

Санкт-Петербургский государственный университет

Кафедра Системного программирования

Группа 25.M71-мм

# CPU троттлинг Go-приложений в Kubernetes

*Бикеев Кирилл Алексеевич*

Отчёт по учебной практике

в форме «Решение»

Научный руководитель:  
доцент кафедры системного программирования, Луцив Д. В.

Санкт-Петербург  
2026

## Оглавление

<b>Введение</b>	<b>3</b>
<b>1. Постановка задачи</b>	<b>4</b>
<b>2. Обзор</b>	<b>5</b>
2.1. Модель ресурсов Kubernetes: Requests и Limits . . . . .	5
2.2. Троттлинг Go-приложений . . . . .	7
2.3. Выводы . . . . .	10
<b>Заключение</b>	<b>12</b>
<b>Список литературы</b>	<b>13</b>

# Введение

В современной экосистеме облачных вычислений язык программирования **Go** стал стандартом де-факто для разработки микросервисов благодаря эффективной модели конкурентности и простоте деплоя в контейнеризированных средах. Однако запуск Go-приложений в среде **Kubernetes** сопряжен с серьезными вызовами в области управления ресурсами CPU. Основным механизмом, вызывающим деградацию производительности, является **процессорное дросселирование (throttling)**, которое выступает «тупым» инструментом ограничения потребления ресурсов процессора ядром Linux [3].

Kubernetes использует контрольные группы (*cgroups*) для управления ресурсами, транслируя параметры `limits.cpu` в квоты планировщика **CFS (Completely Fair Scheduler)** [4]. Троттлинг активируется, когда контейнер исчерпывает выделенную ему квоту времени (например, 25 мс) в рамках фиксированного периода (обычно 100 мс). Проблема Go-приложений заключается в том, что до версии 1.24 включительно рантайм Go устанавливал переменную `GOMAXPROCS` (определяющую параллелизм на уровне ОС) равной общему количеству логических ядер на хосте, а не лимиту в контейнере [3].

В результате Go-приложение на 64-ядерном узле с лимитом в 1 ядро может запустить одновременно 64 потока, которые израсходуют всю 100-миллисекундную квоту всего за 1.56 мс реального времени, после чего выполнение всего приложения будет полностью приостановлено на оставшиеся 98.44 мс. Представьте, что у вас есть бригада из 64 рабочих (потоков), но у вас есть деньги (квота CPU) только на 1 час работы одного человека в день. Если вы позволите всей бригаде выйти на объект одновременно, они отработают меньше одной минуты и остаток дня будут сидеть без дела, потому что бюджет исчерпан.

.

# **1. Постановка задачи**

Целью данной работы является изучение феномена троттлинга в Kubernetes и поиск челленджей в данной области. Поставлены следующие задачи

1. Найти материалы, поясняющие что такое троттлинг и как он влияет на производительность Go-приложений;
2. Проанализировать существующие подходы к решению проблемы троттлинга;
3. Понять их слабые места и сформировать вектор для дальнейшей работы;

## 2. Обзор

Чтобы в полной мере понять механизм троттлинга CPU, необходимо рассмотреть два уровня: модель управления ресурсами, определённую в Kubernetes, и низкоуровневый механизм их принудительного ограничения, реализованный в ядре Linux. Именно взаимодействие этих двух уровней порождает наблюдаемые эффекты, которые пагубно влияют на производительность приложений.

### 2.1. Модель ресурсов Kubernetes: Requests и Limits

В Kubernetes управление ресурсами CPU для каждого контейнера в поде осуществляется с помощью двух ключевых параметров: `requests` и `limits`.

- **CPU Requests** (`resources.requests.cpu`) — это гарантированный минимум процессорного времени, который будет зарезервирован для контейнера. Планировщик Kubernetes (`kube-scheduler`) использует это значение для принятия решения о размещении пода, гарантируя, что суммарные `requests` всех подов не превышают доступные ресурсы узла.
- **CPU Limits** (`resources.limits.cpu`) — это жёсткий верхний предел потребления CPU. Если контейнер попытается использовать больше процессорного времени, чем указано в `limits`, он будет принудительно ограничен (подвергнут троттлингу).

Kubernetes транслирует эти параметры в настройки механизма control groups (cgroups) в ядре Linux. CPU Requests преобразуются в относительный весовой коэффициент (`cpu.shares` в cgroup v1 или `cpu.weight` в cgroup v2), который определяет долю CPU, выделяемую контейнеру при конкуренции за ресурсы. В то же время CPU Limits транслируются в абсолютные значения квоты (`cpu.cfs_quota_us` и `cpu.cfs_period_us` в cgroup v1 или `cpu.max` в cgroup v2) [4].

Соблюдение CPU Limits обеспечивается планировщиком CFS (Completely Fair Scheduler) ядра Linux. Этот механизм работает на

основе циклов, длительность которых составляет 100 миллисекунд. Этот период не является произвольным, а представляет собой жёстко заданную константу как в ядре Linux, так и в кодовой базе Kubernetes (`QuotaPeriod = 100000` микросекунд) [4, ?]. Параметр `cfs_quota_us` определяет, сколько микросекунд процессорного времени контейнер имеет право использовать в течение каждого 100-миллисекундного периода. Например, лимит в `1000m` (1 ядро) означает, что контейнер может потреблять 100 000 микросекунд (100 мс) процессорного времени в каждом 100-миллисекундном периоде. Ключевой аспект этого механизма заключается в его бескомпромиссности: как только контейнер исчерпывает свою квоту, его выполнение **принудительно приостанавливается до начала следующего периода**. Это происходит даже в том случае, если на узле есть свободные процессорные ядра, которые простоявают в данный момент [1]. Таким образом, механизм лимитов, хоть и обеспечивает изоляцию ресурсов, является грубым инструментом, который приводит к серьёзным и не всегда очевидным последствиям для производительности приложений. Троттлинг не просто замедляет приложения; он вносит нелинейную и коварную форму деградации производительности, которая особенно губительна для распределенных систем, чувствительных к задержкам. Когда контейнер подвергается троттлингу, обработка текущих запросов приостанавливается. Это немедленно оказывается на времени ответа, но также вызывает каскадный эффект. Новые входящие запросы начинают накапливаться в очередях, ожидая, пока приложение снова получит доступ к CPU. Исследования показывают, что одно лишь наличие CPU `Limits`, даже при среднем потреблении CPU, не превышающем установленный лимит, способно увеличить **хвостовую задержку (tail latency)** до 5 раз [1]. Это происходит из-за того, что троттлинг не сглаживает кратковременные всплески нагрузки (микро-всплески), а жёстко их прерывает. Такие внезапные задержки приводят к каскадным сбоям. Например, рост очередей запросов в памяти приложения может привести к превышению лимита по памяти и последующему завершению процесса сигналом OOMKilled (Out of Memory) [1].

## 2.2. Троттлинг Go-приложений

Приложения, написанные на языке Go, являются ярким примером того, как архитектура рантайма вступает в прямой конфликт с механизмом троттлинга в Kubernetes. Рантайм Go использует переменную окружения `GOMAXPROCS` для определения максимального количества потоков операционной системы, которые могут одновременно выполнять код Go. До версии Go 1.25 по умолчанию значение этой переменной устанавливалось равным количеству процессорных ядер на *физическом узле*, а не в рамках квоты, выделенной контейнеру [2]. Это несоответствие создает катастрофические последствия. Рассмотрим сценарий:

1. Приложение на Go запущено на узле с 8 ядрами, и `GOMAXPROCS` по умолчанию равно 8.
2. Контейнеру установлен CPU Limit в `1000m` (1 ядро), что даёт ему квоту в 100 мс процессорного времени на каждые 100 мс реального времени.
3. Многопоточный рантайм Go, используя все 8 потоков, исчерпывает свою квоту в 8 раз быстрее — всего за 12.5 мс реального времени ( $12.5 \text{ мс} \times 8 \text{ потоков} = 100 \text{ мс CPU времени}$ ).
4. После этого все 8 потоков приложения **блокируются на оставшиеся 87.5 мс**, дожидаясь начала нового периода CFS [3].

Этот сценарий идеально иллюстрирует грубость механизма квот CFS: ядро, не имея представления о внутреннем планировщике рантайма Go, наказывает все приложение за то, что оно ведет себя в точности так, как и было спроектировано для многоядерного узла, эффективно превращая возможность параллелизма в источник проблем с производительностью. В результате приложение простояивает большую часть времени, что приводит к резкому падению производительности. Эксперименты показывают, что при неверной настройке `GOMAXPROCS` пропускная способность веб-сервиса падает с **94 до 26 запросов в секунду** [3]. Кроме

того, такой режим работы негативно сказывается на внутренних процессах рантайма, таких как сборка мусора (GC) [4].

Хотя троттлинг является предсказуемым следствием механизма лимитов, его обнаружение и диагностика в работающей системе требуют целенаправленного мониторинга и понимания типичных сценариев, которые к нему приводят. К троттлингу CPU чаще всего приводят следующие причины:

- **Недооцененные CPU Limits:** Начальная конфигурация приложения не учитывает пиковые или возросшие нагрузки.
- **Микро-всплески нагрузки:** Кратковременные, но интенсивные всплески, которые легко исчерпывают квоту задолго до окончания 100-миллисекундного периода.
- **Несоответствие в многопоточных приложениях:** Классический пример с переменной `GOMAXPROCS` в Go или аналогичными настройками в других средах исполнения (например, JVM).
- **Фоновые процессы с высоким потреблением CPU:** Интенсивные операции, такие как сборка мусора, JIT-компиляция или другие фоновые задачи.

Для своевременного обнаружения троттлинга необходимо использовать систему мониторинга, такую как Prometheus. Ключевые метрики для этого предоставляет cAdvisor — компонент, встроенный в Kubelet на каждом узле Kubernetes. Основным показателем является `container_cpu_cfs_throttled_seconds_total`, который напрямую измеряет суммарное время, в течение которого контейнер был подвергнут троттлингу.

Решение проблемы троттлинга CPU требует комплексного подхода — от простой коррекции конфигурации конкретных приложений до фундаментального пересмотра общей стратегии управления ресурсами в кластере. Для многопоточных приложений, таких как сервисы на Go, критически важно согласовать настройки рантайма с выделенными ре-

сурсами контейнера. Для решения проблемы с `GOMAXPROCS` существуют три основных подхода:

1. **Ручная настройка через `resourceFieldRef`:** Это достигается с помощью Kubernetes Downward API через `resourceFieldRef`, который во время выполнения внедряет значение `limits.cpu` из определения ресурсов самого контейнера непосредственно в переменную окружения `GOMAXPROCS` [2].
2. **Использование специализированных библиотек:** Библиотека, такая как `go.uber.org/automaxprocs`, при импорте в приложение автоматически определяет лимиты `cgroups` и настраивает `GOMAXPROCS` во время выполнения. Это более надежный подход, который работает даже при динамическом изменении лимитов [6].
3. **Использование современных версий Go:** Начиная с версии Go 1.25, рантайм Go стал "container-aware". Он автоматически определяет лимиты CPU, установленные для контейнера, и соответствующим образом настраивает `GOMAXPROCS`, что устраняет необходимость в ручной настройке или сторонних библиотеках [5].

Радикальный, но набирающий популярность в индустрии подход — **полный отказ от установки CPU Limits** для приложений, чувствительных к задержкам [1]. Аргументация этого подхода следующая: планировщик Kubernetes гарантирует, что сумма `requests` на узле никогда не превышает его емкость. Это, в сочетании с обещанием планировщика CFS о пропорциональной справедливости на основе `cpu.shares`, уже предотвращает проблему "шумного соседа", обеспечивая каждому контейнеру его гарантированную долю CPU во время конкуренции. Таким образом, CPU `Limits` являются избыточным и вредным уровнем контроля для нагрузок, чувствительных к задержкам. В отсутствие `limits`, приложение может свободно использовать незанятые ресурсы узла во время всплесков нагрузки, что значительно улучшает производительность. При таком подходе фокус управления ресурсами смешается с предотвращения превышения жёстких лимитов на грамотное планиро-

вание ёмкости на основе `requests`. Vertical Pod Autoscaler (VPA) — это инструмент Kubernetes, который автоматизирует процесс настройки `requests` и `limits`. Он анализирует историческое потребление ресурсов подами и предлагает оптимальные значения, помогая найти правильный баланс между экономией ресурсов и производительностью [7]. VPA может работать в режиме рекомендаций (`updateMode: "Off"`), в котором он не применяет изменения автоматически, а лишь отображает их в статусе объекта `VerticalPodAutoscaler`. Это позволяет инженерам безопасно получать предложения по оптимизации ресурсов, не рискуя автоматическими перезапусками подов, и принимать информированные решения. Выбор конкретной стратегии или их комбинации зависит от критичности приложения, его архитектуры и зрелости DevOps-процессов в организации.

## 2.3. Выводы

При неправильном использовании, особенно при слепом следовании устаревшим ”лучшим практикам”, троттлинг наносит значительный и трудно диагностируемый вред производительности микросервисных приложений, чувствительных к задержкам. Ключевой вывод заключается в том, что для многопоточных систем, таких как приложения на Go, установка жестких CPU `Limits` контрпродуктивна. Конфликт между логикой рантайма и грубым механизмом квотирования CFS приводит к парадоксальной ситуации, когда приложение простаивает, имея доступные вычислительные ресурсы. Современная стратегия управления ресурсами требует фундаментального сдвига в философии: от ”**ограничения**” (`limits`) к ”**гарантиям**” (`requests`). Это не просто рекомендация, а необходимая эволюция, требуемая для раскрытия полного потенциала современных контейнеризированных приложений. Это подразумевает использование рантаймов, адаптированных к работе в контейнерах, и тщательный мониторинг метрик троттлинга. Для наиболее критичных к задержкам сервисов следует серьезно рассмотреть полный отказ от CPU `Limits`, полагаясь на CPU `Requests`.

для обеспечения предсказуемости и позволяя приложениям эффективно справляться с пиковыми нагрузками.

# Заключение

Были изучены материалы по теме троттлинга в Kubernetes и как он влияет на Go-приложения. Существуют несколько основных подходов к решению проблемы, такие как автоскейлинг подов и настройка `GOMAXPROCS`. Однако на данный момент замечен нарастающий тренд на отказ от использования `CPU limits` в пользу `CPU requests`. Однако Go 1.25 по умолчанию не ориентируется на `requests`, поэтому в будущем стоит развивать работу именно в направлении разработки такого инструмента, который позволит оптимизировать рантайм Go под `CPU requests`.

# Список литературы

- [1] Chirag Shetty UIUC Illinois USA Sarthak Chakraborty UIUC Illinois USA Hubertus Franke IBM Research New York USA Larisa Shwartz IBM Research New York USA Chandra Narayanaswami IBM Research New York USA Indranil Gupta UIUC Illinois USA Saurabh Jha IBM Research New York USA. CPU-Limits kill Performance: Time to rethink Resource Control. — URL: <https://arxiv.org/pdf/2510.10747v1> (дата обращения: 2026-01-03).
- [2] GOMAXPROCS and GOMEMLIMIT in Kubernetes. — URL: <https://blogHowardJohn.info/posts/gomaxprocs/> (дата обращения: 2026-01-03).
- [3] Kennedy William. Kubernetes CPU Limits and Go. — URL: <https://www.ardanlabs.com/blog/2024/02/kubernetes-cpu-limits-go.html> (дата обращения: 2026-01-03).
- [4] Le Phuong. Container CPU Requests Limits Explained with GOMAXPROCS Tuning. — URL: <https://victoriameetrics.com/blog/kubernetes-cpu-go-gomaxprocs/> (дата обращения: 2026-01-03).
- [5] Michael Pratt Carlos Amedee. Container-aware GOMAXPROCS. — URL: <https://go.dev/blog/container-aware-gomaxprocs> (дата обращения: 2026-01-03).
- [6] Uber’s Automaxprox doc. — URL: <https://github.com/uber-go/automaxprocs/blob/master/README.md> (дата обращения: 2026-01-03).
- [7] Vertical Pod Autoscaling: The Definitive Guide. — URL: <https://povilasv.me/vertical-pod-autoscaling-the-definitive-guide/> (дата обращения: 2026-01-03).