

**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: _____ Разработка планировщика ресурсов _____

Выполнил:

Студент группы БПМИ204 _____

18.05.2022 _____

Дата



Подпись

С.Е.Енцов

И.О.Фамилия

Принял:

Руководитель проекта _____

Данил Андреевич Буланкин

Имя, Отчество, Фамилия

Ведущий разработчик, архитектор

Должность, ученое звание

ООО "1С"

Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки 19 мая 2022 _____

Оценка (по 10-ти бальной шкале) _____



Подпись

Москва 2022

Содержание

Введение	2
Интерфейс	3
Реализация	3
Эксперимент	4
Итог	5
Ссылки	6
Приложение. Пример кода	7

Аннотация

В этом отчёте описана реализация инструмента для изоляционного тестирования шедулера kubernetes, благодаря которому возможно исследование эффективности его работы в большом кластере без необходимости использовать реальный кластер kubernetes, что значительно упрощает проверку гипотез об улучшении алгоритма шедулинга и тестирование существующих решений.

Введение

Целью командного проекта было разработать модификацию шедулера kubernetes, алгоритма размещения подов (виртуализированных процессов) по нодам (виртуальным машинам), которая позволит более эффективно выполнять шедулинг. Моя роль в этом проекте состояла в разработке инструмента, с помощью которого можно исследовать эффективность алгоритма шедулинга с целью доказательства полезности модификаций и проверки гипотез об улучшении алгоритма. Более точно, была поставлена задача разработать инструмент, симулирующий шедулинг подов в кластере kubernetes с использованием заданной пользователем конфигурации шедулера, позволяющий получить полезную информацию об эффективности алгоритма шедулинга. При этом были определены следующие приоритеты:

1. Должна быть возможность симуляции шедулинга внутри большого кластера kubernetes.
2. Технические требования к компьютеру для проведения исследования должны быть минимальны.
3. Процесс шедулинга внутри симуляции должен быть максимально приближен к шедулингу подов реальным kubernetes.

Вышеописанные требования в совокупности гарантируют возможность исследовать алгоритм шедулинга на задачах, аналогичных реальным, включая шедулинг внутри крупномасштабного кластера, без необходимости использовать реальный кластер kubernetes, что может быть неоправданно дорого. Третье требование обеспечивает достоверность результатов исследования, полученных при помощи разработанного инструмента.

С учётом описанных требований было решено реализовывать инструмент для исследования как модуль внутри исходного кода kubernetes. Благодаря этому, появляется возможность обращаться к структурам, отвечающим за шедулинг в kubernetes, что упрощает реализацию логики, приближенной к реальному процессу шедулинга. При этом остаётся возможность не использовать значительную часть функционала kubernetes, не относящуюся к шедулингу, что так же упрощает разработку. Недостатком такого подхода является зависимость от исходного кода системы, который может значительно меняться от версии к версии, из-за чего усложняется миграция между версиями k8s. В качестве альтернативного подхода был рассмотрен вариант абстрагироваться от исходного кода системы и вместо этого взаимодействовать с шедюлером посредством https-запросов. Так можно было бы реализовать более универсальный продукт, но из-за недостаточной гибкости во взаимодействии с системой не было гарантий в том, что получится реализовать необходимый функционал, из-за чего этот подход был отвергнут в пользу вышеописанного.

Обобщая вышесказанное, данный отчёт описывает реализацию инструмента для исследования шедулера kubernetes (далее SIT от «scheduler isolation testing [tool]»), который позволяет изучить эффективность алгоритма шедулинга в рамках приближенных к реальным задач, не поднимая реального кластера. В секции 2 разобран интерфейс SIT и упомянуты его возможности. Секция 3 посвящена деталям реализации. В частности,

разобраны компоненты SIT и их взаимодействие со структурами kubernetes. В секции 4 продемонстрированы возможности SIT и разобран кейс его использования. В секции 5 подводятся итоги работы и обсуждаются возможности улучшения SIT.

Интерфейс

SIT позволяет пользователю проводить эксперименты, характеризующиеся двумя параметрами:

- Набор плагинов шедулера – определяют логику выбора нод при шедулинге.
- Сценарий эксперимента – последовательность запросов, обработка которых симулируется.

Запросы могут включать в себя любые типы взаимодействия с системой, влияющие на её состояние: например, шедулинг подов, изменение реального потребления ресурсов подами, добавление/удаление нод. Так как при реализации невозможно было учесть все типы запросов, которые могут понадобиться пользователю, предусмотрена возможность создавать пользовательские запросы. Подобный принцип расширяемости продукта активно учитывался при разработке, о чём подробнее рассказано в секции 3.

Итак, эксперимент характеризуется двумя параметрами. Плагины определяют поведение шедулера: сменяя наборы плагинов, можно узнать, какой из них лучше справляется с задачей, описанной сценарием. Сценарий определяет внешние факторы, влияющие на состояние системы; проведение экспериментов с разными сценариями показывает, как шедюлер в конкретной конфигурации справляется с различными задачами. Также в сценарии можно описать сложное поведение шедулера, которое невозможно описать плагинами: например, работу сервиса, который регулярно опрашивает ноды и перераспределяет поды, вызывающие нехватку ресурсов (подобный, но более простой пример, разобран в секции 4).

За запуск эксперимента отвечает функция `RunSchedulerIsolationTest` со следующей сигнатурой:

```
func RunSchedulerIsolationTest(plUGINS []PluginInfo, scenario RequestGenerator) []core.StateSnapshot
```

Она возвращает последовательность структур `StateSnapshot`, которые описывают состояние системы после обработки каждого запроса. Обработать полученные данные можно непосредственно после получения, используя Go, или используя более подходящий инструмент, предварительно выведя их из программы.

В приложении 1 выписан пример запуска простого эксперимента. В рамках сценария создаются 3 ноды, затем посылаются 16 запросов на шедулинг подов: для каждого пода настроен параметр `affinity`, в котором указано, что его лучше разместить на той ноде, где меньше всего подов такого же типа. Также указаны плагины, благодаря которым шедюлер поддерживает работу с `affinity` подов.

Реализация

Для симуляции шедулинга в SIT используется структура `Scheduler`, отвечающая за шедулинг в реальном kubernetes. Интерфейс, который `Scheduler` использует для взаимодействия с системой, заменён на аналогичный интерфейс, перенаправляющий запросы структуре `StateManager`, через которую ведётся взаимодействие с симулируемой системой. Ниже более подробно описан процесс проведения эксперимента.

Для запуска эксперимента используется функция `RunSchedulerIsolationTest`, принимающая два аргумента: набор плагинов шедулера и сценарий эксперимента. Плагины передаются в структурах, описывающих способ инициализации плагинов и стадии шедулинга, на которых плагины применяются. Сценарий представляет из себя генератор запросов к симулируемой системе. Сценарием может быть любая структура, реализующая интерфейс `RequestGenerator` с единственной функцией `NextRequest`: например, структура, содержащая явный список запросов и возвращающая их по-порядку. Запрос – это объект, в котором описывается действие над системой.

Исчерпывающая информация о симулируемой системе хранится в структуре `State`. Информация включает в себя конфигурацию нод, текущее время в симулируемой системе и описание эффектов, которые оказывает на систему каждый под. Для получения `StateSnapshot`, в котором отражено состояние системы, используется функция `GetSnapshot`, которая инициализирует пустой снэпшот, применяет к нему эффекты всех подов и возвращает полученный объект. Благодаря такой реализации, при проведении эксперимента возможно учитывать сложное поведение процессов, такое как потребление ресурсов согласно произвольно заданной

функции или завершение процесса в произвольный момент. Для создания подов со сложным поведением предусмотрен класс `PodBuilder`.

Структура `StateManager` является адаптером к `State` и при обработке запросов информирует подсистемы, используемые при шедулинге, об изменениях в системе. Структуры `Clientset`, `CoreV1`, `Pods`, `Nodes` реализуют интерфейс, который использует шедюлер для взаимодействия с системой, и перенаправляют запросы в `StateManager`. За инициализацию структур отвечают вспомогательные функции, вызываемые в `RunSchedulerIsolationTest`. Также реализовано несколько менее значимых структур, помогающих проводить шедулинг изолированно от нерелевантных подсистем.

Эксперимент

Для демонстрации функционала SIT была выбрана гипотеза, выдвинутая на начальных стадиях работы над проектом. Стандартные плагины шедюлера позволяют выполнять шедулинг с учётом количества потребляемых ресурсов подами: например, объёмом потребляемого CPU. При этом вместо реального уровня потребления используется величина, указанная в конфигурации пода, запрос ресурсов. Гипотеза заключается в том, что эффективность шедулинга улучшится, если вместо запроса использовать реальный объём потребляемых ресурсов. Это можно достичь, регулярно обновляя запрос в соответствии с реальной величиной. В результате эксперимента гипотеза будет проверена.

Сценарий будет состоять из множества запросов на шедулинг подов. Чтобы создать условия эксперимента, схожие с реальными, будут симулироваться процессы с псевдо-случайными сложными функциями потребления CPU. В качестве семейства таких функций были выбраны ряды Фурье с конечным числом членов и случайными коэффициентами из определённого диапазона. Этот выбор мотивирован полезными свойствами тригонометрических многочленов и лёгкостью генерации случайных функций из семейства. Ниже выписана формула, согласно которой генерируются функции. На Графике 1 показаны примеры сгенерированных функций.

$$f(x) = \sum_{n=1}^4 \left(\frac{a_n}{n+3} \sin(10^{-3}nx) + \frac{b_n}{n+3} \cos(10^{-3}nx) \right) + \sum_{n=1}^4 (|a_n| + |b_n|), \quad a_n, b_n \in [-1000, 1000]$$

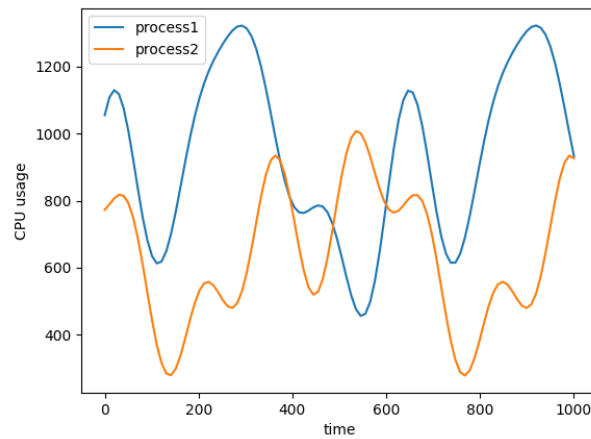


Рис. 1: Примеры сгенерированных функций

В сценарий также будут добавлены запросы на обновление запрашиваемых подами ресурсов: новая величина запроса ресурсов будет равняться среднему потреблению CPU за отрезок времени между предыдущим и текущим запросами на обновление. Чтобы регулярно через короткие промежутки времени собирать информацию о состоянии системы, также будет добавлено большое число пустых запросов: после каждого такого запроса будет сохраняться `StateSnapshot`. Всего в сценарии 10000 пустых запросов, 100 запросов на шедулинг подов и 100 запросов на обновление, все запросы одного типа вызываются через равные промежутки времени.

В качестве метрики эффективности шедулинга выбрано среднее количество времени среди нод, когда нагрузка на CPU превышает заданную границу (average CPU overload time). Среди плагинов шедулера выбраны необходимые для работы системы и **BalancedAllocation**, который отвечает за равномерное распределение подов с учётом запрошенных ресурсов. В системе будет 4 ноды с одинаковой конфигурацией.

В рамках эксперимента сценарий с обновлением запросов будет сравниваться с двумя другими, контрольными: в первом сценарии запрос CPU для каждого пода будет установлен равным среднему потреблению за короткий отрезок времени около начала эксперимента; во втором сценарии запрос будет равен среднему значению функции потребления на всём отрезке времени, симулируемого в рамках эксперимента. Второй контрольный сценарий отражает ситуацию, когда пользователь обладает полной информацией о потенциальном потреблении ресурсов процессом, первый сценарий симулирует ситуацию, когда информация неполная.

Чтобы убедиться в корректности кода, генерирующего сценарий, был проведён пробный эксперимент, использующий сценарий с обновлениями запросов. На **Графике 2** показаны графики реальной нагрузки на CPU каждой ноды. Единицы измерения нерелевантны в контексте эксперимента, поэтому опущены.

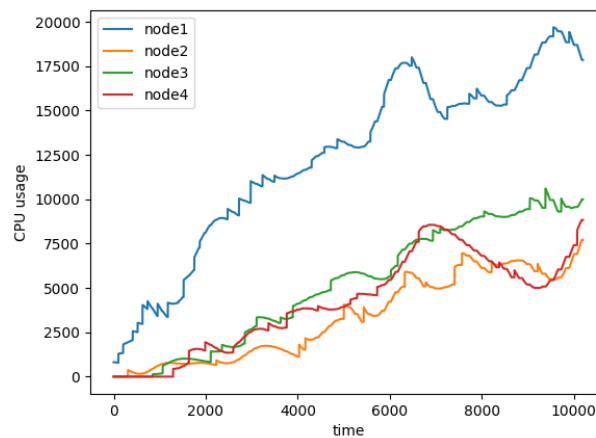


Рис. 2: Реальная нагрузка на CPU

Наконец, был проведён ряд экспериментов с использованием трёх вышеописанных сценариев. Всего было проведено 1000 экспериментов для каждого сценария, после чего на основании полученных данных построены функции плотности случайных величин, соответствующих исследуемой метрике: average CPU overload time. На **Графике 3** показаны графики полученных функций. Средние значения метрики приведены в **Таблице 1**. Код, при помощи которого можно воспроизвести данный эксперимент, доступен по ссылке^[2].

По результатам эксперимента видно, что регулярное обновление запросов ресурсов увеличило матожидание average CPU overload time, что демонстрирует негативный эффект на эффективность шедулинга. Вероятно, это связано с большой амплитудой колебаний функций и отсутствием в них трендов. Итого, можно утверждать, что (а) регулярные обновления запросов ресурсов не всегда улучшают эффективность шедулинга и (б) при поведении процессов, схожим с рассмотренным в этой секции, регулярные обновления запросов несут негативный, хоть и незначительный эффект на эффективность шедулинга с точки зрения выбранной метрики. Как итог, гипотеза была отвергнута.

Итог

Разработан инструмент для анализа эффективности шедулера kubernetes, при помощи которого можно имитировать шедулинг внутри крупного кластера, не поднимая реальный кластер. Это позволяет значительно упростить изучение эффективности алгоритма шедулинга, что было продемонстрировано в Секции 4. Его уже можно применять по назначению, но для удобного проведения более сложных исследований потребуется реализовать более обширный вспомогательный функционал для создания сложных сценариев, а также доработать поддержку плагинов, так как на данный момент не протестировано корректное взаимодействие всех стандартных плагинов шедулера с компонентами SIT.

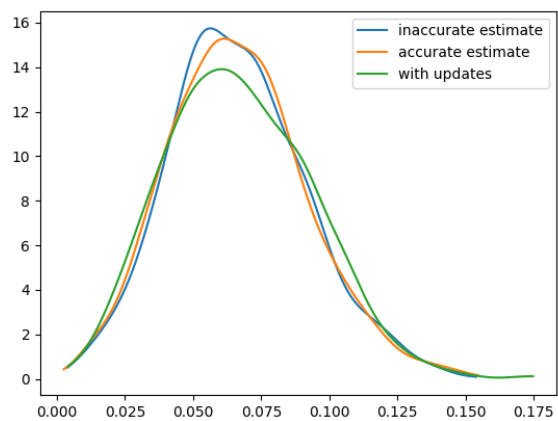


Рис. 3: Плотность с.в., соответствующих метрике

Эксперимент	Среднее значение метрики
inaccurate estimate	0.0665
accurate estimate	0.0662
with updates	0.0669

Ссылки

1. Исходный код SIT. <https://github.com/s-walrus/k8s-scheduler-test>
2. Исходный код экспериментов. <https://github.com/s-walrus/k8s-scheduler-test/tree/scratch/scheduler-eval/pkg/sit/experiments>

Приложение. Пример кода

```
func NewSelfAntiAffinityPods() *execution.StaticRequestGenerator {
    var reqs []execution.Request

    for i := 0; i < 3; i++ {
        // add node creation request
        reqs = append(reqs, requests.NewAddNode(NewTestNode(fmt.Sprintf("node%d", i+1))))
    }

    affinityPodBuilder := podbuilder.NewPodBuilder("affinity-pod")
    affinityPodBuilder.AddPreferredPodAntiAffinity(map[string]string{"affinity-group": "1"})
    affinityPodBuilder.SetLabel("affinity-group", "1")
    for i := 0; i < 16; i++ {
        // add pod scheduling request
        reqs = append(reqs, requests.NewSchedulePod(affinityPodBuilder.GetPod()))
    }

    return execution.NewStaticRequestGenerator(reqs)
}

func main() {
    plugins := []execution.PluginInfo{
        execution.NewPluginInfo(queueSort.Name, queueSort.New, "QueueSort"),
        execution.NewPluginInfo(interPodAffinity.Name, NewInterPodAffinity,
            "PreFilter", "Filter", "PreScore", "Score"),
        execution.NewPluginInfo("TrueFilter", st.NewTrueFilterPlugin, "Filter"),
        execution.NewPluginInfo(defaultBinder.Name, defaultBinder.New, "Bind"),
    }

    snapshots := execution.RunSchedulerIsolationTest(plugins, scenarios.NewSelfAntiAffinityPods())
    PrintTestResult(snapshots)
}
```