plugin [26]. At the time of writing it is not functional, because with big clusters it would make the scheduler too slow, but in combination with an optimal solver it could work.

Our implementation is far from perfect, and, currently, does not include a working plugin for the scheduler. Our goal was to try a different approach, and see if it would work. While the results are not as good as expected, we are still quite happy with how the plugin performed.

Bibliography

[1] Carmen Carrión. "Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges". en. In: *ACM Computing Surveys* 55.7 (July 2023), pp. 1–37. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3539606. URL: https://dl.acm.org/doi/10.1145/3539606 (visited on 09/20/2023).

[2] ClickHouse. *Saving Millions of Dollars by Bin-Packing ClickHouse Pods in AWS EKS*. en. URL: https://clickhouse.com/blog/packing-kubernetes-pods-more-efficiently-saving-money (visited on 04/11/2024).

[3] *Cloud computing*. en. Page Version ID: 1223981135. May 2024. URL: https://en.wikipedia.org/w/index.php?title=Cloud_computing&oldid=1223981135 (visited on 05/18/2024).

[4] *Containerization (computing)*. en. Page Version ID: 1221360097. Apr. 2024. URL: https://en.wikipedia.org/w/index.php?title=Containerization_(computing)&oldid=1221360097 (visited on 05/18/2024).

[5] György Dósa. "The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I) $\leq$ 11/9OPT(I) + 6/9". en. In: *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Ed. by Bo Chen, Mike Paterson, and Guochuan Zhang. Berlin, Heidelberg: Springer, 2007, pp. 1–11. ISBN: 978-3-540-74450-4. DOI: 10.1007/978-3-540-74450-4_1.

[6] *Fog computing*. en. Page Version ID: 1219850944. Apr. 2024. URL: https://en.wikipedia.org/w/index.php?title=Fog_computing&oldid=1219850944 (visited on 05/04/2024).

[7] Aled James and Daniel Schien. "A low carbon kubernetes scheduler: 6th International Conference on ICT for Sustainability, ICT4S 2019". In: *6th International Conference on ICT for Sustainability, ICT4S 2019*. CEUR Workshop Proceedings 2382 (June 2019). Publisher: CEUR-WS. URL: http://www.scopus.com/inward/record.url?scp=85067802264&partnerID=8YFLogxK (visited on 06/06/2024).

[8] *Jolie, the service-oriented programming language*. en. URL: https://www.jolie-lang.org/ (visited on 05/22/2024).

[9] Kuljeet Kaur et al. "KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem". In: *IEEE Internet of Things Journal* 7.5 (May 2020). Conference Name: IEEE Internet of Things Journal, pp. 4228–4237. ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2939534.

[10] *Kubernetes Components*. en. Section: docs. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 06/03/2024).

[11] *Kubernetes Scheduler*. en. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ (visited on 05/18/2024).

[12] *kubernetes-sigs/scheduler-plugins*. original-date: 2020-01-16T13:34:00Z. Mar. 2024. URL: https://github.com/kubernetes-sigs/scheduler-plugins (visited on 03/21/2024).

[13] Torgeir Lebesbye et al. "Boreas – A Service Scheduler for Optimal Kubernetes Deployment". en. In: *Service-Oriented Computing*. Ed. by Hakim Hacid et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 221–237. ISBN: 978-3-030-91431-8. DOI: 10.1007/978-3-030-91431-8_14.

[14] Vlad-Ioan Luca and Madalina Erascu. *SAGE – A Tool for Optimal Deployments in Kubernetes Clusters*. en. arXiv:2307.06318 [cs]. July 2023. URL: http://arxiv.org/abs/2307.06318 (visited on 09/21/2023).

[15] *Microservices*. en. Page Version ID: 1223072775. May 2024. URL: https://en.wikipedia.org/w/index.php?title=Microservices&oldid=1223072775 (visited on 05/18/2024).

[16] *MiniZinc - Challenge*. URL: https://www.minizinc.org/challenge2022/results2022.html (visited on 04/02/2024).

[17] Nguyen Dinh Nguyen et al. "ElasticFog: Elastic Resource Provisioning in Container-Based Fog Computing". In: *IEEE Access* 8 (2020). Conference Name: IEEE Access, pp. 183879–183890. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3029583.

[18] *Nodes*. en. Section: docs. URL: https://kubernetes.io/docs/concepts/architecture/nodes/ (visited on 06/03/2024).

[19] *OR-Tools*. en. URL: https://developers.google.com/optimization (visited on 04/01/2024).

[20] *Orchestration (computing)*. en. Page Version ID: 1224178459. May 2024. URL: https://en.wikipedia.org/w/index.php?title=Orchestration_(computing)&oldid=1224178459 (visited on 05/18/2024).

[21] Yanghua Peng et al. "DL2: A Deep Learning-Driven Scheduler for Deep Learning Clusters". In: *IEEE Transactions on Parallel and Distributed Systems* 32.8 (Aug. 2021). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 1947–1960. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3052895.

[22] *Pod Priority and Preemption*. en. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/ (visited on 06/03/2024).

[23] *Pods*. en. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 06/03/2024).

[24] Julio Renner. *K8S- Creating a kube-scheduler plugin*. en. July 2021. URL: https://medium.com/@juliorenner123/k8s-creating-a-kube-scheduler-plugin-8a826c486a1 (visited on 04/11/2024).

[25] José Santos et al. "Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications". In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. June 2019, pp. 351–359. DOI: 10.1109/NETSOFT.2019.8806671.

[26] *scheduler-plugins/pkg/crossnodepreemption at master · kubernetes-sigs/scheduler-plugins*. en. URL: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/crossnodepreemption (visited on 06/05/2024).

[27] *Scheduling Framework*. en. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ (visited on 03/21/2024).

[28] *Scheduling Policies*. en. Section: docs. URL: https://kubernetes.io/docs/reference/scheduling/policies/ (visited on 06/07/2024).

[29] *Serverless computing*. en. Page Version ID: 1222264115. May 2024. URL: https://en.wikipedia.org/w/index.php?title=Serverless_computing&oldid=1222264115 (visited on 05/18/2024).

[30] *Totally avoid Pod starvation (HOL blocking) or clarify the user expectation on the wiki · Issue #86373 · kubernetes/kubernetes*. en. URL: https://github.com/kubernetes/kubernetes/issues/86373 (visited on 03/21/2024).

[31] *Virtual machine*. en. Page Version ID: 1222494162. May 2024. URL: https://en.wikipedia.org/w/index.php?title=Virtual_machine&oldid=1222494162 (visited on 05/18/2024).

[32] Zhang Wei-guo, Ma Xi-lin, and Zhang Jin-zhong. "Research on Kubernetes' Resource Scheduling Scheme". en. In: *Proceedings of the 8th International Conference on Communication and Network Security*. Qingdao China: ACM, Nov. 2018, pp. 144–148. ISBN: 978-1-4503-6567-3. DOI: 10.1145/3290480.3290507. URL: https://dl.acm.org/doi/10.1145/3290480.3290507 (visited on 09/21/2023).

[33] *Workload Management*. en. URL: https://kubernetes.io/docs/concepts/workloads/controllers/ (visited on 03/26/2024).

[34] *Z3*. en-US. URL: https://www.microsoft.com/en-us/research/project/z3-3/ (visited on 03/20/2024).