Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Odin Ugedal

Bandwidth Control And Fairness In The Linux Scheduler

Characterizing And Improving Performance Characteristics And Fairness In The Linux CFS Scheduler And Its Bandwidth Control System

Master's thesis in Computer Science
Supervisor: Rakesh Kumar
June 2021

NTNU
Norwegian University of Science and Technology

Odin Ugedal

Bandwidth Control And Fairness In The Linux Scheduler

Characterizing And Improving Performance Characteristics And Fairness In The Linux CFS Scheduler And Its Bandwidth Control System

Master's thesis in Computer Science
Supervisor: Rakesh Kumar
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

As more and more of the data center industry focus on shared infrastructure, the need for proper resource control between different applications with different priorities is increasing. After the introduction of Kubernetes in 2014, it has turned the table upside down when it comes to how infrastructure is managed. It has allowed system administrators to run multiple applications with different priorities on the same physical machines in a safe and fault-tolerant way. All of this requires a set of complex resource management APIs in the operating system, and this is really where the Linux kernel shines. Linux containers, or control groups, allow for this with excellent performance and flexibility. To manage the usage of CPU time between groups of processes, Linux has a CFS bandwidth controller, used by all systems managing containers, including Kubernetes. Unfortunately, it has a set of configuration pitfalls, potentially causing performance issues. Containers reaching their limit will be punished by being paused for an amount of time, also known as throttled. This throttling is often worse and more aggressive than it is supposed to be.

Our research dives deep into the Linux CFS scheduler and its bandwidth controller. We investigate the reasons for the unnecessary throttling and propose multiple techniques to avoid it. Our results show that our proposed techniques mitigate the throttling, resulting in a performance increase of about **3%** on a busy system. We also found that our proposed techniques reduce the overhead of CFS bandwidth control by about **95%**.

Incidentally, we also discovered a set of issues related to fairness in the Linux scheduler. These issues can stall programs on CPU congested systems in both theory and practice, slowing them down by orders of magnitude. Our testing shows that we were able to generate a real-world situation where a program took approximately **170** times longer to execute than expected. Our findings have been reported to the Linux community, together with our techniques for mitigating them. Some of our proposed techniques, implemented as kernel patches, have already been merged into the mainline kernel, while others are still in flight.

Sammendrag

Samtidig som mer og mer av datasenterindustrien fokuserer på delt infrastruktur, er nødvendigheten av skikkelig ressursstyring og prioritering av forskjellige programmer med forskjellige prioriteringer blitt enda viktigere. Etter introduksjonen av Kubernetes i 2014, har det snudd måten man håndterer infrastruktur på, på hodet. Kubernetes lar systemadministratorer kjøre forskjellige programmer med forskjellig prioritet, på den samme fysiske maskinvaren, på en sikker og feiltolerant måte. Alt dette krever mye av operativsystemet når det kommer til metoder for å oppnå slik ressursstyring. Det er her Linux virkelig skinner. Konteinere, eller kontrollgrupper, gjør dette til en lek. For å passe på bruken av CPU-tid, har Linux en CFS båndbreddekontroller som brukes av alle konteinersystemer, inkludert Kubernetes. Denne funksjonaliteten har dog et sett med fallgruver som potensielt kan gå utover ytelsen. Konteinere som bruker mer enn sin tilmålte tid blir straffet ved at de settes på pause, også kalt struping. Denne strupingen er ofte verre og mer aggressiv enn den skal være.

I vår forskning går vi i dybden på hvordan Linux sin prosess-planlegger og dens båndbreddekontroller fungerer. Vi ser på måter å minske strupingen av konteinere, og ser på forskjellige teknikker for å gjøre det. Resultatene våre viser at teknikkene gir opp til **3%** bedre ytelse på ellers travle maskiner. Vi ser også at teknikkene våre reduserer kostnaden av å bruke båndbreddekontrolleren med opp til **95%**.

Samtidig har vi også funnet en rekke problemer relatert til rettferdighet mellom programmer in Linux. Disse problemene kan i verste fall stoppe programmer helt, og gjøre dem treigere vesentlig treigere. Testing vår viser at problemene kunne gjøre at programmer bruker mer enn **170** ganger så mye tid som de skal. Vi har rapportert disse problemene til postlistene til Linux, sammen med våre løsninger. Noen av løsningene våre er allerede blitt en del av Linux, mens andre fortsatt diskuteres.

Preface

This is my master thesis written between Jan 2021 and June 2021, as the final deliverable of my degree in Computer Science at Norwegian University of Science and Technology. The research started with an interest in Linux and its scheduler, and on how Linux containers *really* work. So, I decided to dedicate my last semester to it, and write my thesis about the result.

I used the first few months to get familiar with the kernel in general, and the file `kernel/sched/fair.c`, where most of the CFS scheduler logic is located. The CFS bandwidth patches gradually saw life in February-March, while iterating through my ideas. Those presented here are some of my ideas, while others didn't make it.

Sometime early in April, I discovered that all my results were pretty skewed without any reason, with vast amounts of variance, forcing me to debug *why*. After two weeks with more or less no sleep and with much more knowledge about the scheduler later, I discovered the fairness issues presented in the thesis. Although they were not intended to be part of the thesis initially, they became a pretty important part when I started to write. Thankfully they are now fixed in the Linux kernel itself (or, *some* of them), so none of you have to deal with the same!

Acknowledgments

I would like to thank my supervisor Rakesh Kumar for all his help and feedback during the course of this thesis. I would also like to thank the Kubernetes community for making me interested in the topic, and all the contributors to Kubernetes and the Linux kernel.

I would also like to give a special thanks to all the people who have been around on F252 for the last five years, and for making my time at NTNU so memorable. All those late evenings, late nights, and of course all the too early mornings before six. You know how you are, and you will all be missed. Thanks!

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

Over the last decade, software and hardware development has kept growing, and the data center market has become a more and more critical part of the game. More and more focus has been shifted away from having huge and expensive servers in-house, and towards renting a fleet of smaller autonomous servers from cloud providers, shared between different applications.

Running multiple applications with different priorities, latency requirements, and uptime requirements, introduces a new set of challenges in enforcing resource limits and ensuring fairness during spikes in load. This requires multiple levels of quality of service and sandboxing techniques that allow developers to run applications without full access to all the resources on the system. It is also important for avoiding noice for others, and ensuring security when multiple customers are running on the same physical hardware. These requirements necessitate a vast set of kernel-level tools for enforcement, together userspace tools for managing them. In 2015, Google disclosed information about their internal cluster management tool called Borg [1]. Borg later acted as an inspiration to the open-source cluster management tool Kubernetes, [2], which has become the state of the art tool for managing containers, supported by all major cloud providers.

All these high-level userspace tools rely on kernel-level sandboxing techniques with the corresponding APIs for exposing them to the userspace tools. Linux[1] has a set of virtualization APIs for running virtual machines. It also has control group controllers for enforcing isolation, prioritization, limits, and accounting of resources. These controllers are the building blocks of all cluster management tools for Linux that do not rely on full virtualization. The ergonomics, usability, and performance of these tools are therefore essential to the efficient operation of modern data centers. Together, they make it possible to overcommit resources efficiently and transparently, making it easier to maximize resource utilization

[1] Linux is a Registered Trademark of Linus Torvalds.

on a cluster-wide scale. Like the overcommitment of memory possible when using virtual memory, this type of resource distribution allows for greater flexibility and resource utilization.

This is especially important when it comes to sharing CPU resources between a set of programs to maximize CPU usage, while still ensuring relative fairness between groups. This allows for multiple programs with multiple priority classes to run together on the same physical hardware, without sacrificing on the performance characteristics of the programs with the highest priorities. This is also important to allow low-priority tasks to run on otherwise idle hardware while still ensuring the interference with other tasks is limited.

In this thesis, we take a deep dive into the functionality of Linux kernel that makes it possible to hard limit the max CPU usage for a set of processes, also known as CFS bandwidth control. We look at its performance characteristics, how it behaves in various situations, and if any aspects of the bandwidth control mechanism can be improved. We also look at how the Linux scheduler tries to achieve fairness between processes, and how we can further improve it.

The current CFS bandwidth control implementation does have a lot of pitfalls when it comes to configuration. Still, it also has some fundamental issues when it comes to how it enforces the limit. Processes running close to, but under their limit, often see themselves getting punished as if they went over their limit, even though that is not the case. When the CFS Bandwidth Controller discovers that a process has reached its limit, the process is paused until a new CFS Bandwidth period starts. This process is called throttling. Such throttling cause problems in many ways. Kubernetes integrates thightly with the CFS bandwidth controller, and gives easy access to the CFS bandwidth metrics. Due to this, users have reported a set of complaints about CFS bandwidth throttling when using Kubernetes [3, 4]. Furthermore, to account for the CPU usage, the CFS Bandwidth controller also has to communicate between physical CPU cores. As this accounting often happens tens of thousands of times every second, the accounting overhead can also become a performance bottleneck.

1.1 Research Goals

Based on our motivation, we define the following research goals:

> **G1: Improve CFS bandwidth controller by avoiding as much unnecessary throttling as possible.**

The current CFS bandwidth control implementation is difficult to understand properly, and has some significant configuration pitfalls that are not well studied and documented. Some areas might benefit from improvements to help avoid unnecessary throttling, with the corresponding overhead that throttling cause for the overall system. The throttling itself also hurts performance quite substan-

tially, especially when it comes to user-facing tasks. Together with **G1**, we set a more significant overall goal:

> **G2: Reduce the overhead of CFS bandwidth control to enable more precise distribution of runtime and lowering the performance penalty.**

During the last decade, the average CPU core count per physical chip has increased steadily. Since CFS bandwidth control relies on cross-core communication to ensure the CPU usage is below the quota, it is becoming more and more essential to keep the overhead of this communication as small as possible. Lastly, we define a broader goal to understand how all this affect users:

> **G3: Find evidence about how CFS bandwidth throttling affects scheduler fairness.**

1.2 Contributions

The main deliverables of this research are the research data and its findings, together with our proposed changes to the Linux Kernel. Together, they answer all three of our research goals. We have made the following key contributions;

- **Throttling Patches**: The patches for solving unnecessary CFS bandwidth throttling problems have shown good results. Together with the data from this research, we will submit them to the Linux kernel mailing lists for feedback. The patches can be found in Appendix A.1.
- **CFS Bandwidth Overhead Patch**: The patch for reducing the overhead of CFS bandwidth control has also shown good results. Together with the data from this research, we will submit it to the Linux kernel mailing lists for feedback. The patch can be found in Appendix A.2
- **Fairness Patches**: Our most important contribution is the discovery of fairness issues in the Linux CFS scheduler, each causing excessive fairness problems. All issues have been reported to the Linux mailing lists, together with the corresponding patches to solve them. The patches can be found in Appendix A.3. Some of the patches have already been merged into the mainline Kernel, together with other patches based on our discoveries, and will be part of the 5.13 release. The merged patches have also been backported to the 5.12, 5.10, 5.11, 5.4, and 4.19 stable versions of the Linux kernel, essentially meaning that most modern Linux distributions are running fixed kernels. Others are still in review, together with other fixes for issues found becuase of our research.

The research also gives insight into how CFS bandwidth interferes with the fairness in the schedueIr. It also shows the overhead of CFS bandwidth control.

1.3 Structure

We structure our thesis into three parts:

Background

The first two chapters describe background information and give necessary insights into relevant topics.

- **Chapter 2** introduces the Linux kernel and gives an insight into how it works under the hood. It mainly focuses on the Linux CFS scheduler and its bandwidth controller.
- **Chapter 3** goes into more depth about the Linux CFS scheduler and presents its caveats and challenges.

Design Improvements And Implementation

The second part is where we present our ideas to mitigate the issues discussed in the background part. We also look at how our ideas work out and how they affect different workloads. This part is divided into three distinct chapters, each presenting its goals, in-depth analysis, implementation, result, and discussion.

- **Chapter 4** demonstrates how our ideas to solve unnecessary throttling, as presented in **G1**, works. It also looks at how throttling affects fairness, as explained in **G3**.
- **Chapter 5** demonstrates how our ideas to reduce the CFS bandwidth overhead, as explained in **G2**, works.
- **Chapter 6** presents our findings and mitigations of fairness-related issues in the Linux CFS scheduler, as presented in **G3**.

Discussion And Conclusion

The last part of the thesis discusses our overall results and then concludes with the findings of this research.

- **Chapter 7** discusses our overall results and how they affect the Linux kernel. It also describes the limitations of our research and possible critics.
- **Chapter 8** concludes this research.

Part I

Background

Chapter 2

The Linux Kernel

Over the last three decades, The Linux Kernel has evolved from a small university hobby project, into one of the world's most influential and well-functioning software projects. It runs on all kinds of hardware, ranging from small energy-efficient smartphones to the world's biggest data centers. All of the world's top 500 supercomputers are also running Linux [5]. All this while still being a fully open and community-driven project where everyone can contribute.

In this chapter, we will give an overview of the Linux Kernel, with a primary focus on the Linux Scheduler. We provide an in-depth review of how the scheduler works, how it has evolved over the last two decades, and how it connects to other central parts of the kernel. We also focus on what the scheduler tries to achieve when managing a set of processes and what it does to archive those goals.

2.1 Development model

In contrast to most modern software projects, the Linux kernel did not start as a project inside a big corporation. Instead, it was created as a free and open-source hobby project by a Finnish graduate student called Linus Torvalds [6]. The new project was released under the GPLv2 License, which has later turned out to be one of its great success factors. The original author, Linus Torvalds, has subsequently stated that he thinks the license choice has been one of the defining factors in the success of Linux [7].

Now, more than 30 years after the initial release, Linux is still under heavy development. Although both hardware and software have evolved tremendously, the development model of Linux is mostly the same today as it was 20 years ago. Everyone can submit new proposed patches via email to public mailing lists, where everyone can comment and suggest changes. Certain people have the informal responsibility of a given subsystem of the Kernel, and they gather and accept proposed patches to their respective subsystem. Then, at the beginning of a new release cycle, those subsystem maintainers submit their accumulated set of changes to the mainline tree, the main source tree controlled by Linus Torvalds himself.

Linux comes in many flavors and distributions, where the distributions have taken different choices for different system requirements. This means that Linux can be packaged in many ways, depending on the use case. Linux supports multiple hardware architectures and use-cases, unlike most other operating systems. The vast majority of functionality can be turned on or off at compile time, making it fairly flexible. This makes it very versatile for everything between and including advanced servers and low-powered battery-driven sensors. The common denominator is the Linux kernel itself, and its core components.

2.2 The hierarchy of processes

Like all moderns operating systems, Linux is centered around handling and executing processes and threads[1]. The first process Linux starts when the system is booting up is known as the init process, and it always has process id 1. It is therefore often referred to as PID 1. By default, when Linux starts up, it executes the file `/sbin/init` as the init system. It is the responsibility of the system administrator to copy or create a symbolic link to the correct program on that path. The init system is responsible for bootstrapping the necessary kernel functions and starting other system and background processes to make the system ready to use.

When a new process is started, it inherits the capabilities of the process starting

[1] Even though processes and threads are different things, in Linux scheduling terms, they are similar; schedulable entities, often just called processes for simplicity

it. We call the process starting a new process for the parent process of the newly created process. This makes a tree of processes, where the PID 1 is the root. An example where two users, *Alice* and *Bob* each run a shell session with the GNU Bourne-Again SHell (bash), each editing a file, and *Bob* compiling using the GNU project C compiler (GCC), can be found in Section 2.2.

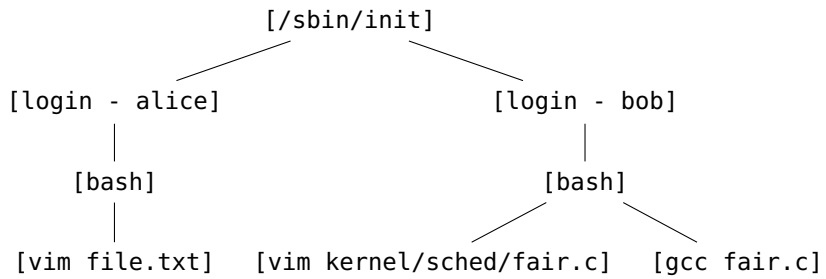


Figure 2.1: Process hierarchy with two users

2.2.1 Control groups

The process hierarchy can help find the connections between processes and where they originate. The transient nature of this grouping makes it challenging to utilize for both accounting and resource distribution. Processes often spawn new processes and threads, and later kills them. Processes in different parts of the process hierarchy might communicate and work together from a high-level overview, forming a logical unit together. New processes can also be started elsewhere in the hierarchy, making it harder for applications to know the entire state without manually tracking every process. This is also a complicated problem since some processes are short-lived while others are long-lived. To overcome these grouping problems, Linux has a separate mechanism called Control groups[2], for organizing processes hierarchically [8]. Control groups, often referred to as *cgroups*, are implemented as two main components;

- **cgroup core**: The core is responsible for the internal infrastructure for grouping processes hierarchically.
- **cgroup controllers**: A control group controller is responsible for enforcing a certain policy and/or distribution of a given resource, configurable by the user.

An example of a control group hierarchy can be found in Figure 2.2.

The control group hierarchy is controlled from userspace via a file system mount, often mounted on `/sys/fs/cgroup`. Each directory in that mount point corresponds to a control group, and each file represents some data that could either

[2]Linux has two versions of control groups; Legacy/v1 and Unified/v2. When we refer to control groups, we refer to the Unified version.

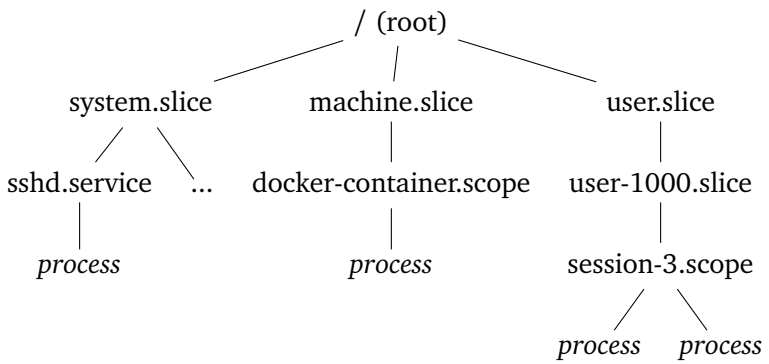


Figure 2.2: Example cgroup hierarchy on a system using systemd as init system

be read or written, depending on its function. When moving a process to a control group, the administrator can write the process id into the file `cgroup.procs` of the control group. This also means that creating new control groups is as simple as creating new directories. System administrators can delegate permissions for creating and modifying control groups via file system permissions.

Over the last five years, the popularity of Linux containers has increased dramatically, and control groups are among the essential tools used to implement them. A container can be viewed as a separate control group from a process, scheduler, and resource isolation perspective. Linux containers also rely heavily on the namespace APIs. Containers are managed by container runtimes that communicate with the control group APIs when creating, managing, and destroying containers.

Resource Distribution models

Depending on the resources a control group controller is made for, it has a certain way of distributing resources. The general convention is that resources are distributed in a top-down manner from the root control group. This means that the effective value of a given resource that a process can use is limited by the ancestor with the strictest policy and/or limit. The four ways of resource distributions are;

- **Weights**: The resources available at the parent control group is distributed in a manner where each children control group gets a fraction of the resources approximately to the ratio between its weight and the sum of all the weights of its siblings, including itself.
- **Limits**: Sets the maximum amount of a given resource a control the group can use, untied to the actual availability of the resource. Allows for overcommitment.
- **Protections**: Protects a given resource from usage by all processes in its control group children.

- **Allocations**: Allows access to a given amount of resources there exists a finite amount of, and cannot be overcommitted.

A control group controller can use either one or multiple of these resources distribution models, all depending on what resource it controls.

2.3 The Linux Scheduler

The central piece of the operating system that decides what process[3] to execute is called a process scheduler. As seen in Table 2.1, Linux supports six different scheduling policies, each with a different way of prioritizing what to execute in what order. The default scheduling policy assigned to newly created processes, `SCHED_NORMAL`, is the policy used for all normal user tasks. It is the implementation of this scheduling class (and its equivalent on other systems) that is often meant when people are referring to the *"Scheduler"* of a given system.

Table 2.1: The different scheduling policies supported by Linux

Policy Name	Class	Description
SCHED_NORMAL[4]	fair	Normal time sharing scheduling
SCHED_FIFO	rt	First in-first out scheduling
SCHED_RR	rt	Round-robin scheduling
SCHED_BATCH	fair	Like SCHED_NORMAL, with less preemption
SCHED_IDLE	idle	Low priority tasks
SCHED_DEADLINE	dl	Deadline scheduling

The six different scheduling policies can be divided into four internal scheduler classes, as seen in Table 2.1. Each scheduling policy maps to a scheduling class that prioritizes between processes assigned to that scheduling class. All scheduling classes implement the same interface that the scheduler's core can call to make decisions. The scheduling classes are ordered by priority. When the core scheduler looks for the next process to execute, it will iterate through the scheduling classes and start executing the first process returned from a scheduling class.

The scheduler will also notice when a sleeping process wakes up from sleep and is marked ready for execution. Depending on its priority compared to the currently running process, it might either enqueue it or preempt the current process to execute the newly awoken process. The same applies when new processes are started, processes are terminated, interrupts need handling, and a process is blocked and goes to sleep.

[3] Due to the slight difference between processes and threads, we don't distinguish between them

[4] Named `SCHED_OTHER` in user space.

Switching between processes involves a set of operations to make sure it happens transparently for the processes themselves, resulting in a measurable overhead.

The Linux scheduling classes ordered with descending priority: `dl`, `rt`, `fair` and `idle`.

2.4 The Completely Fair (CFS) Process Scheduler

The Completely Fair (CFS) Scheduler is the default scheduler in Linux[5], and its internal red-black tree [9] and support for weighted group scheduling have proved to be reliable, efficient, and predictable. As with all other schedulers, CFS has to compromise the system's fairness, latency, and throughput. Just as the name suggests, CFS tries to archive complete fairness between processes while still making it archive high throughput.

2.4.1 The Internal Red-Black Tree

The main difference from other process schedulers is that CFS utilizes a time-ordered red-black tree for storing runnable processes[6]. The index value of a scheduling entity is called the virtual runtime of the entity. This is in significant contrast to other schedulers that often use runqueues, arrays with all the processes and their respective metadata. When using a runqueue, each process gets to run for a certain amount of time, often called a time slice, before going to the next process. In great contrast, the red-black tree in the CFS scheduler acts as a timeline for the order in which processes will run, as seen in Figure 2.3. When the scheduler admits a process, it is allowed to run for as long as it is the left-most element in the red-black tree. The minimum scheduling granularity, the minimum time a task will run for before being replaced by the new process with a smaller virtual runtime, can be configured based on the compromise between immediate fairness and throughput.

2.4.2 Weighted Group Scheduling

The other significant advantage of the CFS scheduler is its support for weighted group scheduling. Group scheduling makes the scheduler handle groups of processes in a hierarchy, together with a weighting system for prioritizing between groups. This is archived by utilizing the control group functionality as seen in Section 2.2.1 and exposing the *cpu* controller. The controller allows system administrators to set a weight[7] per control group, as seen in Section 2.2.1. The scheduler, therefore, has a separate red-black tree per control group, allowing

[5] The processes with `SCHED_NORMAL` and `SCHED_BATCH`

[6] There is a separate Red-Black tree per scheduling unit / logical CPU.

[7] In Legacy/v1 control groups, the term *shares* are used, but the functionality is the same

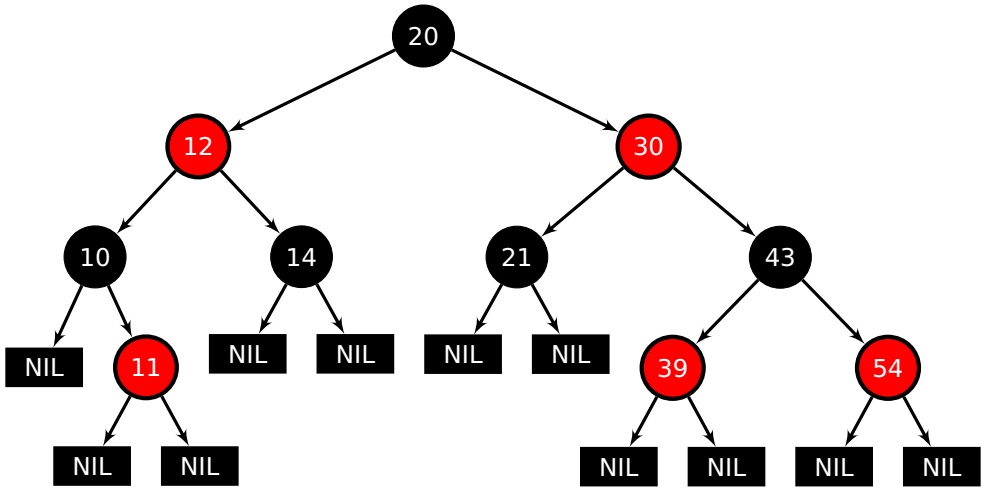


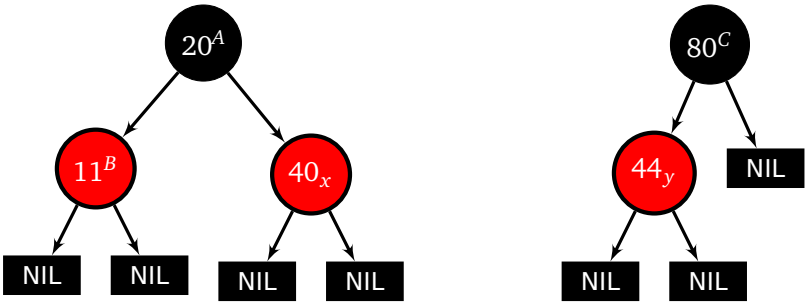
Figure 2.3: CFS Red-Black tree indexed by virtual runtime, where the process associated to the leftmost node with a virtual runtime of 10 will be executed first.

for better fairness between logical applications with more than one process, without having to consider the process count of each of the applications. It is also vital to notice that the weighting only applies when the scheduler needs to decide what processes to run, where there are other elements in the red-black tree. This means that if there is no competition, a process will continue to run uninterrupted, no matter its weight.

When selecting the next task for execution, the CFS scheduler starts at the root control group and selects the leftmost entity with the smallest virtual runtime. In case the selected entity is a process, the scheduler will schedule it. However, if the entity is a control group, it will select the leftmost entity from that group's red-black tree. It will keep traversing the tree until the chosen entity is a process, and then schedule it. An example of such a traversal in Figure 2.4.

The weighting between control groups works by modifying how the virtual runtime used as the index of the red-black tree is calculated. When a process has been executed for a given time, the virtual runtime is increased with the runtime delta divided by the weight, as seen in Equation (2.1). This means that doubling the weight will make the virtual runtime increase half as fast. The side effect of this is that it makes it possible to prioritize control groups, and it gives a more predictable behavior when there is pressure on a CPU. It also makes it easy to create low-priority groups that don't interfere with other tasks in a significant manner, while still ensuring they get a small proportion of resources when there is pressure. An example of such a hierarchy, together with the calculated CPU time each process and control group will get, can be found in Figure 2.5.

Setting weights via the control group API is only supported on control groups



(a) Red-Black tree of root group

(b) Red-Black tree of group B

Figure 2.4: CFS Red-Black forest with two trees in the control group hierarchy. Nodes with a superscript represent a control group, while nodes with a subscript represent a process. When traversing this tree, process y with virtual runtime of 44 will be set for execution.

themselves and not processes. However, the CFS Scheduler will map the *nice*[8] value of the process into a CFS weight. This will, however, only impact processes in the same control group, in the same way as group weighting only works between sibling control groups. It is only the root control group that can have processes attached to it, and having children control groups at the same time; other control groups have to be a leaf node to get processes attached to them[9].

$$\Delta vruntime = \Delta runtime \cdot \frac{1}{weight} \tag{2.1}$$

2.4.3 Fairness On Multi Core Systems

Even though the scheduler may look simple when looking at it from a single logical scheduling unit implementation, support for multiprocessors increases the complexity quite extensively. First off, each scheduling unit, often referred to as a logical Linux CPU, has a separate red-black forest for the control group hierarchy. These trees are owned and managed by their corresponding logical Linux CPU. On the x86 architecture, a logical Linux CPU is the equivalent of an SMT thread. These are often also called *Compute Unit Core* in some contexts.

To enforce fairness between groups and entities when processes in the same group run on different logical Linux CPUs, it takes the load from all processes into account when calculating how fast it should increase the virtual runtime of a given entity. To do this, all scheduling entities, both processes and groups, have their load on the system tracked. The accounting of load takes various

[8] Process priority system inherited from Unix; integer range with decreasing priority: $[-20, 19]$

[9] The Legacy/v1 control group hierarchy does not have this limitation

[10] Assuming both process B and C share the same nice value

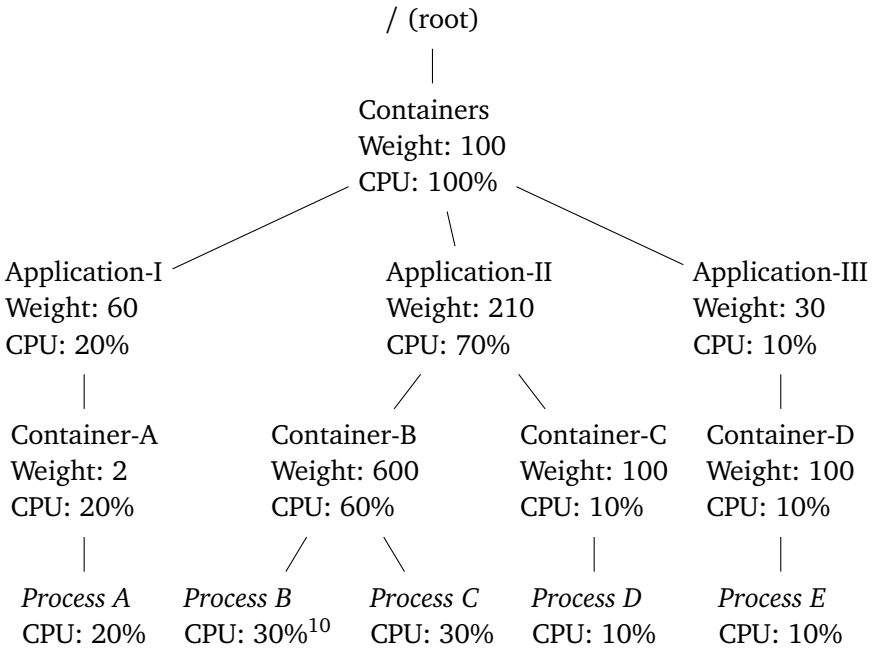


Figure 2.5: Cgroup with CPU weights, and the CPU time in percentage they will get in case all processes are running busy loops on the same logical Linux CPU.

parameters into account, making it a fair bit more complex than just summing the time spent in execution. This load tracking is called *Per Entity Load Tracking* (PELT)[10].

Overall, the result of *PELT* is that when calculating the delta virtual runtime for a timeslice used by a scheduling entity, the load of all the children scheduling entities of the control group, on all logical Linux CPUs, is used. This means that given two sibling control groups with the same weight, the virtual runtime of a scheduling entity of the busiest control group running on multiple logical Linux CPUs will advance faster than the other one. Furthermore, this means that a hierarchy like the one seen on Figure 2.5 will also preserve the weighted fairness when running on multiple logical CPUs. The absolute numbers may deviate from the target since the process count and the count of logical Linux CPUs will always stay integers. Processes may have to move around to archive the desired behavior, depending on the count of logical Linux CPUs.

To maximize the fairness and throughput on a multi-core system, the CFS Scheduler will try to load balance processes between logical Linux CPUs at a given interval. This ensures that CPUs with relatively low priority load can increase fairness by stealing processes from highly congested CPUs. When a CPU turns idle when a running process stops, and there are no other runnable processes on the CPU, it will also look at neighbors for possible processes it can steal to avoid

having to idle down. This is often referred to as CPU *load balancing*. Depending on the topology of the processors, Linux will group the logical Linux CPUs into a hierarchy of scheduling domains. Therefore, it will strive as best as possible to move and steal processes from the closest logical Linux CPUs.

2.4.4 Automatic process grouping

The initial CFS implementation did not automatically create control groups, and placed new processes into the root group, effectively losing out on the main benefit of the CFS implementation. Later, support for automatically creating new groups per tty[11] was added [11]. This made a huge impact on responsiveness, and has later been further developed. This will therefore give all the benefits described in Section 2.4.2 automatically, without any explicit configuration. Most Linux distributions in use today use *systemd* as init system, and *systemd* map all their logical units to control groups. This means that *systemd* will create a separate control group per system daemon and make a sub-hierarchy per user to group all processes, as seen in Figure 2.2.

2.5 CFS Bandwidth Control

Unlike the weight assigned to control groups for prioritizing on congested systems, CFS Bandwidth Control is a way of limiting the amount of CPU time a set of processes can use. The bandwidth control implementation was proposed in 2010 by an engineering team at Google [12], trying to discover badly behaving applications and hard cap them to limit the overall damage on the system and other applications. With group weighting, it is possible to set the minimum amount of CPU resources a control group should get, while using bandwidth control to limit the maximum amount. CFS Bandwidth control is a part of the *cpu* control group controller.

Together with making it possible to limit the damage a misbehaving set of processes can do to a system, bandwidth control also significantly improves the predictability of workloads across different types of hardware. More and more often, a homogeneous and semi-homogeneous set of servers are set up in a compute cluster where a broad set of applications run together, with various priorities and quality of service agreements. This dramatically improves the overall utility of the hardware since more tasks can share the same physical hardware and scale depending on the overall cluster load. CFS bandwidth control help applications to scale correctly in respect to their allocated resources instead of the system they are running on. Since the amount of CPU time is hard limited, this means that a CPU-bound task will work similarly on a machine with two logical Linux CPUs and one with $100x$ that amount.

[11] TeleTYpewriter. Often used for referring to a terminal session.

Another critical aspect bandwidth control helps with is billing and accounting purposes. Measuring total CPU time over a few hours or days makes a reasonable estimate of the overall average use over time, but it does not take spikes into account. Spikes in CPU usage often come with a cost of latency for other tasks on the system, and the cost of hardware does not scale linearly with logical Linux CPUs, making larger servers way more expensive than smaller ones.

CFS Bandwidth control is an integral part of all Linux Container runtimes and is vital to make it possible to control how CPU time a container or application can use at a maximum. It is used as the tool for limiting CPU usage by the container orchestration tool Kubernetes [13], and it is used directly by mapping container CPU limits to an equivalent CFS bandwidth configuration. The same applies to the well known container runtime *Docker* [14]. CFS bandwidth control is also mentioned as a critical aspect of Borg [1]. Borg is the internal container orchestrator at Google.

2.5.1 Bandwidth Control API

Via the control group API, as seen in Section 2.2.1, the bandwidth control exposes two configuration parameter per control group;

- **Period**: The period used when accounting CPU bandwidth. Default: *100ms*.
- **Quota** The total amount of CPU time that the control group can use in each period. At the beginning of a new period, the available quota will be set to this value. Default: *unlimited* (disabled).

The *period* and the *quota* can be seen in Figure 2.6.

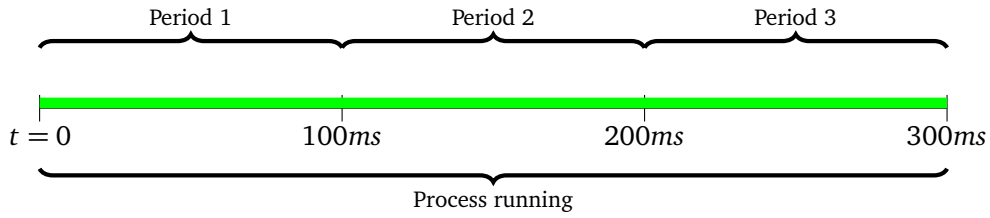
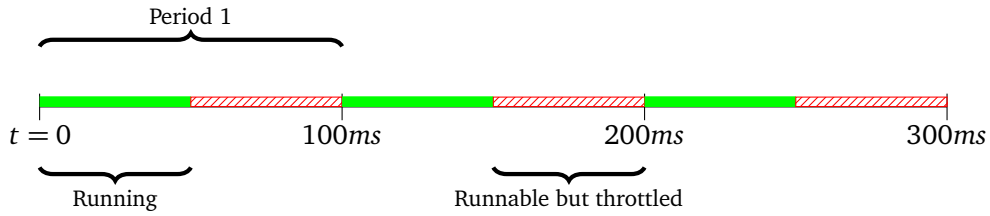


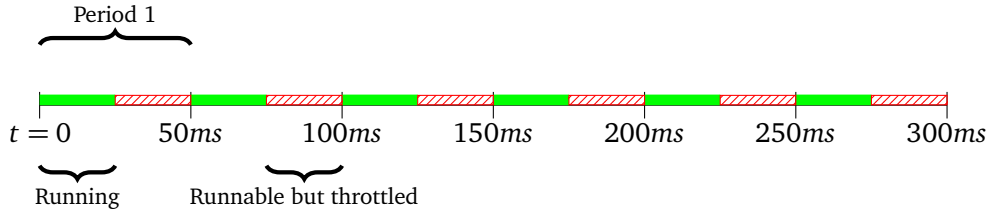
Figure 2.6: Timeline of a cpu bound process running with a period of 100*ms* and a quota of 100*ms*. The quota is the total CPU time that can be used per period.

The ratio between the *quota* and the *period* can be seen as the number of logical Linux CPUs a control can continuously use without being throttled. This means that a period of 100*ms* and a quota of 300*ms* can be viewed as an equivalent to getting 3 logical Linux CPUs. There is no special meaning to integer ratios, and non-integer ratios are also supported. This is useful for small low priority tasks, where a period of 100*ms* and a quota of 50*ms* can be viewed as the equivalent of half of a logical Linux CPU, as seen in Figure 2.7a. This can also be viewed as

an equivalent to a control group with a period of $50ms$ and a quota of $25ms$, as seen in Figure 2.7b.



(a) Timeline with $100ms$ period and $50ms$ quota



(b) Timeline with $50ms$ period and $25ms$ quota

Figure 2.7: Timeline of a CPU bound process running with a quota to period ratio of 0.5, with two different periods. The actual CPU time used will be the same in both cases.

It also exposes these statistics per control group;

- **Periods**: The number of periods where processes in the control group have been active and executing.
- **Periods throttled**: The number of periods where the control group has been throttled; stopped from running due to using too much CPU time on one or more logical Linux CPU.
- **Time throttled**: The total time the control group has spent throttled, stopped from executing due to using too much CPU time. Summed across all logical Linux CPUs.

2.5.2 Enforcement

The scheduler itself continuously enforce the CFS bandwidth control configurations on the system, and all the time a descendant processes of the given control group is executing, is accounted for. If the *quota* is set to *unlimited*, all bandwidth control accounting for the given control group is disabled. The enforcement is divided into two main parts, which exists on a per control group basis;

- **The global pool**: The global pool is an integer protected by a spinlock, holding the runtime available for use, and is filled/set to the given quota at the beginning of each period. When the global pool is zero, there is no

more runtime available in the current period. Such local pools exist per control group.

- **The local pool**: Each logical Linux CPU, as defined in Section 2.4.3, has a local pool that is used to account for the runtime available and used on that logical Linux CPU. Such local pools exist per control group, per logical Linux CPU.

An example of a control group running with CFS bandwidth enabled can be seen in Figure 2.8. As seen in Figure 2.9, not all local pools will end up being throttled at the same time.

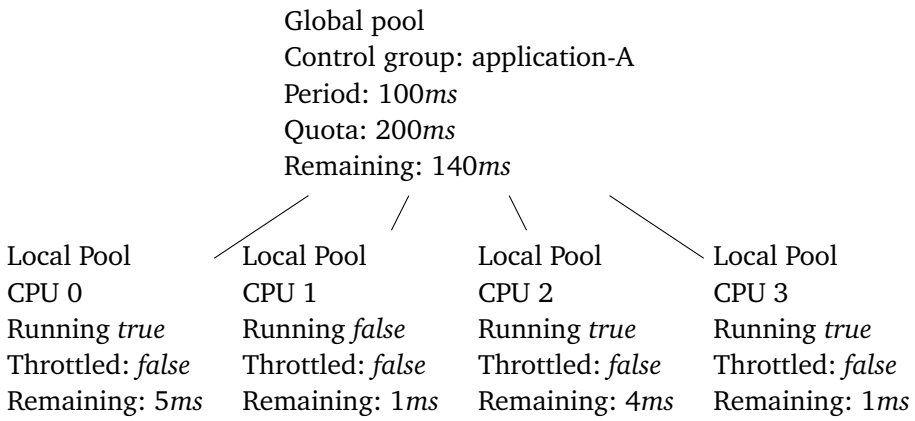


Figure 2.8: Control group with the equivalent of 3 logical Linux CPUs set via CFS bandwidth, running on a system with 4 logical Linux CPUs, and still having remaining runtime.

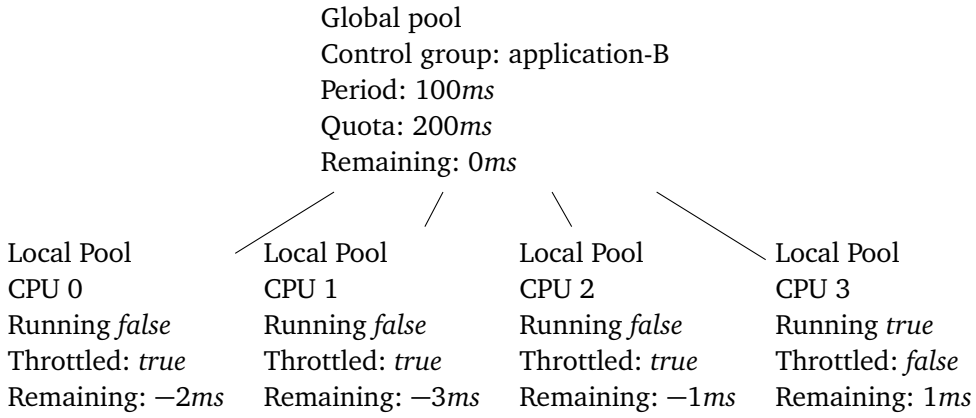


Figure 2.9: Control group with the equivalent of 3 logical Linux CPUs set via CFS bandwidth, running on a system with 4 logical Linux CPUs, where three of the four local pools have been throttled. The negative remaining runtime has to be accounted on next quota refill.

When a children process or scheduling entity of a given control group with CFS bandwidth enabled has been executed for a given amount of time, that time is removed from the CPU local pool. This accounting happens when the running process is changed, as well as on scheduler ticks. If the local pool is smaller or equal to zero, the scheduler will acquire more runtime. It will then try to take the lock protecting the global pool, and acquire an amount that can make the local pool equal to a *slice*. The value of a *slice* is a system-wide value that system administrators can configure at runtime, and defaults to $5ms$.

When a local pool is unable to refill from the global pool, and the local pool has a value equal to or smaller than zero, it will be *throttled*. The scheduler will then mark all the descendant scheduling entities below the control group as throttled, making sure they will not be scheduled. This will happen per local pool, and throttle of one local pool does not mean that other local pools will be throttled. On the rising edge of the next period, the timer responsible for refilling the global quota will then try to distribute the newly released quota between the throttled local pools to make them able to run again. When the throttled local pools are filled, they will be *unthrottled*, and the scheduler will mark all the descendants as ready to schedule again.

Since CFS bandwidth control is a control group controller, CFS bandwidth constraints can be set in each level of the control group hierarchy described in Section 2.2.1. This means that in case one local pool is throttled, all the local pools connected to the same logical Linux CPU, on all the descendant control groups, will also be marked as throttled.

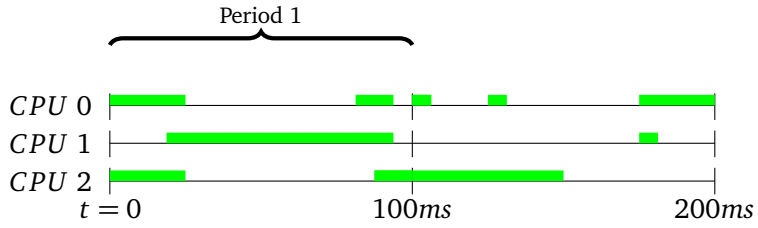
2.5.3 Alternatives To CFS Bandwidth Control

There are currently no alternatives to using CFS Bandwidth Control in the Linux kernel, but some of the same benefits can be achieved via the control group controller *cpuset*. The *cpuset* controller allows administrators to limit what logical Linux CPUs the leaf processes under a control group can run on. This is often referred to as *CPU pinning*. This is useful on specific latency-sensitive workloads that run on systems without a uniform memory layout due to increased latencies on memory access.

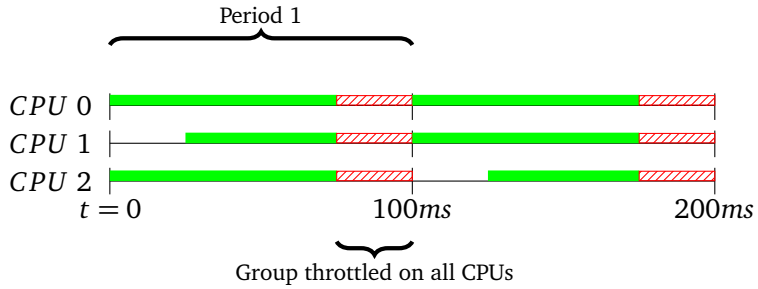
The major drawback of *cpuset* is that it will interfere with the way the CFS scheduler deals with fairness across different logical Linux CPUs, as seen in Section 2.4.3. *cpuset* is, therefore, most commonly used when a given control group is allowed access to an exclusive set of logical Linux CPUs, keeping the interference relatively low. CPU pinning also limits the possibility to overcommit CPU resources, especially when exclusivity is used.

The major selling point of CFS Bandwidth Control is that a group with the equivalent of n logical Linux CPUs can run on more than n logical Linux CPUs at once. As seen in Figure 2.10a, 3 processes can run simultaneously if necessary,

even if the CFS bandwidth limit is equivalent to 2 logical Linux CPUs. This means that it can handle small spikes of load more efficiently due to the extra access to computing resources. When pinning a group to n logical Linux CPUs, they will never be allowed to execute on more than those n logical Linux CPUs. However, as seen in Figure 2.10b, the scheduler will throttle the control group in case the usage surpasses the given quota in a period.



(a) Timeline with 100*ms* period and 200*ms* quota, and three processes running on three different CPUs. Using less than the given quota in both periods.



(b) Timeline with 100*ms* period and 200*ms* quota, and three processes running on three different CPUs. Using more than the given quota in both.[12]

Figure 2.10: Two timelies where both have a 100*ms* period and a 200*ms* quota, giving the equivalent of two logical Linux CPUs. Each with three processes, with different load.

[12] In a real life situation, not all the CPUs will end up being throttled at the same time.

Chapter 3

The Caveats And The Challenges Of CFS

Like all kinds of kernel programming, with the scheduler being no exception, making sure regressions or other adverse side effects of changes or new features don't happen, is hard. Since most users use older kernel versions, and rarely start using new ones, bugs can live for a long time before being surfaced. Small scheduler regressions are also hard to spot, and multiple bugs have sneaked into the mainline over the years.

In this chapter, we will look at the challenges and the caveats of the CFS scheduler and its design choices. We will briefly look at the general scheduler design and fairness aspects, but will primarily focus on the CFS bandwidth functionality. We will investigate potential situations leading to unnecessary throttling, and look at the consequences of such behavior. We also take a look at other related research on the topic.

3.1 Challenges And Caveats Of Fairness CFS Scheduler

The CFS scheduler is an essential part of the Linux kernel, and although the design is elegant, there are a set of complex challenges it tries to solve in the best way possible. In this section, we will look at some of the challenges with the current implementation.

3.1.1 Control Group Load Calculation

The design of the CFS scheduler makes it possible for logical Linux CPUs to handle most of the priority handling between control groups and processes on a CPU local level; some synchronization is however necessary. When a scheduling entity corresponding to a control group does its accounting, and calculates the updated load, it propagates that load to the global load sum for the control group. This is done by adding the difference between the old and the new load of the scheduling entity to the control group-wide load sum. This operation is atomic and makes sure all the logical Linux CPUs' load is accounted for properly. To limit this cross-communication and avoid costly atomic operations, this is only done when strictly necessary.

As described in Section 2.4.3, the Per Entity Load Tracking accounts for various aspects of the life cycle of the process and scheduling entity to define how much they impact the system. The load is not strictly tied to the time a scheduling entity is running; other aspects are also accounted for. This means that a CPU local scheduling entity can have a non-null load even though it no longer contains any processes. The load for all scheduling entities is continuously decayed in the background at a given interval, even for idle entities, to ensure that the calculated load is correct. This means that after some time, an idle control group will eventually reach a load of zero.

To limit these background updates, the CFS scheduler keeps a list of active scheduling entities that have load attached to them. This is important to avoid unnecessary work on entities without load and on entities that cannot be scheduled and don't contribute to load in their ancestors. The total number of scheduling entities on the system is the number of control groups with the *cpu* controller enabled, multiplied with the number of logical Linux CPUs. In a real-world scenario, especially on an underutilized system, most of these entities will be idle and without load. Initially, the CFS scheduler added all entities to this list the first time it queued them for execution. It was later added a check to remove all fully decayed entities with zero load [15]. It has also been patched to remove entities throttled due to CFS Bandwidth constraints, before adding them again during the unthrottling procedure [16].

3.2 Challenges And Caveats Of CFS Bandwidth Control

The CFS bandwidth control functionality has challenging aspects that have to be handled, often with contradictory objectives. First of all, precision is essential when it comes to both accounting and enforcement. However, greater precision and stricter accounting often lead to more overhead, especially when cross NUMA[1] communication is necessary. In this section, we will look at the challenges and caveats of the current CFS bandwidth control implementation, and how they can result in either throttling or increased system overhead.

3.2.1 The Accounting Mechanism

One of the great things about the CFS scheduler is that it does not use scheduler ticks to steer the scheduling. This means that the performance and responsiveness of the scheduler don't depend on the frequency of the scheduler ticks. Again, this allows for using a lower scheduler tick frequency without sacrificing responsiveness, and essentially minimizing the overhead of the ticks themselves. The scheduler tick frequency is a compile-time configuration parameter. As seen in Table 3.1, multiple values are supported. The length of the time between two scheduler ticks is called a *jiffy*. *Jiffies* are used for kernel time measurements without the need for higher precision.

Table 3.1: The different supported timer frequencies in Linux together with corresponding jiffy length

CONFIG_HZ	jiffy
100	10ms
250	4ms
300	3.33ms
1000	1ms

However, the scheduler ticks' frequency is crucial for CFS bandwidth control due to how the accounting is done. CFS bandwidth control accounts for all the runtime used by a process. The accounting occur when the process is stopped from executing, and on scheduler ticks occurring when it is running. That means that processes running for longer than the length of a jiffy gets their runtime accounted after use, in portions roughly the size of a *jiffy*. CFS bandwidth accounting is done with nanosecond precision. Constantly synchronizing clocks, and using the timeout of the CFS bandwidth timer to calculate where accounting should be done is possible, but would result in huge amounts of extra overhead due to the extra synchronization and locking required.

As seen in Section 2.5.2, CFS bandwidth control will fill the local pool when its runtime budget is smaller or equal to zero. This means that local pools reserve

[1] NUMA: Non-uniform memory access

runtime they plan to use. However, since the granularity of accounting often is on a level with a *jiffy*, local pools often end up with negative values, meaning that they have used more than their reserved value. When a scheduler tick race with a refill of the global quota, the accumulated negative value in a local pool can be accounted for with the newly refilled global pool, even though the runtime accounted for was used in the previous period. This also applies the other way, where runtime in a local pool can originate from the previous period but can still be used in the next.

Overall, this accounting mechanism has some major drawbacks causing issues because of unnecessary throttling. All this, depending on the characteristics of the workload, together with how the system is configured.

3.2.2 CFS Bandwidth runtime expiry

To strictly enforce the CFS bandwidth a control group could use, there was previously an expiry on the runtime acquired to the local pool. This runtime expiry mechanism ensured that the control groups could only use the runtime in the same period as it was acquired. At first glance, this sounds like a good idea. It later turned out to be quite problematic. The mechanism caused a lot of extra throttling because runtime was discarded when the control groups couldn't use it, for many of the reasons as described in Section 3.2.1. It also leads to a thundering herd problem on the global pool lock when all the local pools saw that their runtime had expired, and they had to acquire more from the global pool. Notably, there were issues with the clock synchronization between the different CPU cores, causing clock drift issues that also caused excessive throttling [17]. Sometime later, the kernel maintainers removed this expiry mechanism altogether to reduce the throttling caused by it [18].

3.2.3 Length Of The CFS Bandwidth Slice

Another essential factor to accounting and enforcement is the slice length, the amount of runtime a local pool tries to get when refilling from the global pool. As described in the CFS bandwidth documentation, the length of a slice is a compromise between increased overhead and fine-grained consumption [19]. A larger slice means less communication between the local and global pools, reducing the stress on the lock. It also increases the chance of local pools taking more runtime from the global pool than they can use for the rest of the period, possibly starving and throttling other local pools. The default slice length is $5ms$.

To avoid having local pools without any runnable processes while still having runtime available, all remaining runtime except the hardcoded value of $1ms$ will be donated back when a local pool goes idle. This means that when processes run for less than $1ms$ and start and stop often, for example, when communicating extensively through busy synchronization primitives, it can be a bad thing.

Having a bigger slice than 1*ms* can, in this situation, lead to taking the global pool lock twice each time the local pool is refilled, when refilling, and when giving back when it turns idle.

Another often overlooked result of this is that when the slice is not a multiple of a *jiffy*, the local pool ends up with a negative value. This is most often not a problem, but in the area around the refill, this can be problematic, as seen in Section 3.2.1. As seen in Figure 3.1, this can, in theory, cause severe throttling on a single process in a control group with a quota to period ratio of 1. In that situation, with a 10*ms* quota, 10*ms* period, a 5*ms* slice (default), can theoretically result in a process being throttled for 3*ms* every three or four periods, possibly resulting in being throttled for up to 10% of the time. This effect increase with the decrease in the CFS period.

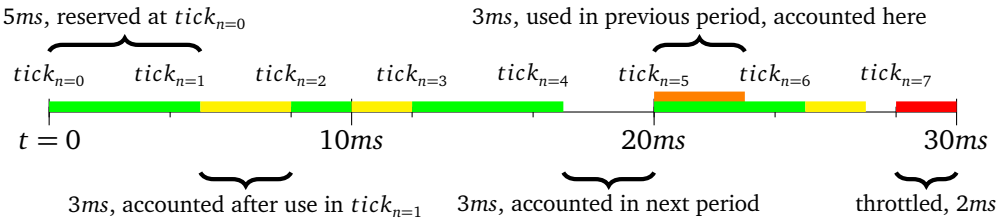


Figure 3.1: Timeline of runtime accounting on a cpu bound process, with a bandwidth period of 10*ms* and quota of 10*ms*, with slice length of 5*ms* and jiffy length of 4*ms*.

3.2.4 Overhead Of The Global Lock

Most of the accounting and reservation compromises come down to the spinlock's overhead guarding the global pool. As described in Section 3.2.3, this pressure is reduced by various aspects. However, if the slice is smaller than a *jiffy*, all CPUs running leaf processes under a control group with CFS bandwidth enabled will start spinning this spinlock once every scheduler tick. Depending on the memory layout, performance and topology of the system, this can be a costly process. This lock is also acquired during global pool refill, local pool throttling, local pool unthrottling, and during the update of the configuration parameters.

3.2.5 Timer life cycle

A high-precision timer for each control group with CFS bandwidth enabled is scheduled to refill the global pool. It uses the high precision timer, *hrtimer* functionality in the Linux kernel, that gives a higher level of precision than relying on *jiffies*. This timer has a timeout of the length of a period and is started when CFS bandwidth is enabled. To save resources, when a control group is empty or all its processes are idle, the timer is stopped. The timer is restarted again when

a local pool tries to acquire runtime from the global pool.

This mechanism will therefore also contribute to more congestion on the per global pool spinlock. To ensure that the timer is properly restarted after being stopped, the local pools restart it when refilling from the global pool. In turn, this also contributes to the congestion on the lock when the timer is running.

The timer restart also has consequences for applications with periodic bursts while staying idle for the rest of the time. As seen in Section 3.2.1, a local pool often ends up with a substantial negative value. This can potentially cause situations where an idle process in an otherwise idle control group starts executing. In such a case, it might end up with a negative local pool equal to the length of a *jiffy*. When the local pool then tries to acquire runtime from the global pool, it will also start the timer. There is then a chance that the next refill of the global pool will take a complete *period* worth of time. This means that the global pool will have a value equal to the *quota* minus one *jiffy* minus one *slice*. Therefore, this situation can also lead to throttling a control group that effectively uses less than its requested quota per period.

3.3 The Effects Of CFS Bandwidth Throttling

Even though the primary goal of CFS Bandwidth Control is to throttle control groups using more than their quota in a period, the throttling can have undesired side effects. Other than the extra, but relatively small, cost of handling the throttling, there is no particular incentive to avoid throttling a local pool in a control group from the systems perspective. When control groups should be throttled, the scheduler will throttle them. For the applications themselves, throttling can have undesired consequences, depending on the most important performance characteristics. For some applications, throttling can be harmless, while it might result in highly undesired behavior for others.

3.3.1 Overall Performance Impact

Each time a local pool gets throttled, one or more of the processes beneath it will have to wait before starting executing again. This effectively means that a single process running a batch job that gets throttled for 30*ms* will take at least 30*ms* longer to finish. Each time a local pool over the process in the control group hierarchy is throttled, this will happen.

For a batch job intended to use more CPU time than the quota, getting throttled is the desired and predicted behavior. However, when a control group cannot use all the runtime in one period and then gets throttled in the next, it effectively uses less than the quota on average, but still gets throttled due to how accounting works. This results in batch jobs inside their quota still getting throttled and taking more time than they are supposed to.

3.3.2 Impact On Latency Sensitive Workloads

Controlling and avoiding throttling is especially important for user-facing programs that strive for low and consistent latency. An example of this is web servers serving requests for users. In case such a process gets throttled for *20ms*, it will result in a delay of at least *20ms* for all requests in flight at that moment. Requests arriving during the throttling period will see the same effect but with a minor delay. When throttling starts, it is the consequence of too much CPU usage, and together with the work piling up during such a throttle, tail latencies often skyrocket.

The overall goal for latency-sensitive applications is to avoid getting throttled. The most straightforward mitigation is not to spawn more threads or processes than the amount of logical Linux CPUs the given CFS bandwidth is equivalent to. In case of severe throttling in such an application, increasing the quota is an alternative. Another alternative is to change the CFS bandwidth period. Decreasing the period will result in a higher chance of throttling but with shorter throttling periods. In contrast, a longer period will result in less throttling, while throttling will last longer. This, especially lowering the period, can result in other side effects like more throttling, as seen in Section 3.2.3 and Section 3.2.1.

As seen on Figure 3.2, going from one process to three processes that can load balance the requests helps on tail latency as long as CFS bandwidth is not enabled. However, one can see that when a control group with three processes that together use on average the equivalent of one logical Linux CPU worth of CPU time, throttling becomes a problem. Even though the mean response time is similar for all test cases, one can see that the latency explodes for the case with three processes and a CFS bandwidth with the equivalent of one logical Linux CPU. We see that the results are pretty similar for a single process when CFS bandwidth is enabled and disabled.

3.3.3 Impact On Scheduler Fairness

Often, when a control group uses close to its CFS bandwidth limit, it ends up being throttled for a small and unnoticeable amount of time without affecting the performance characteristics of the processes inside. On an idle system, such a situation may have no other side effects, but when there are more processes ready to run than logical Linux CPUs running them, fairness comes into the picture. As described in Section 2.4.3, an idle CPU will try to steal runnable processes from its neighbors. When a local pool gets throttled, the scheduler will load loadbalance since it can no longer execute the previously runnable process(es). If the scheduler can take another process from another logical Linux CPU, it will start running that process right away. When the throttled local pool is unthrottled, the scheduler will mark the processes under that control group as runnable. As described in Section 2.4.2, the scheduler will then start prioritizing between the process previously throttled and the process it took from its neigh-

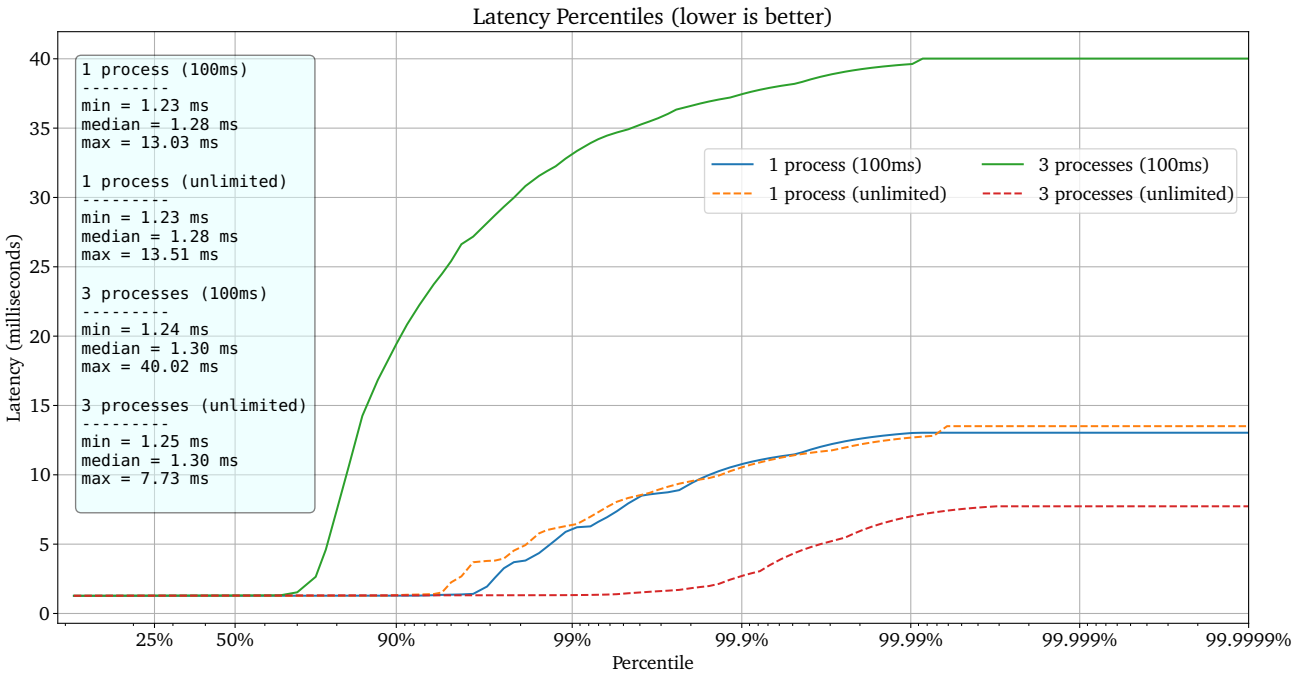


Figure 3.2: Plot of latency percentiles for webserver requests as seen in C.1 with a static load of the equivalent of 0.99 logical Linux CPU. It shows the latency effect of throttling. Using a 100*ms* period and the respective quota as shown in the parentheses.

bor, ensuring both processes gets their share of CPU time, depending on their respective weight. Then, after a while, the load balancing logic might end up moving the process back to its original logical Linux CPU, effectively reversing the previous load balance. This situation can then continue to be repeated each time the control group gets throttled, causing a high amount of context switches and unnecessary traffic between logical CPUs. This shows that even small throttling periods can have a significant impact on the fairness between groups.

3.4 Related Work

There has been a lot of work contributing to the Linux scheduler over the years, and most of the work goes unnoticed. As seen in Section 2.1, most of the development of the Linux Kernel is discussed through the public mailing lists. Most improvements are therefore laid out as *patches*, and then later discussed. The majority of the work is done by maintainers hired by companies working on the Linux kernel. However, there are countless contributions from outsiders, and the mailing lists are open for everyone to subscribe or browse online.

3.4.1 The Art of CPU-Pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms

Other than the initial proposition of the CFS Bandwidth Control functionality [12], there hasn't been much research on the overhead and its caveats. In 2020, a set of researchers from the High-Performance Cloud Computing (HPCC) lab, University of Louisiana at Lafayette, USA, looked at the overhead of using the CFS bandwidth control compared to virtualization and CPU pinning [20]. Their findings show a clear overhead using CFS bandwidth, or vanilla containers as they call it, compared to CPU pinning and virtualization. Especially when comparing the average execution time of database queries and the latency of web server requests, it shows a massive increase in the time it takes. Some experiments show an overhead of more than 3.5 compared to running with CPU pinning. However, their research does not describe application configuration regarding the number of processes spawned, or what they used as CFS bandwidth configuration. Their results show that the overhead decreases when reducing the ratio between the CFS bandwidth equivalent CPU cores given to a task and the actual core count on the machine.

It is not clear if they researched with the fixes described in Section 3.2.2, making it impossible to know if those affect their results. They later conclude by stating that users should avoid containers, or control groups, with small CPU limits compared to the total CPU count on the system. They say that CPU-bound applications should use CPU pinning when possible to minimize the overhead.

Although these conclusions are based on their data, there are still many unanswered questions about what factors contribute to the results. They executed their tests on a four-socket NUMA machine, so communication and memory-intensive applications running on different NUMA nodes will result in higher latencies and increased overhead. Most modern applications today also spawn as many threads as there are logical Linux CPUs on the system. This means that even though CFS bandwidth allows the equivalent of 4 logical Linux CPUs, the application researched could have spawned 112 threads. This would result in all 112 threads running for a few milliseconds at the beginning of each period, before being throttled. This results in significant congestion on the spinlock guarding the global CFS bandwidth pool for the control group. It would also result in a substantial overhead when all 112 processes try to communicate simultaneously. This would be especially evident when they try to access and update shared memory possibly guarded by locks.

Part II

Design Improvements And Implementation

Chapter 4

Reducing Unnecessary CFS Bandwidth Throttling

In this chapter, we examine the reason for unnecessary CFS bandwidth throttling, and propose two possible solutions to mitigate the issue. We will also look at how this throttling affects the processes being throttled. Our results show that our proposed techniques reduce the runtime of CPU bound tasks on an otherwise busy system, by as much as **3%**. They also reduce the variance in runtime dramatically, with a decrease in runtime of about **5%** between the worst baseline results and the worst results using our techniques.

4.1 Introduction

Starting the main part of the project, we aim to reduce the unnecessary throttling due to the way the runtime is accounted for. As seen in Section 2.5, as long as a local pool has a positive amount of runtime remaining, it is allowed to continue running one of its children processes. As seen in Section 3.2, this often results in local pools getting in a situation where they have a substantial amount of negative accumulated runtime. This negative runtime value will always have an upper bound of the size of a *jiffy* per local pool, since the bandwidth controller will always account for runtime at least once on each scheduler tick. For highly threaded applications running simultaneously on multiple logical Linux CPUs, this can add up to a fair bit compared to the total quota per period.

When a local pool accounts for its running time, and the total amount of runtime available is less or equal to zero, it has to refill itself from its respective global pool. When filling itself from the global pool, the local pool has to pay back its negative runtime, the runtime used without previously having reserved for it. If this accounting happens right after the quota refill, the debt is paid back using the quota from the new period. The local pool can as a result from this, end up being throttled for a small amount of time at the end of the period, even though it has used less runtime in the given period than the quota.

Another major reason for throttling is due to the timer life cycle described in Section 3.2.5, where the runtime is not adequately accounted for since the timer may be started too late. A similar issue occurs when the CFS bandwidth controller is configured from userspace. When a user-space application updates the CFS bandwidth configuration, the controller will reset all local pools to zero and refill the global pool even though the values are the same. In such a case, depending on how long the slice is, it can cause throttling. System daemons or container runtimes often set these values regularly to ensure the values are correct.

To mitigate these issues, we propose two different solutions. We first introduce a way of accounting for negative accumulated values in local pools using the previous period's quota. Next, we propose a set of improvements to the CFS bandwidth updating logic to minimize the throttling issues occurring in that situation.

4.2 Implementation

In this section, we look at the implementation of the proposed changes to reduce the throttling.

4.2.1 Adding Accounting Slack On Period Start

To allow for backward accounting, we modify both the refilling of the global pool and the local pools.

Code listing 4.1: Original CFS global pool refill logic

```
1 void refill_global_pool(cfs_global_pool global_pool)
2 {
3     global_pool->runtime = global_pool->quota;
4 }
```

As seen in Code listing 4.1, the remaining runtime quota from a period is discarded during a refill. We define this runtime as the runtime *lost* on quota refilling. To mitigate this, we introduce a new attribute to the global pools, called `slush_fund`.

Code listing 4.2: Modified CFS global pool refill logic

```
1 void refill_global_pool(cfs_global_pool global_pool)
2 {
3     global_pool->slush_fund = global_pool->runtime;
4     global_pool->runtime = global_pool->quota;
5     global_pool->period_nr += 1;
6 }
```

During refill of the global pool, we save the old runtime to the `slush_fund`, before refilling the global pool again, as seen in Code listing 4.2. At the same time, we track the period number.

Code listing 4.3: Local pool refill using slush fund

```
1  int refill_local_pool_with_slush(cfs_global_pool global_pool,
2      cfs_local_pool local_pool)
3  {
4     if (global_pool->period > local_pool->period_nr){
5         int refill = min(global_pool->slush_fund, -local_pool->remaining_runtime);
6         global_pool->slush_fund -= refill;
7         local_pool->remaining_runtime += refill;
8
9         local_pool->period_nr = global_pool->period_nr;
10    }
11    return refill_local_pool(global_pool, local_pool);
12 }
```

We later, when refilling a local pool, use this value to account for the negative accumulated runtime we see on the first refill in a new period, as seen in Code listing 4.3.

Together with a slice value of $1\mu s$, the lowest possible in the kernel, this solution will mitigate all throttling as long as the ratio between CFS bandwidth quota and its period is smaller or equal to the count of processes in the control group as in Equation (4.1), or CPUs in the available *cpuset* as in Equation (4.2).

$$\frac{quota}{period} \geq |procs| \tag{4.1}$$

$$\frac{quota}{period} \geq |cpus| \tag{4.2}$$

To mitigate the throttling caused when starting the idle timer, as described in Section 3.2.5, we make sure the new accounting mechanism takes that into account.

The proposed patch implementing this can be seen in Appendix A.1.1.

4.2.2 Updating The CFS Bandwidth Update Logic

To ensure that the CFS bandwidth control is not interfered with when configuration parameters are changed, we update the logic to ensure that no configuration updates are done unless strictly necessary. In order to keep this simple, we verify if changes are setting the same value as before during a configuration update and turns that into a NOP[1] that doesn't interfere with the other logic.

When either the period or the quota is updated with a new value, the behavior will continue as before. Due to how high-resolution timers work in Linux, changing the period will only affect the period length of the next period, not the current one. In the same way, setting a new value to the quota will result in a quota refill with the new quota and a reset of all the local pools.

The proposed patch implementing this can be seen in Appendix A.1.2.

4.3 Results

To investigate the absolute amount of throttling in CPU intensive applications, how many times one or more local pools are throttled, and for how long, we use the *Sysbench CPU benchmark* as described in Appendix C.2. We do this to be able to view how much less throttling we see with our proposed changes. During all tests in this section, we keep setting the CFS bandwidth once a second. We do this to properly mimic the behavior of container runtimes and other init systems like *systemd*.

As described in the details about the Implementation of the changes, as seen in Section 4.2, we divide our changes into four configurations. We abbreviate them as following:

- **Baseline**: Unpatched Linux Kernel.
- **Slack**: Adding Accounting Slack On Period Start, as seen in Section 4.2.1.

[1] no operation

- **Config**: Updating The CFS Bandwidth Update Logic, as seen in Section 4.2.2.
- **All**: Both the proposed changes.

4.3.1 Sysbench CPU Benchmark

Absolute Throttling

Table 4.1: Baseline results for Sysbench CPU Benchmark with 30 0000 events with three processes, on three exclusive logical Linux CPUs. Mean values of 10 executive runs.

Period	Quota	Periods	Periods Throttled	Throttled Ratio	Throttled Time
N/A	Unlimited	734	0	0%	0 ms
100 ms	300 ms	735	50	6.8%	94 ms
50 ms	150 ms	1486	661	45%	2 457 ms
10 ms	30 ms	7840	3835	49%	14 608 ms

As a baseline, we run the test without any changes. In this situation, as seen in Table 4.1, there is an increasing amount of throttling when there is a decrease in the CFS bandwidth period. This occurs even though the ratio between the CFS bandwidth quota and period stays the same, with the same value of both running processes and logical Linux CPUs. All this throttling can therefore be seen as unnecessary. We also see that while the time throttled while using a 100*ms* period is a modest 94 milliseconds, while it is more than 14 seconds while using a 10*ms* period. As described in Section 2.5, the number of periods where one or more local pools are throttled is impossible to predict, and a higher throttle count does not strictly imply longer runtimes or more latency. The same applies to the time throttled since processes can be moved to other local pools and executed there. However, the values can be used as a proxy value for investigating the impact of throttling, even though the values have to be read and interpreted with a grain of salt. After all, the most important part is the actual runtime of the program being executed.

Looking at the results in Figure 4.1, we see that there is a substantial amount of throttling, with it happening at about 5 to 10 percentage of the periods. We clearly see that when running with *Config*, the number of periods with throttling is about halved. The most significant change is, however, when using *Slack* and when using both. We then see zero throttling.

As seen in the results in Table 4.1, the throttling increase with a decrease in the CFS bandwidth period. Looking at the results when using a 10*ms* period in Figure 4.2, we see that the number of periods with throttling is increasing with the *Config* change. However, when looking at the difference between those, the relative change is pretty small. Even though the throttling is not entirely mitigated as seen when using a 100*ms* period, we see that the number of periods where throttling occurs is reduced from about 50% to less than one whe using both

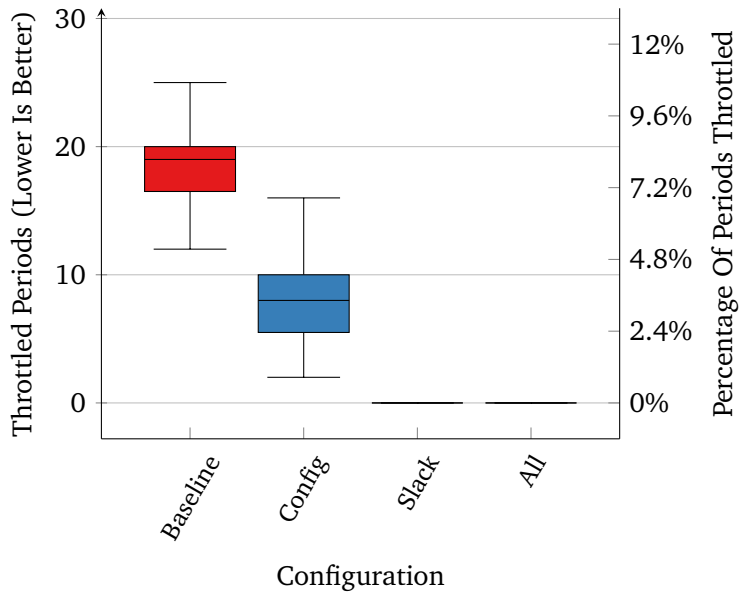


Figure 4.1: Sysbench CPU Benchmark result, showing periods throttled with the given changes. All results with a 100ms CFS Period and 300ms quota, and three threads. Values based on 30 executive runs.

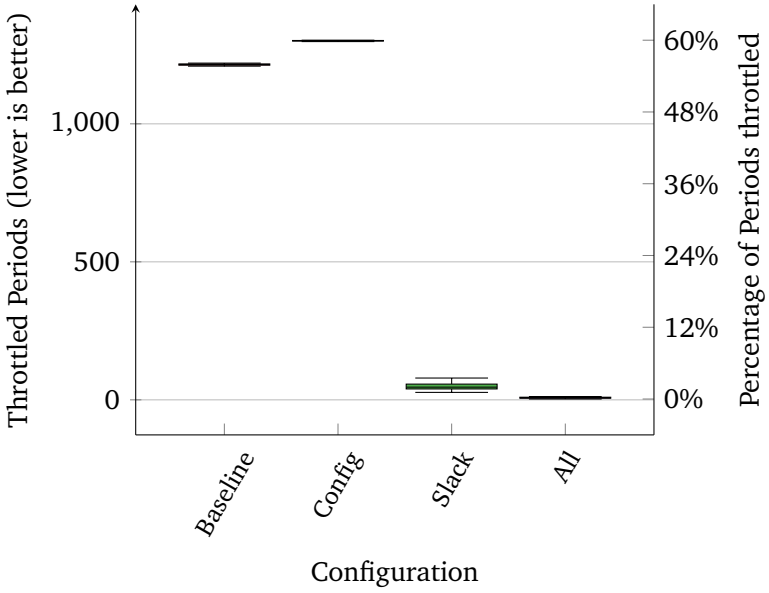
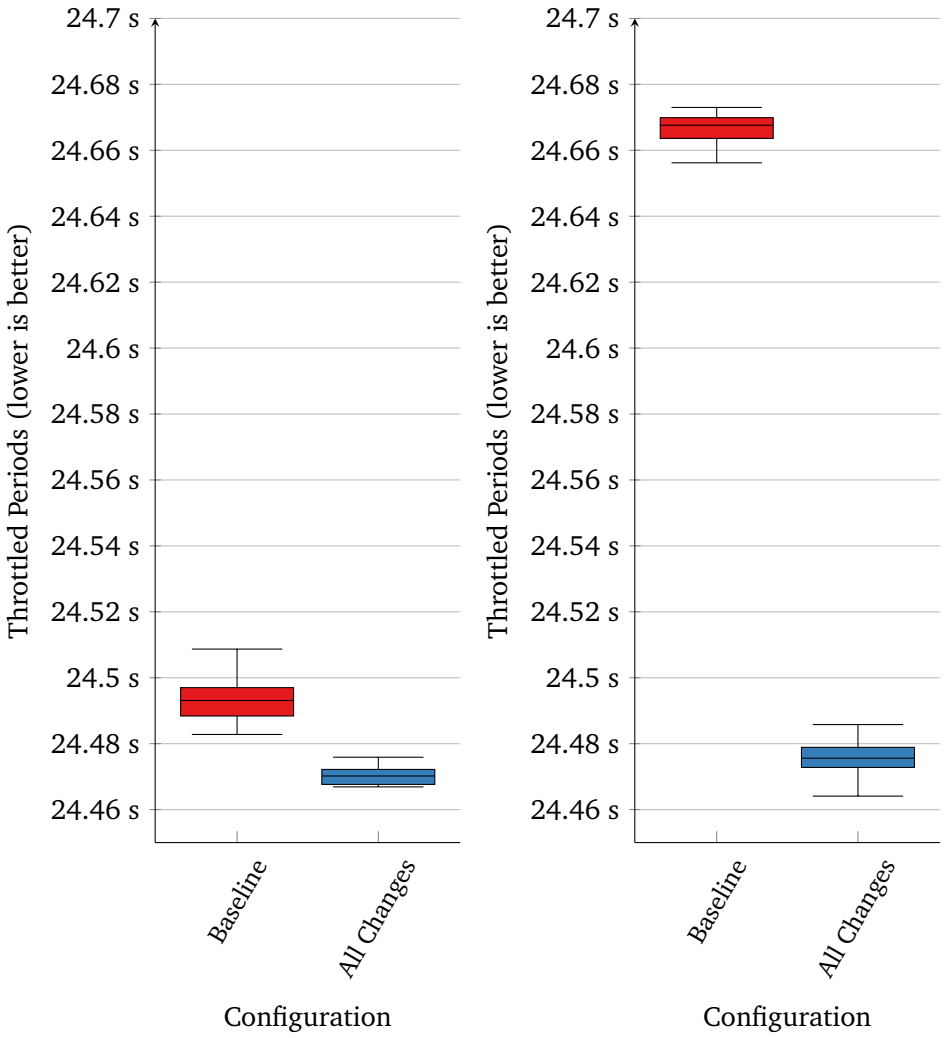


Figure 4.2: Sysbench CPU Benchmark result, showing periods throttled with the given changes. All results with a 10ms CFS Period and 30ms quota, and three threads. Values based on 30 executive runs.

changes.. Even though the throttling increased when using *Config*, we see a decrease in throttling from *Slack* compared to when using both. Even though the

throttling in both cases is more or less negligible, the throttling when using both is about four times smaller compared to *Slack*. We also see that the variance is much more significant for *Slack* compared to when using both.

Runtime impact



(a) 100ms Period **(b)** 10ms Period

Figure 4.3: Sysbench CPU Benchmark result showing runtime. Values based on 30 executive runs, all using three threads.

When looking at the total runtime in Figure 4.3, we can see that the throttling does have an impact on the runtime of a program. There is a relative decrease in overall runtime when running with the proposed changes, although it is not huge compared to the total runtime. For the 100*ms* period test, this translates to a reduction in the runtime of about 1‰. For 10*ms* period test, this translates to

just below 1%. However, we see that when using the 100*ms* period, where the result was no throttling at all, the variance is reduced significantly.

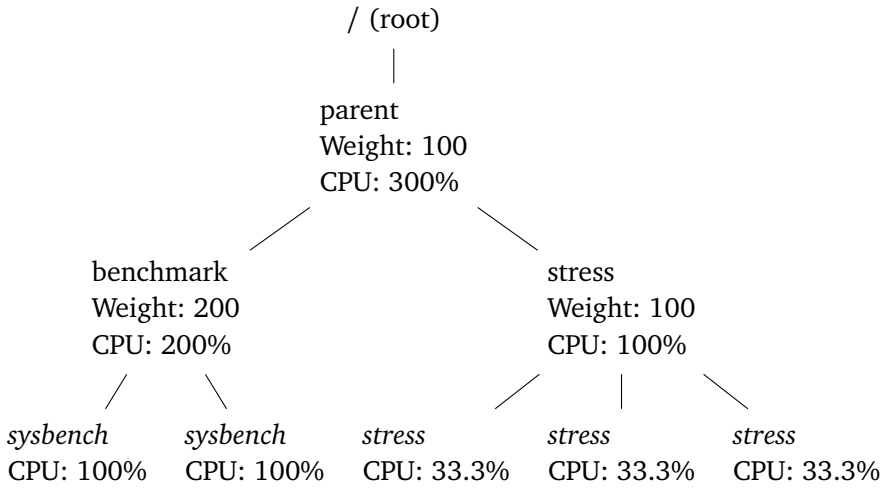


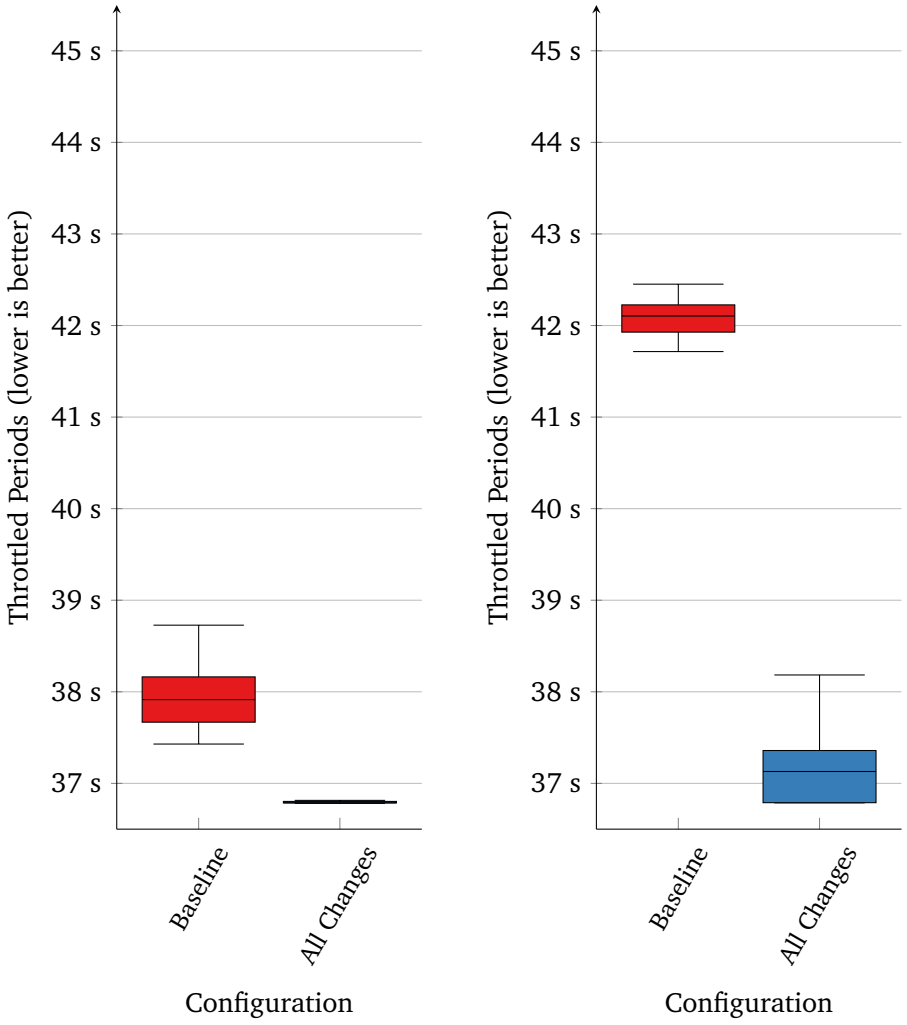
Figure 4.4: Cgroup hierarchy together with internal weighting, and predicted CPU load during congestion on three logical Linux CPU. 100% load translates to one logical Linux CPU.

When running on a CPU congested system with a control group hierarchy as seen in Figure 4.4, we get the results as seen in Figure 4.5. We see a significant reduction in runtime, where using 100*ms* period reduces the overall runtime with 3% when calculating using the median value. The variance is reduced significantly and is more or less nonexistent with the proposed changes. When looking at the highest runtime seen, there is a 5% decrease when using the proposed changes. When using a CFS period of 10*ms*, we get a total reduction in the runtime of almost 12%, even though there is a great amount of variance in that case. This result might, however, be slightly different on different kernel setups.

4.4 Discussion

We see that both our proposed patches effectively reduce unnecessary throttling, no matter what CFS bandwidth period is used. The patches decrease the percentage of periods being throttled from 50% to less than 1% given a 10*ms* period, and from about 6% to zero percent when using a 100*ms* period, where the 100*ms* period is the one used by default. Even though it is hard to estimate the added overhead to the CFS bandwidth implementation with the slack accounting change, it will during most local pool refills only consist of a single new branch, so that should not be substantial.

As seen in the results, when using a short CFS Bandwidth period and the default



(a) 100ms Period **(b)** 10ms Period

Figure 4.5: Sysbench CPU Benchmark result showing runtime in a busy environment as described in Appendix C.4. Values based on 30 executive runs, all using two threads.

slice length of 5*ms*, we still see some throttling. As described in Section 3.2.3, this is the result of local pools acquiring an entire slice right before the timeout. The simplest way to avoid this is to lower the slice length. Another possible solution is to do the opposite of our proposed accounting using the previous runtime quota, using the quota of the next period. If a local pool is unable to refill itself, it can calculate the time remaining before the global pool is refilled. If the remaining time before a refill is smaller than the length of a slice, the local pool can refill using the quota of the next period. However, the amount of throttling seen is negligible, and with a slice length on half the period, some throttling must be expected.

Overall, the most interesting part is that we see the runtime of the benchmark in an otherwise busy system decrease 3% when running with our changes. This clearly shows the effect described in Section 3.3.3, where other processes are being moved to the logical Linux CPU when the local pool of the benchmark is throttled. After the unthrottle of the local pool, the processes will keep fighting for runtime, before eventually the load balancer algorithm described in Section 2.4.2, move the processes to reach an equilibrium in regards to fairness. It is also evident that the throttling mechanism is unpredictable and causes vast amounts of variance in runtime between different runs. When our proposed changes mitigate the short and unnecessary throttling, we see that this variance decrease dramatically.

Chapter 5

Reducing The Overhead Of CFS Bandwidth Control

In this chapter, we measure the overhead of CFS bandwidth control, and implement an alternative distribution solution using atomic variables instead of the current spinning lock. We see how using atomic variables reduces the overhead of CFS bandwidth control and how it allows for more fine-grained control of runtime distribution. Our results show that we are able to reduce the overhead of CFS bandwidth control by about **95%** on scheduler-intensive workloads when using short slice lengths.

5.1 Introduction

As seen in Section 3.2.4, the global spinlock used to protect the global pool is one of the most significant factors when it comes to the overhead introduced by CFS bandwidth control. This spinlock has to be acquired each time the global pool is modified, and when multiple logical Linux CPUs are waiting on it, this introduces overhead. This overhead increases with the number of logical Linux CPUs running processes below the same control group with CFS bandwidth enabled.

Although spinning locks are quite frequently used in kernel code, they have some overhead. As long as there are no other users of the spinlock waiting, this overhead is negligible. Even when others are waiting, spinlocks perform well. The major problem occurs when there are many users, logical Linux CPUs, trying to acquire the same lock simultaneously; something possibly happening multiple thousand times per second in CFS bandwidth control.

Most computer architectures implement atomic instructions that allow for atomic modification of memory. For integer values, these instructions can be used to increment or decrement the value, together with getting or setting the value.

As the most essential part of the CFS bandwidth control implementation is the move of quota from a global to local pools, we propose a new mechanism using atomic variables instead of the current spinning lock.

5.2 Implementation

In this section, we look at the implementation of the proposed changes to reduce CFS bandwidth overhead.

5.2.1 Using An Atomic Integer For The Global Pool Quota

Linux has a wrapper for atomic variables via the *atomic_t* and *atomic64_t* types [21], together with implementations for each supported architecture. Architectures without native support for atomic variables fall back to transparently using a spinlock, meaning that there should be no performance penalties for systems without such instructions.

In order to use these types, we introduce an atomic 64 bit integer on the global pool. This replaces the old non-atomic version.

Code listing 5.1: Atomic implementation refill_local_pool

```
1 int refill_local_pool(cfs_global_pool global_pool,
2     cfs_local_pool local_pool)
3 {
4     int target_runtime = slice_length;
5     int amount = (target_runtime - cfs_rq->runtime_remaining);
6     int old = atomic_fetch_and_subtract(amount, &global_pool->runtime);
```

```
7
8       if (old < amount){
9           if (old <= 0){
10              amount = 0;
11          } else {
12              amount = old;
13          }
14      }
15      local_pool->runtime_remaining += amount;
16      return local_pool->runtime_remaining > 0;
17  }
```

When using this atomic integer, we decrement the atomic variable during local pool refill and inspect its previous value to determine the result. Given the atomic nature of the decrement function, we have three possible outcomes, depending on the old value of the global pool:

1. *The old value is bigger or equal to the amount we try to acquire*

- If the old value is bigger or equal to the requested amount, we see the operation as successful. We can then conclude with the fact that we got that amount from the global pool, and we can refill the local pool with that amount. The local pool can then continue to run.

2. *The old value is positive, but smaller than the amount we try to acquire*

- If the old value is smaller than the requested value, but it is still positive, we view it as a partial refill. We then refill the local pool with the old value of the global pool. Depending on the resulting value of the local pool, it can either continue executing in case the new value of the local pool is positive, or it has to be throttled in case the new value of the local pool is zero or smaller.

3. *The old value is negative*

- In case the old value is negative, the global pool is empty. The local pool will then have to be throttled and await a refill of the global pool.

During the refill of the global pool, the quota can be set via an atomic *set* instruction. This is also the same for most cases where the global pool is modified throughout the scheduler. During *throttling* and *unthrottling* of the local pools, together with the configuration of the CFS bandwidth parameters, the spinlock will have to be used. This ensures that the other parts of the global pool stay correct and avoid all race conditions.

In order to ensure the maximum possible performance on the atomic modifications, we ensure that the runtime of the global pool has its own cache line. This

ensures that using the spinlock and changing the other attributes of the global pool will not affect the performance of the atomic operations.

There are, however, a few caveats when it comes to the pausing and the restart of the period timer, since that relies on the spinlock of the global pool. The proposed implementation disables this mechanism, so further implementation work will be required to be production-ready.

The proposed patch implementing this can be seen in Appendix A.2.1.

5.3 Results

In this section, we look at the implications of CFS bandwidth throttling, and how our proposed changes are helping to mitigate those changes.

In order to test the overhead of the CFS bandwidth implementation, we use the *Sysbench Threads benchmark* as described in Appendix C.3. In all tests, we use a 100*ms* period, together with a quota that will never be reached. We do this to avoid measuring the impact of throttling itself, just the overhead while running without being throttling. In order to measure the overhead, we test with three different slice lengths:

- **5 ms**
    - The default value
- **1 ms**
- **1 μs**
    - The smallest possible value

Even though a slice length of 1μs is not a sensible value to use in real life, it will significantly increase the lock congestion allowing us to see the difference between using the previously used spinlock and our proposed atomic variable.

5.3.1 Test configurations

Due to how CFS bandwidth is implemented, we define two baselines in this test. This gives us a total of four different test configurations;

System Disabled

The first baseline we define is the *System Disabled* configuration. The *System disabled* baseline is the situation where no control groups on the system have CFS bandwidth enabled. This is necessary because CFS bandwidth uses a kernel function called static branching [22]. This is a low-level optimization that live patches the running kernel based on a condition. This is used to replace conditional jumps with static jumps without any overhead. This improves the

performance as the jumping condition does not have to be calculated when the code is reached, as the running kernel is modified to reflect it. This essentially means that there will be a slight difference in the scheduler code flow when no control groups have CFS bandwidth enabled, compared to when one or more has.

Group Disabled

The second baseline we define is the *Group Disabled* configuration. The *Group Disabled* baseline is the situation where non of the ancestors' control groups of a process have CFS bandwidth enabled, *but* some other control group has it enabled.

Spinlock

The third configuration we define is the *Spinlock* configuration. The *Spinlock* configuration is a situation where one of the ancestors' control groups of a process has CFS bandwidth enabled, using the old spinlock approach. This is an unmodified kernel.

Atomic

The last configuration we define is the *Atomic* configuration. The *Atomic* configuration is a situation where one of the ancestors' control groups of a process has CFS bandwidth enabled, using our proposed implementation using an atomic variable, as described in Section 5.2.1.

5.3.2 Sysbench Threads Benchmark

As seen in Figure 5.1, the increase in runtime is most noticeable for the runs using a slice length of $1\mu s$. As seen in Figure 5.2, where the result for *Spinlock* $1\mu s$ is ommitted, we see that the resulsts for all the other configurations, except those using a slice length of $1\mu s$, are fairly similar. We clearly see that for a slice length of, $1\mu s$, our proposed atomic verions perform much better.

5.4 Discussion

As seen in the results, we see a significant decrease in overhead when using our proposed changes, altough it is only noticeable when using a slice length of $1\mu s$. However, the meadian value goes from about 210 seconds when using the old spinlock, to about 28 seconds when using our proposed atomic implementation. This clearly show that the reduction in cross-cpu communication is working.
As described in Appendix C, the test was executed on a physical machine with

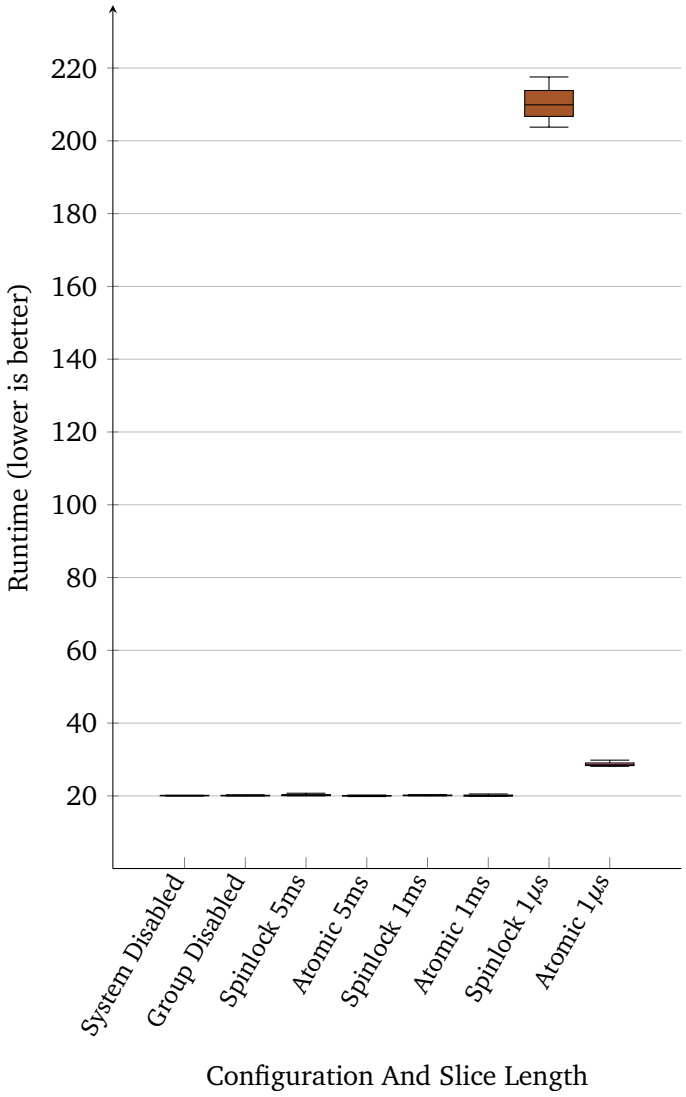


Figure 5.1: Sysbench Threads Benchmark results. 38 threads on 38 logical Linux CPUs, 500 000 events and 10 locks. Values based on 30 executive runs.

two physical sockets and two NUMA[1] nodes. Our test clearly show that the overhead is greatly reduced in such a system by using atomic instructions. For slice lengths of both $1ms$ and $5ms$ there is a slight decrease in runtime for the *atomic* implementation, but the differentce is too small to conclude, since the difference from the baseline is negligible.

By defining the overhead in this example as the time spent over the baseline, we see that for a slice length of $1\mu s$, the respective overhead is 190 seconds for the spinlock implementation and 8 seconds for the atomic implementation. That

[1]NUMA: Non-uniform memory access

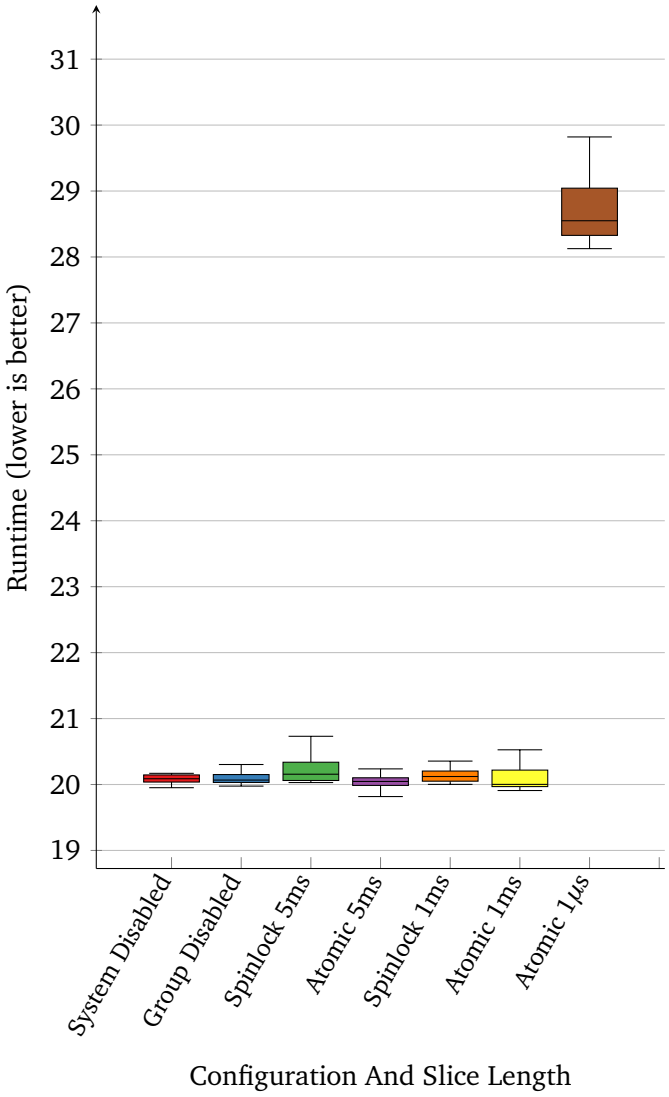


Figure 5.2: Sysbench Threads Benchmark results. 38 threads on 38 logical Linux CPUs, 500 000 events and 10 locks. Values based on 30 executive runs. Spinlock 1μs omitted.

means that our proposed technique reduce the overhead of CFS bandwidth control by more than 95% in our experiment.

Since our tests are hammering the scheduler with work, the measurements are not representative of the overhead users will see in real life. A slice length of 1μs is also not something used in production. It is also important to note that the runtime measurements are just a proxy value for the overhead. However, it is the overall runtime we are most interested in, since that is the most important value for the end-users.

Chapter 6

Mitigating Fairness Issues

In this chapter, we will look at how the fairness issues in the Linux kernel can cause serious performance issues, and how they are related to CFS bandwidth throttling. We will investigate their impact, and propose several kernel changes in order to mitigate these issues. We see that in some circumstances, these issues lead to a fairness skew that can cause programs to take more than **170** times their predicted execution time. Our mitigation techniques solves this issues, and removes this fairness skew. Depending on the configuration, we also see that these issues can cause even more significant performance decreases, depending on the number of logical Linux CPUs, and the depth of the control group hierarchy.

6.1 Introduction

As described in Section 3.1.1, the calculation used in order to ensure global fairness between control groups is a complex and expensive procedure. Ensuring this calculation is as precise and performant as possible is essential, as long as it stays correct. It is also crucial that the load is adequately decayed to ensure that the old load will not interfere with fairness at a later stage.

This is getting increasingly more complex when dealing with the load on local pools throttled by CFS bandwidth.

During our research on how throttling interferes with fairness, we identified three separate bugs related to fairness. All bugs cause severe fairness problems, in the long run, often resulting in a complete stall where processes essentially stop. A common factor for all of the discovered issues is that they are all hard to discover without looking explicitly, since there is no simple way to discover them from userspace. Without using introspection into the kernel, the only way to spot the issues is to inspect the debug file `/proc/sched_debug`, and use that information to infer that there is a problem.

6.2 Discovered Issues

We divide these issues into three separate but related issues, each causing severe fairness problems.

6.2.1 Process Moved Into A New Control Group

The first of the discovered fairness issues we found during this research is a fairness issue caused by a new process moved into a control group, and then moved to another logical Linux CPU before it was enqueued for execution. As seen in Section 3.1.1, the CFS scheduler keeps an internal list of active scheduling entities in order to make sure they are adequately decayed. In order to keep this list updated, scheduler entities are appended whenever one of their descendant processes is enqueued and ready for execution. When moving a process into a control group, the corresponding scheduler entity was not appended to this list.

However, when a process is moved into a group, its load is automatically added to the scheduling entity it is connected to, and then later propagated to the full CFS red-black tree. This is done to make the prioritization between processes and groups correct. In turn, this resulted in a situation where the scheduling entity was not added to the global list, and its load was never properly decayed.

As seen in Section 2.4.2, weighted group scheduling uses the total load of a control group when it calculates how much it should advance the virtual runtime of a scheduling entity. This later results in a fairness skew. Depending on the structure of the control group hierarchy, the type of load, and the internal weighting

between sibling control groups in the control group hierarchy, the effects of this could be pretty substantial.

A way of reproducing can be found in Appendix B.1.

6.2.2 CPU Affinity Triggered Process Move While Throttled

In the same way, as seen in Section 6.2.1, when a throttled process is moved from one logical Linux CPU to another, with a corresponding move from one scheduling entity to another, an issue appears. Such a move should, in theory, work fine, but as described in Section 3.1.1, the scheduling entities connected to throttled local pools are being removed from this list. During unthrottle of a local pool, only scheduling entities with at least one process ready to run are added back to the list, resulting in a stalled load.

A way of reproducing can be found in Appendix B.2.

6.2.3 Load Not Properly Decayed

In a particular situation where the load of a scheduling entity was almost decayed, but not wholly, the scheduler entity was removed while still contributing to the overall load of the control group. In the same way, as seen in Section 6.2.1 and Section 6.2.2, the scheduler entity ended up in a stalled state while still containing load.

Overall, this issue is a result of calculation issues in the PELT hierarchy described in Section 2.4.3, that later results in the scheduling entity being removed from the list.

A way of reproducing can be found in Appendix B.3.

6.3 Implementation

The implementation of fixes for all the three issues mentioned in Section 6.2 can be found in Appendix A.3. All fixes ensure that the list of scheduling entities is updated correctly, ensuring that these issues never occur.

6.4 Results

In the nature of all these CFS bandwidth issues, the impact varies quite extensively depending on many factors. As seen in Section 2.4.2, predicting the desired behavior of the CFS scheduler is straightforward, especially while running on a single logical Linux CPU.

Since all the issues share the same nature, we use the *Sysbench CPU benchmark* as described in Appendix C.2 and *Sysbench Threads benchmark* as described in

Appendix C.3 to study the impact of the *Process Moved Into A New Control Group* issue described in Section 6.2.1. We use the reproduction script found in Appendix B.1 to create a control group hierarchy where all processes are limited to run on a single logical Linux CPU. We then replace one of the *stress* processes located in the */slice/cg-1/sub* control group with our benchmark. The control group hierarchy should therefore converge toward what is seen in Figure 6.1. The synthetic load, the *stress* processes, are described in Appendix C.4.

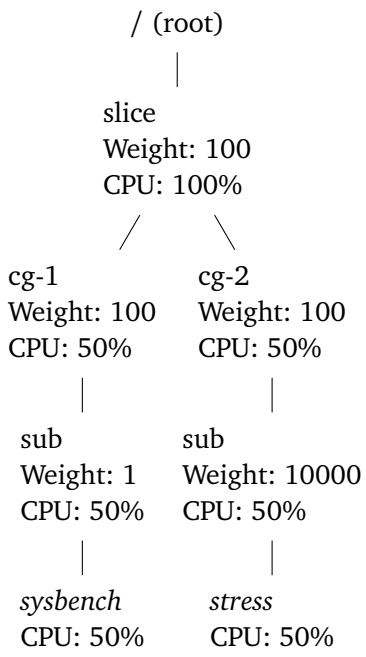


Figure 6.1: Cgroup hierarchy together with internal weighting, and predicted CPU load during congestion.

6.4.1 Sysbench CPU Benchmark

Table 6.1: Result of Sysbench CPU Benchmark with 5000 events. (lower duration is better)

Name	Runtime	Increase Compared To Prediction
Baseline without load	3.67s	N/A
Predicted	7.33s	-
Unpatched kernel	**1255.8s**	**17 032%**
Patched kernel	7.33s	0%

The test results can be found in Table 6.1. The runtime increase compared to the predicted runtime of the benchmark is more than 17 000%.

6.4.2 Sysbench Threads Benchmark

Table 6.2: Result of Sysbench Threads Benchmark with 5000 events. (lower duration is better)

Name	Benchmark duration	Increase Compared To Prediction
Baseline without load	2.35s	N/A
Predicted	4.70s	-
Unpatched kernel	**456.6**	**9 615%**
Patched kernel	4.85s	3.19%

The test results can be found in Table 6.2. The runtime increase compared to the predicted runtime of the benchmark is more than 9 615%.

6.5 Discussion

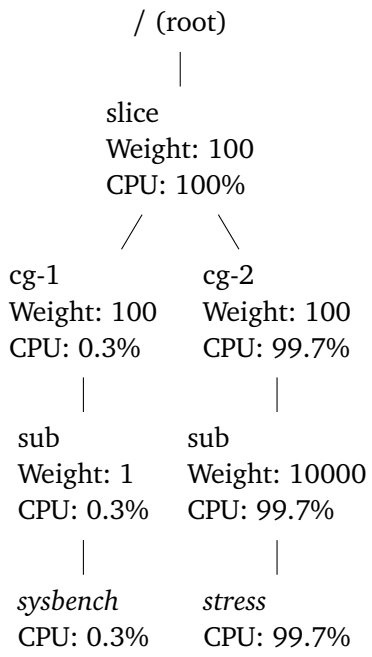


Figure 6.2: Cgroup hierarchy together with internal weighting, and actual CPU load during the test.

Upon investigation, the actual distribution of CPU time between the stress- and the benchmark processes is fairly skewed, as seen in Figure 6.2.

Even though the impact of these issues is varying, we see that it, in some cases, can be astronomical. For the *Sysbench CPU Benchmark* with 5000 events, which should take a bit over 7 seconds, ends up taking more than 1200. Such fairness skew will make low priority batch jobs take CPU time from other latency-

sensitive workloads, essentially stalling them. The same applies to user-facing processes on mobile devices, where a simple operation could end up taking almost 100 times longer than expected. As the nature of this fairness issue, we see that this priority skew will increase linearly with the amount of logical Linux CPUs that have an undecayed load, together with the depth of the control group hierarchy. This issues could therefore possibly make programs run `100x`, `1000x` or even `10 000x` slower than what they are supposed to do. This effectively means we can tinker with the control group hierarchy by starting processes a specific way, causing the issue to keep growing.

Another important aspect is that these issues affect real workloads, since all Linux users are effectively using these control group APIs. As these issues can be easily reproduced on both desktops and servers, and with most of the modern container runtimes used, this is affecting many end-users without them knowing. Kubernetes [13] highly depends on these container runtimes and on the low-level APIs provided by the kernel in general, making it highly plausible that this is affecting all modern cloud providers and users of them. Although these issues only surface during resource congestion, they will also result in preemption of critical and latency-sensitive processes even in times without much congestion. In the worst case, these issues could cause stalling of processes, essentially disallowing them to run, potentially causing fatal consequences.

Part III

Discussion And Conclusion

Chapter 7

Discussion

Our proposed changes have proven to reduce overhead and improve overall performance. In this chapter, we will discuss how the results relate to our research goals and how they affect end users of the Linux kernel.

7.1 Research Goals

The research goals for this research were all about improving performance, reducing overhead, and ensuring fairness in the Linux scheduler, as thoroughly described in Section 1.1. Although the Linux scheduler is a complex piece of code, we have shown that there are possible improvements and the fact that there are bugs, as in all other software projects.

As represented in the result section of Chapter 4, our proposed changes for reducing the unnecessary throttling and reaching our research goal **G1** showed promising results. Using the default configuration, a CFS bandwidth period of $100ms$, our test removed all unnecessary throttling when we used the same amount of processes as the CFS quota was equivalent to. This reduced the variance in runtime significantly, and decreased the runtime of about 3% on an otherwise busy system.

In Chapter 5 we proposed a patch for reducing the CFS bandwidth overhead, as described in research goal **G2**. In our synthetic scheduler benchmark, our testing showed a 95% decrease in the overhead of CFS bandwidth control, possibly improving both performance and latency, especially on big servers.

Last, research goal **G3**, about the effect all this gives on fairness. As seen with **G1** in Chapter 4, throttling does indeed interfere with the fairness on otherwise busy systems, as described in Section 3.3.3. As seen in Chapter 6, the impact of the fairness issues found in the scheduler is also reasonably huge.

7.2 End User Implications

All our proposed changes have improved overall performance without introducing new APIs that need to be enabled or configured. This means that all the benefits can be achieved right away, by all end users of Linux. This is especially true for the fairness improvements described in Chapter 6, since that applies to all users of Linux. Even though measuring the final impact is impossible, all users will benefit from it. The CFS bandwidth control improvements will also benefit servers serving requests for end-users, improving end-to-end latencies on congested systems.

7.3 Limitations and Critics

There are some limitations with the testing throughout this theses, especially when it comes to actual real-world workloads. All benchmarks used are synthetic programs designed to test a specific thing, different from real-world workloads. Our tests do highlight the change in scheduler performance based on our metrics, although it might happen that some other parts will suffer more.

Another vital thing to note is that the scheduler does have a massive set of customizable knobs that can be tweaked depending on the characteristics of the workload. As described in Appendix C, all our research has been done with the kernel configuration `CONFIG_HZ=250`, making the scheduler tick every 4 milliseconds. This is the default in most Linux distributions, while others use `=100`, `=1000` or `=300`. As described in Section 2.4, the scheduler is tickless, meaning the scheduling itself does not rely on these ticks. However, some accounting is done on these ticks, including the CFS bandwidth accounting. This means that the unnecessary throttling, in theory, could be less severe with a frequency of $1000Hz$ and more severe when using $100Hz$. Another important thing is that many defaults depend on the amount of logical Linux CPUs on the system. This includes a set of parameters in the load balancing algorithm, potentially making the effect of throttling on busy systems with many logical Linux CPUs even more severe than we have seen.

Finally, the number of systems tested on is pretty small. The system's topology is especially important when measuring the overhead, and more testing of variuous hardware would be beneficial. We have also only been testing on 64-bit x86 machines, whereas things might behave drastically different on modern arm servers, as an example. This especially applies to the use of atomic instructions used to reduce the CFS bandwidth overhead since the implementation of atomic instructions varies significantly between architectures. Since the runtime variable is a 64-bit integer, it would also perform better on 64-bit systems since 32-bit architectures without special 64-bit instructions might need custom and expensive locking logic to atomically update 64 bits of data.

Chapter 8

Conclusion

The overall goal of this research was to reduce the overhead and performance impact of CFS bandwidth control based on real-world use cases where these effects come to play. To mitigate these issues, we proposed a set of techniques to improve these performance aspects of the Linux CFS scheduler. These techniques reduce unnecessary throttling and lower the overhead of the CFS bandwidth control functionality in general. Our analysis shows that with the most common kernel configurations, the runtime of a program could be reduced by about **3%** during CPU congestion, with possibly more on bigger machines. Our synthetic benchmark also showed that we were able to reduce the overhead of CFS bandwidth control by about **95%**. Our techniques, and their implementations, will be posted to the Linux kernel mailing lists to gather feedback, and eventually get them merged into the mainline.

Incidentally, we also discovered a set of issues related to fairness in the Linux scheduler during our research. In both theory and practice, these issues could stall programs on busy systems, slowing them down by orders of magnitude. Our analysis shows that we were able to generate real-world situations where a program took about **170** times longer to execute than expected. We have created a set of techniques in order to mitigate the issues, and posted them as patches on the Linux kernel mailing lists. Some of our patches have already made it to the mainline kernel, and have been backported to all the officially supported versions. Others are making their way, together with fixes for other related issues discovered because of our work. ■

Bibliography

[1] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune and J. Wilkes, 'Large-scale cluster management at google with borg,' in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, Bordeaux, France: Association for Computing Machinery, 2015, ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. [Online]. Available: https://doi.org/10.1145/2741948.2741964.

[2] T. K. Authors, *Kubernetes - production-grade container orchestration*, https://kubernetes.io/, (Accessed on 04/29/2021).

[3] *Avoid setting cpu limits for guaranteed pods | issue #51135 " kubernetes/kubernetes*, https://github.com/kubernetes/kubernetes/issues/51135, (Accessed on 05/06/2021).

[4] *Disable cpu quota(use only cpuset) for pod guaranteed | issue #70585 | kubernetes/kubernetes*, https://github.com/kubernetes/kubernetes/issues/70585, (Accessed on 05/06/2021).

[5] *Operating system family / linux | top500*, https://www.top500.org/statistics/details/osfam/1/, (Accessed on 06/08/2021).

[6] L. Torvalds, *What would you like to see most in minix?* https://groups.google.com/g/comp.os.minix/c/dlNtH7RRrGA/m/SwRavCzVE7gJ?pli=1, (Accessed on 04/13/2021), 1991.

[7] *Linus torvalds' love-hate relationship with the gpl | zdnet*, https://www.zdnet.com/article/linus-torvaldss-love-hate-relationship-with-the-gpl/, (Accessed on 05/07/2021).

[8] *Control group v2 the linux kernel documentation*, https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html, (Accessed on 05/07/2021).

[9] L. J. Guibas and R. Sedgewick, 'A dichromatic framework for balanced trees,' in *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3.

[10] *Per-entity load tracking [lwn.net]*, https://lwn.net/Articles/531853/, (Accessed on 05/11/2021).

[11] *[rfc/rft patch] sched: Automated per tty task groups - mike galbraith*, https://lore.kernel.org/lkml/1287479765.9920.9.camel@marge.simson.net/, (Accessed on 05/13/2021).

[12] P. Turner, B. B. Rao and N. Rao, 'Cpu bandwidth control for cfs,' in *Proceedings of the Linux Symposium*, 2010, pp. 245–254. [Online]. Available: http://www.linuxsymposium.org/LS_2010_Proceedings_Draft.pdf.

[13] *Assign cpu resources to containers and pods | kubernetes*, https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/#cpu-units, (Accessed on 05/26/2021).

[14] *Runtime options with memory, cpus, and gpus | docker documentation*, https://docs.docker.com/config/containers/resource_constraints/#configure-the-default-cfs-scheduler, (Accessed on 05/26/2021).

[15] *[patch 2/2] sched/fair: Fix o(nr_cgroups) in load balance path - vincent guittot*, https://lore.kernel.org/lkml/1549469662-13614-3-git-send-email-vincent.guittot@linaro.org/, (Accessed on 05/13/2021).

[16] *[patch 1/2] sched/fair: Optimization of update_blocked_averages() - vincent guittot*, https://lore.kernel.org/lkml/1549469662-13614-2-git-send-email-vincent.guittot@linaro.org/, (Accessed on 05/13/2021).

[17] *[patch v2 1/2] sched/fair: Fix bandwidth timer clock drift condition - xunlei pang*, https://lore.kernel.org/lkml/20180620101834.24455-1-xlpang@linux.alibaba.com/, (Accessed on 05/26/2021).

[18] *[patch v6 1/1] sched/fair: Fix low cpu usage with high throttling by removing expiration of cpu-local slices - dave chiluk*, https://lore.kernel.org/linux-doc/1563900266-19734-2-git-send-email-chiluk+linux@indeed.com/, (Accessed on 05/26/2021).

[19] *Cfs bandwidth control the linux kernel documentation*, https://www.kernel.org/doc/html/v5.11/scheduler/sched-bwc.html, (Accessed on 05/13/2021).

[20] D. Ghatrehsamani, C. Denninnart, J. Bacik and M. Amini Salehi, 'The art of cpu-pinning: Evaluating and improving the performance of virtualization and containerization platforms,' in *49th International Conference on Parallel Processing - ICPP*, ser. ICPP '20, Edmonton, AB, Canada: Association for Computing Machinery, 2020, ISBN: 9781450388160. DOI: 10.1145/3404397.3404442. [Online]. Available: https://doi.org/10.1145/3404397.3404442.

[21] *Semantics and behavior of atomic and bitmask operations the linux kernel documentation*, https://www.kernel.org/doc/html/v5.12/staging/index.html#atomic-types, (Accessed on 05/31/2021).

[22] *Https://www.kernel.org/doc/documentation/static-keys.txt*, https://www.kernel.org/doc/Documentation/static-keys.txt, (Accessed on 06/08/2021).

[23] *[PATCH 1/1] sched/fair: Fix unfairness caused by missing load decay - Odin Ugedal*, https://lore.kernel.org/lkml/20210425080902.11854-2-odin@uged.al/, (Accessed on 05/31/2021).

[24] *[patch 2/3] sched/fair: Correctly insert cfs_rq's to list on unthrottle - odin ugedal*, https://lore.kernel.org/lkml/20210518125202.78658-3-odin@uged.al/, (Accessed on 05/31/2021).

[25] *[patch 1/3] sched/fair: Add tg_load_contrib cfs_rq decay checking - odin ugedal*, https://lore.kernel.org/lkml/20210518125202.78658-2-odin@uged.al/, (Accessed on 05/31/2021).

[26] *[patch 0/1] sched/fair: Fix unfairness caused by missing load decay - odin ugedal*, https://lore.kernel.org/lkml/20210425080902.11854-1-odin@uged.al/, (Accessed on 06/11/2021).

[27] *[patch 0/3] sched/fair: Fix load decay issues related to throttling - odin ugedal*, https://lore.kernel.org/lkml/20210518125202.78658-1-odin@uged.al/, (Accessed on 06/11/2021).

[28] *Arch linux*, https://archlinux.org/, (Accessed on 05/31/2021).

[29] *Control group v2, cpuset controller the linux kernel documentation*, https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#cpuset, (Accessed on 05/31/2021).

[30] *Tsenart/vegeta: Http load testing tool and library. it's over 9000!* https://github.com/tsenart/vegeta, (Accessed on 05/31/2021).

[31] *Hdrhistogram by giltene*, https://hdrhistogram.github.io/HdrHistogram/, (Accessed on 05/31/2021).

[32] *Akopytov/sysbench: Scriptable database and system performance benchmark*, https://github.com/akopytov/sysbench, (Accessed on 05/31/2021).

[33] A. Waterland, *Stress project page*, https://web.archive.org/web/20131123082734/https://people.seas.harvard.edu/~apw/stress/, (Accessed on 04/14/2021), 2013.

Appendices

Appendix A

Proposed patches to the Linux kernel

A.1 Proposed Linux Kernel Patches For Avoiding Throttling

A.1.1 Accounting Slack

Code listing A.1: sched/fair: Add cfs bandwidth slush fund label

```
From 835ff9d7ef2c94ceb9f3d18dae09dbe3b6b9be94 Mon Sep 17 00:00:00 2001
From: Odin Ugedal <odin@uged.al>
Date: Mon, 26 Apr 2021 11:24:31 +0200
Subject: [PATCH] sched/fair: Add cfs bandwidth slush fund

Signed-off-by: Odin Ugedal <odin@uged.al>
---
 kernel/sched/fair.c  | 22 ++++++++++++++++++---
 kernel/sched/sched.h |  2 ++
 2 files changed, 21 insertions(+), 3 deletions(-)

diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
index cbcb770f696c..71c9f9e8b158 100644
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -4623,8 +4623,10 @@ static inline u64 sched_cfs_bandwidth_slice(void)
  */
 void __refill_cfs_bandwidth_runtime(struct cfs_bandwidth *cfs_b)
 {
-   if (cfs_b->quota != RUNTIME_INF)
+   if (cfs_b->quota != RUNTIME_INF){
+       cfs_b->runtime_slush_fund = cfs_b->runtime;
        cfs_b->runtime = cfs_b->quota;
+   }
 }

 static inline struct cfs_bandwidth *tg_cfs_bandwidth(struct task_group *tg)
@@ -4636,10 +4638,11 @@ static inline struct cfs_bandwidth *tg_cfs_bandwidth(struct
    ↪  task_group *tg)
```

```
 static int __assign_cfs_rq_runtime(struct cfs_bandwidth *cfs_b,
                   struct cfs_rq *cfs_rq, u64 target_runtime)
 {
-    u64 min_amount, amount = 0;
+    u64 min_amount, slush_amount, amount = 0;

     lockdep_assert_held(&cfs_b->lock);

+
     /* note: this is a positive sum as runtime_remaining <= 0 */
     min_amount = target_runtime - cfs_rq->runtime_remaining;

@@ -4647,6 +4650,14 @@ static int __assign_cfs_rq_runtime(struct cfs_bandwidth *
     ↪ cfs_b,
         amount = min_amount;
     else {
         start_cfs_bandwidth(cfs_b);
+        if (cfs_rq->nr_periods != cfs_b->nr_periods) {
+            slush_amount = min(cfs_b->runtime_slush_fund,
+                 (u64) -cfs_rq->runtime_remaining);
+            cfs_b->runtime_slush_fund -= slush_amount;
+            cfs_rq->runtime_remaining += slush_amount;
+            cfs_rq->nr_periods = cfs_b->nr_periods;
+            min_amount = target_runtime - cfs_rq->runtime_remaining;
+        }

         if (cfs_b->runtime > 0) {
             amount = min(cfs_b->runtime, min_amount);
@@ -5291,13 +5302,18 @@ static void init_cfs_rq_runtime(struct cfs_rq *cfs_rq)

 void start_cfs_bandwidth(struct cfs_bandwidth *cfs_b)
 {
+    int overrun;
     lockdep_assert_held(&cfs_b->lock);

     if (cfs_b->period_active)
         return;

     cfs_b->period_active = 1;
-    hrtimer_forward_now(&cfs_b->period_timer, cfs_b->period);
+    overrun = hrtimer_forward_now(&cfs_b->period_timer, cfs_b->period);
+    if (overrun){
+        __refill_cfs_bandwidth_runtime(cfs_b);
+        cfs_b->nr_periods++;
+    }
     hrtimer_start_expires(&cfs_b->period_timer, HRTIMER_MODE_ABS_PINNED);
 }

diff --git a/kernel/sched/sched.h b/kernel/sched/sched.h
index 10a1522b1e30..15debeeeeb6b 100644
--- a/kernel/sched/sched.h
+++ b/kernel/sched/sched.h
@@ -357,6 +357,7 @@ struct cfs_bandwidth {
     ktime_t         period;
     u64         quota;
     u64         runtime;
+    u64         runtime_slush_fund;
     s64         hierarchical_quota;

     u8          idle;
```

```
@@ -587,6 +588,7 @@ struct cfs_rq {
 #ifdef CONFIG_CFS_BANDWIDTH
     int         runtime_enabled;
     s64         runtime_remaining;
+    s64         nr_periods;

     u64         throttled_clock;
     u64         throttled_clock_task;
--
2.31.1
```

A.1.2 CFS bandwidth Config

Code listing A.2: sched/fair: Make CFS bandwidth updates with same values a NOP

```
From cc6f7b5a0b5236e7a346093bc754214d39a98582 Mon Sep 17 00:00:00 2001
From: Odin Ugedal <odin@uged.al>
Date: Sun, 6 Jun 2021 14:24:38 +0200
Subject: [PATCH] sched/fair: Make CFS bandwidth updates with same values a NOP

Signed-off-by: Odin Ugedal <odin@uged.al>
---
 kernel/sched/core.c | 5 +++++
 1 file changed, 5 insertions(+)

diff --git a/kernel/sched/core.c b/kernel/sched/core.c
index 98191218d891..6c26b31d0635 100644
--- a/kernel/sched/core.c
+++ b/kernel/sched/core.c
@@ -8982,6 +8982,11 @@ static int tg_set_cfs_bandwidth(struct task_group *tg, u64
    ↪ period, u64 quota)
     if (quota != RUNTIME_INF && quota > max_cfs_runtime)
         return -EINVAL;

+    /* Ignore update in case both quota and period stays the same */
+    if (quota == cfs_b->quota && period == ktime_to_ns(cfs_b->period)){
+        return ret;
+    }
+
     /*
      * Prevent race between setting of cfs_rq->runtime_enabled and
      * unthrottle_offline_cfs_rqs().
--
2.31.1
```

A.2 Proposed Linux Kernel Patches Reducing CFS Bandwidth overhead

A.2.1 Replacing Global Pool Spinlock

Code listing A.3: sched/fair: Add atomic runtime handling

```

From 62aa35bd5f85d9b300546671c591d23a516722a9 Mon Sep 17 00:00:00 2001
```

```
From: Odin Ugedal <odin@uged.al>
Date: Tue, 27 Apr 2021 09:51:20 +0200
Subject: [PATCH] sched/fair: Add atomic runtime handling

Signed-off-by: Odin Ugedal <odin@uged.al>
---
 kernel/sched/fair.c  | 96 ++++++++++++++++++++-------------------------
 kernel/sched/sched.h |  4 +-
 2 files changed, 42 insertions(+), 58 deletions(-)

diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
index 794c2cb945f8..8509bb875d93 100644
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -4605,7 +4605,8 @@ static inline u64 sched_cfs_bandwidth_slice(void)
 void __refill_cfs_bandwidth_runtime(struct cfs_bandwidth *cfs_b)
 {
     if (cfs_b->quota != RUNTIME_INF)
-        cfs_b->runtime = cfs_b->quota;
+        atomic64_set(&cfs_b->runtime, cfs_b->quota);
+
 }

 static inline struct cfs_bandwidth *tg_cfs_bandwidth(struct task_group *tg)
@@ -4617,41 +4618,27 @@ static inline struct cfs_bandwidth *tg_cfs_bandwidth(struct
     ↪  task_group *tg)
 static int __assign_cfs_rq_runtime(struct cfs_bandwidth *cfs_b,
                    struct cfs_rq *cfs_rq, u64 target_runtime)
 {
-    u64 min_amount, amount = 0;
-
-    lockdep_assert_held(&cfs_b->lock);
+    s64 old, amount;
+    int ret = 1;

-    /* note: this is a positive sum as runtime_remaining <= 0 */
-    min_amount = target_runtime - cfs_rq->runtime_remaining;
+    amount = (target_runtime - cfs_rq->runtime_remaining);
+    old = atomic64_fetch_sub(amount, &cfs_b->runtime);

-    if (cfs_b->quota == RUNTIME_INF)
-        amount = min_amount;
-    else {
-        start_cfs_bandwidth(cfs_b);
-
-        if (cfs_b->runtime > 0) {
-            amount = min(cfs_b->runtime, min_amount);
-            cfs_b->runtime -= amount;
-            cfs_b->idle = 0;
-        }
+    if (unlikely(old < amount)){
+        amount = max(old, (s64) 0);
+        ret = (cfs_rq->runtime_remaining + amount) > 0;
     }

     cfs_rq->runtime_remaining += amount;
-
-    return cfs_rq->runtime_remaining > 0;
+    return ret;
 }
```

```
62
63  /* returns 0 on failure to allocate runtime */
64  static int assign_cfs_rq_runtime(struct cfs_rq *cfs_rq)
65  {
66      struct cfs_bandwidth *cfs_b = tg_cfs_bandwidth(cfs_rq->tg);
67  -    int ret;
68
69  -    raw_spin_lock(&cfs_b->lock);
70  -    ret = __assign_cfs_rq_runtime(cfs_b, cfs_rq, sched_cfs_bandwidth_slice());
71  -    raw_spin_unlock(&cfs_b->lock);
72  -
73  -    return ret;
74  +    return __assign_cfs_rq_runtime(cfs_b, cfs_rq, sched_cfs_bandwidth_slice());
75  }
76
77  static void __account_cfs_rq_runtime(struct cfs_rq *cfs_rq, u64 delta_exec)
78 @@ -4911,7 +4898,7 @@ void unthrottle_cfs_rq(struct cfs_rq *cfs_rq)
79  static void distribute_cfs_runtime(struct cfs_bandwidth *cfs_b)
80  {
81      struct cfs_rq *cfs_rq;
82  -    u64 runtime, remaining = 1;
83  +    u64 remaining = 1;
84
85      rcu_read_lock();
86      list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,
87 @@ -4926,19 +4913,11 @@ static void distribute_cfs_runtime(struct cfs_bandwidth *
        ↪ cfs_b)
88          /* By the above check, this should never be true */
89          SCHED_WARN_ON(cfs_rq->runtime_remaining > 0);
90
91  -        raw_spin_lock(&cfs_b->lock);
92  -        runtime = -cfs_rq->runtime_remaining + 1;
93  -        if (runtime > cfs_b->runtime)
94  -            runtime = cfs_b->runtime;
95  -        cfs_b->runtime -= runtime;
96  -        remaining = cfs_b->runtime;
97  -        raw_spin_unlock(&cfs_b->lock);
98  -
99  -        cfs_rq->runtime_remaining += runtime;
100 -
101 -        /* we check whether we're throttled above */
102 -        if (cfs_rq->runtime_remaining > 0)
103 +        if (__assign_cfs_rq_runtime(cfs_b, cfs_rq, 1)){
104              unthrottle_cfs_rq(cfs_rq);
105 +        } else {
106 +            remaining = 0;
107 +        }
108
109  next:
110          rq_unlock_irqrestore(rq, &rf);
111 @@ -4977,7 +4956,7 @@ static int do_sched_cfs_period_timer(struct cfs_bandwidth *
        ↪ cfs_b, int overrun, u
112
113      if (!throttled) {
114          /* mark as potentially idle for the upcoming period */
115  -        cfs_b->idle = 1;
116  +        cfs_b->idle = 0;
117          return 0;
118      }
119
```

```
@@ -4987,11 +4966,8 @@ static int do_sched_cfs_period_timer(struct cfs_bandwidth *
    ↪ cfs_b, int overrun, u
     /*
      * This check is repeated as we release cfs_b->lock while we unthrottle.
      */
-    while (throttled && cfs_b->runtime > 0) {
-        raw_spin_unlock_irqrestore(&cfs_b->lock, flags);
-        /* we can't nest cfs_b->lock while distributing bandwidth */
+    while (throttled && atomic64_read(&cfs_b->runtime) > 0) {
         distribute_cfs_runtime(cfs_b);
-        raw_spin_lock_irqsave(&cfs_b->lock, flags);

         throttled = !list_empty(&cfs_b->throttled_cfs_rq);
     }
@@ -5068,19 +5044,24 @@ static void __return_cfs_rq_runtime(struct cfs_rq *cfs_rq)
     if (slack_runtime <= 0)
         return;

-    raw_spin_lock(&cfs_b->lock);
-    if (cfs_b->quota != RUNTIME_INF) {
-        cfs_b->runtime += slack_runtime;
-
-        /* we are under rq->lock, defer unthrottling using a timer */
-        if (cfs_b->runtime > sched_cfs_bandwidth_slice() &&
-            !list_empty(&cfs_b->throttled_cfs_rq))
-            start_cfs_slack_bandwidth(cfs_b);
-    }
-    raw_spin_unlock(&cfs_b->lock);

-    /* even if it's not valid for return we don't want to try again */
+    atomic64_add(slack_runtime, &cfs_b->runtime);
     cfs_rq->runtime_remaining -= slack_runtime;
+
+    /*
+     * Atomic runtime does not support the slack timer currently,
+     * since it would require locking the global cfs_b. The plan is
+     * to lower overhead to make this unneccecary.
+     *
+     *    if (cfs_b->quota != RUNTIME_INF) {
+     *        cfs_b->runtime += slack_runtime;
+     *
+     *        // we are under rq->lock, defer unthrottling using a timer
+     *        if (cfs_b->runtime > sched_cfs_bandwidth_slice() &&
+     *            !list_empty(&cfs_b->throttled_cfs_rq))
+     *            start_cfs_slack_bandwidth(cfs_b);
+     *    }
+     */
 }

 static __always_inline void return_cfs_rq_runtime(struct cfs_rq *cfs_rq)
@@ -5100,7 +5081,7 @@ static __always_inline void return_cfs_rq_runtime(struct
    ↪ cfs_rq *cfs_rq)
  */
 static void do_sched_cfs_slack_timer(struct cfs_bandwidth *cfs_b)
 {
-    u64 runtime = 0, slice = sched_cfs_bandwidth_slice();
+    u64 runtime = 0;//, slice = sched_cfs_bandwidth_slice();
     unsigned long flags;

     /* confirm we're still not at a refresh boundary */
```

```
@@ -5112,8 +5093,9 @@ static void do_sched_cfs_slack_timer(struct cfs_bandwidth *
    ↪ cfs_b)
         return;
     }

-    if (cfs_b->quota != RUNTIME_INF && cfs_b->runtime > slice)
-        runtime = cfs_b->runtime;
+    // Not in use
+    //if (cfs_b->quota != RUNTIME_INF && cfs_b->runtime > slice)
+    //    runtime = cfs_b->runtime;

     raw_spin_unlock_irqrestore(&cfs_b->lock, flags);

@@ -5252,7 +5234,7 @@ static enum hrtimer_restart sched_cfs_period_timer(struct
    ↪ hrtimer *timer)
 void init_cfs_bandwidth(struct cfs_bandwidth *cfs_b)
 {
     raw_spin_lock_init(&cfs_b->lock);
-    cfs_b->runtime = 0;
+    atomic64_set(&cfs_b->runtime, 0);
     cfs_b->quota = RUNTIME_INF;
     cfs_b->period = ns_to_ktime(default_cfs_period());

diff --git a/kernel/sched/sched.h b/kernel/sched/sched.h
index 10a1522b1e30..9cd93366f076 100644
--- a/kernel/sched/sched.h
+++ b/kernel/sched/sched.h
@@ -35,6 +35,7 @@

 #include <uapi/linux/sched/types.h>

+#include <linux/atomic.h>
 #include <linux/binfmts.h>
 #include <linux/blkdev.h>
 #include <linux/compat.h>
@@ -356,7 +357,6 @@ struct cfs_bandwidth {
     raw_spinlock_t        lock;
     ktime_t            period;
     u64            quota;
-    u64            runtime;
     s64            hierarchical_quota;

     u8            idle;
@@ -370,6 +370,8 @@ struct cfs_bandwidth {
     int            nr_periods;
     int            nr_throttled;
     u64            throttled_time;
+
+    atomic64_t        runtime ____cacheline_aligned;
 #endif
 };

--
2.31.1
```

A.3 Proposed scheduler fairness patches

A.3.1 Process Moved Into A New Control Group

Proposed patch to fix *Process Moved Into A New Control Group* issue. Sent to the Linux kernel mailing list [23] for further discussion.

Code listing A.4: sched/fair: Fix unfairness caused by missing load decay

```
From 026a845ef164787921b12085d35b41749a4ce6da Mon Sep 17 00:00:00 2001
From: Odin Ugedal <odin@uged.al>
Subject: sched/fair: Fix unfairness caused by missing load decay
Date: Sun, 25 Apr 2021 10:09:02 +0200

This fixes an issue where old load on a cfs_rq is not properly decayed,
resulting in strange behavior where fairness can decrease drastically.
Real workloads with equally weighted control groups have ended up
getting a respective 99% and 1%(!!) of cpu time.

When an idle task is attached to a cfs_rq by attaching a pid to a cgroup,
the old load of the task is attached to the new cfs_rq and sched_entity by
attach_entity_cfs_rq. If the task is then moved to another cpu (and
therefore cfs_rq) before being enqueued/woken up, the load will be moved
to cfs_rq->removed from the sched_entity. Such a move will happen when
enforcing a cpuset on the task (eg. via a cgroup) that force it to move.

The load will however not be removed from the task_group itself, making
it look like there is a constant load on that cfs_rq. This causes the
vruntime of tasks on other sibling cfs_rq's to increase faster than they
are supposed to; causing severe fairness issues. If no other task is
started on the given cfs_rq, and due to the cpuset it would not happen,
this load would never be properly unloaded. With this patch the load
will be properly removed inside update_blocked_averages. This also
applies to tasks moved to the fair scheduling class and moved to another
cpu, and this path will also fix that. For fork, the entity is queued
right away, so this problem does not affect that.

For a simple cgroup hierarchy (as seen below) with two equally weighted
groups, that in theory should get 50/50 of cpu time each, it often leads
to a load of 60/40 or 70/30.

parent/
  cg-1/
    cpu.weight: 100
    cpuset.cpus: 1
  cg-2/
    cpu.weight: 100
    cpuset.cpus: 1

If the hierarchy is deeper (as seen below), while keeping cg-1 and cg-2
equally weighted, they should still get a 50/50 balance of cpu time.
This however sometimes results in a balance of 10/90 or 1/99(!!) between
the task groups.

$ ps u -C stress
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       18568  1.1  0.0   3684   100 pts/12   R+   13:36   0:00 stress --cpu 1
root       18580 99.3  0.0   3684   100 pts/12   R+   13:36   0:09 stress --cpu 1

```

```
parent/
  cg-1/
    cpu.weight: 100
    sub-group/
      cpu.weight: 1
      cpuset.cpus: 1
  cg-2/
    cpu.weight: 100
    sub-group/
      cpu.weight: 10000
      cpuset.cpus: 1

This can be reproduced by attaching an idle process to a cgroup and
moving it to a given cpuset before it wakes up. The issue is evident in
many (if not most) container runtimes, and has been reproduced
with both crun and runc (and therefore docker and all its "derivatives"),
and with both cgroup v1 and v2.

Fixes: 3d30544f0212 ("sched/fair: Apply more PELT fixes")
Signed-off-by: Odin Ugedal <odin@uged.al>
---
 kernel/sched/fair.c | 13 ++++++++++++
 1 file changed, 13 insertions(+)

diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
index 794c2cb945f8..ad7556f99b4a 100644
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -10916,6 +10916,19 @@ static void attach_task_cfs_rq(struct task_struct *p)

     if (!vruntime_normalized(p))
         se->vruntime += cfs_rq->min_vruntime;
+
+    /*
+     * Make sure the attached load will decay properly
+     * in case the task is moved to another cpu before
+     * being queued.
+     */
+    if (!task_on_rq_queued(p)) {
+        for_each_sched_entity(se) {
+            if (se->on_rq)
+                break;
+            list_add_leaf_cfs_rq(cfs_rq_of(se));
+        }
+    }
 }

 static void switched_from_fair(struct rq *rq, struct task_struct *p)
--
2.31.1
```

A.4 CPU Affinity Triggered Process Move While Throttled

Proposed patch to fix *CPU Affinity Triggered Process Move While Throttled* issue. Sent to the Linux kernel mailing list [24] for further discussion.

Code listing A.5: sched/fair: Correctly insert cfs_rq's to list on unthrottle

```
From 026a845ef164787921b12085d35b41749a4ce6da Mon Sep 17 00:00:00 2001
From: Odin Ugedal <odin@uged.al>
Subject: sched/fair: Correctly insert cfs_rq's to list on unthrottle
Date: Tue, 18 May 2021 14:52:01 +0200

This fixes an issue where fairness is decreased since cfs_rq's can
end up not being decayed properly. For two sibling control groups with
the same priority, this can often lead to a load ratio of 99/1 (!!).

This happen because when a cfs_rq is throttled, all the descendant cfs_rq's
will be removed from the leaf list. When they initial cfs_rq is
unthrottled, it will currently only re add descendant cfs_rq's if they
have one or more entities enqueued. This is not a perfect heuristic.

This fix change this behavior to save what cfs_rq's was removed from the
list, and readds them properly on unthrottle.

Can often lead to sutiations like this for equally weighted control
groups:

$ ps u -C stress
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       10009 88.8  0.0   3676   100 pts/1    R+   11:04   0:13 stress --cpu 1
root       10023  3.0  0.0   3676   104 pts/1    R+   11:04   0:00 stress --cpu 1

Fixes: 31bc6aeaab1d ("sched/fair: Optimize update_blocked_averages()")
Signed-off-by: Odin Ugedal <odin@uged.al>
---
 kernel/sched/fair.c  | 11 ++++++-----
 kernel/sched/sched.h |  1 +
 2 files changed, 7 insertions(+), 5 deletions(-)

diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
index ceda53c2a87a..e7423d658389 100644
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -376,7 +376,8 @@ static inline bool list_add_leaf_cfs_rq(struct cfs_rq *cfs_rq)
     return false;
 }

-static inline void list_del_leaf_cfs_rq(struct cfs_rq *cfs_rq)
+/* Returns 1 if cfs_rq was present in the list and removed */
+static inline bool list_del_leaf_cfs_rq(struct cfs_rq *cfs_rq)
 {
     if (cfs_rq->on_list) {
         struct rq *rq = rq_of(cfs_rq);
@@ -393,7 +394,9 @@ static inline void list_del_leaf_cfs_rq(struct cfs_rq *cfs_rq)

         list_del_rcu(&cfs_rq->leaf_cfs_rq_list);
         cfs_rq->on_list = 0;
+        return 1;
     }
+    return 0;
 }

 static inline void assert_list_leaf_cfs_rq(struct rq *rq)
@@ -4742,9 +4745,7 @@ static int tg_unthrottle_up(struct task_group *tg, void *data
    ↪ )
     if (!cfs_rq->throttle_count) {
```

```
        cfs_rq->throttled_clock_task_time += rq_clock_task(rq) -
                         cfs_rq->throttled_clock_task;
-
-       /* Add cfs_rq with already running entity in the list */
-       if (cfs_rq->nr_running >= 1)
+       if (cfs_rq->insert_on_unthrottle)
            list_add_leaf_cfs_rq(cfs_rq);
    }

@@ -4759,7 +4760,7 @@ static int tg_throttle_down(struct task_group *tg, void *data
    /* group is entering throttled state, stop time */
    if (!cfs_rq->throttle_count) {
        cfs_rq->throttled_clock_task = rq_clock_task(rq);
-       list_del_leaf_cfs_rq(cfs_rq);
+       cfs_rq->insert_on_unthrottle = list_del_leaf_cfs_rq(cfs_rq);
    }
    cfs_rq->throttle_count++;

diff --git a/kernel/sched/sched.h b/kernel/sched/sched.h
index a189bec13729..12a707d99ee6 100644
--- a/kernel/sched/sched.h
+++ b/kernel/sched/sched.h
@@ -602,6 +602,7 @@ struct cfs_rq {
    u64         throttled_clock_task_time;
    int         throttled;
    int         throttle_count;
+   int         insert_on_unthrottle;
    struct list_head    throttled_list;
 #endif /* CONFIG_CFS_BANDWIDTH */
 #endif /* CONFIG_FAIR_GROUP_SCHED */
--
2.31.1
```

A.5 Load Not Properly Decayed

Proposed patch to fix *Load Not Properly Decayed* issue. Sent to the Linux kernel mailing list [25] for further discussion.

Code listing A.6: sched/fair: Add tg_load_contrib cfs_rq decay checking

```
From: Odin Ugedal <odin@uged.al>
Subject: sched/fair: Add tg_load_contrib cfs_rq decay checking
Date: Tue, 18 May 2021 14:52:00 +0200

Make sure cfs_rq does not contribute to task group load avg when
checking if it is decayed. Due to how the pelt tracking works,
the divider can result in a situation where:

cfs_rq->avg.load_sum = 0
cfs_rq->avg.load_avg = 4
cfs_rq->avg.tg_load_avg_contrib = 4

If pelt tracking in this case does not cross a period, there is no
"change" in load_sum, and therefore load_avg is not recalculated, and
keeps its value.

```

```
If this cfs_rq is then removed from the leaf list, it results in a
situation where the load is never removed from the tg. If that happen,
the fiarness is permanently skewed.

Fixes: 039ae8bcf7a5 ("sched/fair: Fix O(nr_cgroups) in the load balancing path")
Signed-off-by: Odin Ugedal <odin@uged.al>
---
 kernel/sched/fair.c | 3 +++
 1 file changed, 3 insertions(+)

diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
index 3248e24a90b0..ceda53c2a87a 100644
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -8004,6 +8004,9 @@ static inline bool cfs_rq_is_decayed(struct cfs_rq *cfs_rq)
     if (cfs_rq->avg.runnable_sum)
         return false;

+   if (cfs_rq->tg_load_avg_contrib)
+       return false;
+
     return true;
 }

--
2.31.1
```

Appendix B

Reproduction Scripts For CFS Fairness Issues

All reproduction scripts mimic a container runtime spawning two containers pinned to the same logical Linux CPU, with the same CFS weight. The expected results is that both processes get 50% of the CPU time.

B.1 Process Moved Into A New Control Group

Script for reproducing *Process Moved Into A New Control Group*. Also shared on the Linux kernel mailing lists [26].

Code listing B.1: Reproduction script for Process Moved Into A New Control Group issue

```
CGROUP=/sys/fs/cgroup/slice
CGROUP_TMP=/sys/fs/cgroup/tmp
CGROUP_OLD=/sys/fs/cgroup"$(cat␣/proc/self/cgroup␣|␣cut␣-c4-)"
CPU=1

function run_sandbox {
  local CG="$1"
  local SUB_WEIGHT="$2"
  local CMD="$3"

  local PIPE=$(mktemp -u)
  mkfifo "$PIPE"
  sh -c "read␣<␣$PIPE␣;␣exec␣$CMD" &
  local TASK="$!"
  sleep .01
  mkdir -p "$CG"/sub
  tee "$CG"/cgroup.subtree_control <<< "+cpuset␣+cpu"
  tee "$CG"/sub/cgroup.procs <<< "$TASK"
  tee "$CG"/sub/cpuset.cpus <<< "$CPU"
  tee "$CG"/sub/cpu.weight <<< "$SUB_WEIGHT"

  sleep .1
```

```
  tee "$PIPE" <<< sandox_done
  rm "$PIPE"
}

mkdir -p "$CGROUP"
mkdir -p "$CGROUP_TMP"
tee /sys/fs/cgroup/cgroup.subtree_control <<< "+cpuset +cpu"
tee "$CGROUP"/cgroup.subtree_control <<< "+cpuset +cpu"

echo $$ | tee "$CGROUP_TMP"/cgroup.procs
tee "$CGROUP_TMP"/cpuset.cpus <<< "0"

run_sandbox "$CGROUP/cg-1" 1    "stress --cpu 1"

cat "$CGROUP/cg-1/sub/cgroup.procs" | xargs kill
tee "$CGROUP_TMP"/cpuset.cpus <<< "2"
run_sandbox "$CGROUP/cg-1" 1    "stress --cpu 1"

cat "$CGROUP/cg-1/sub/cgroup.procs" | xargs kill
tee "$CGROUP_TMP"/cpuset.cpus <<< "2"
run_sandbox "$CGROUP/cg-1" 1    "stress --cpu 1"

tee "$CGROUP_TMP"/cpuset.cpus <<< "1"
run_sandbox "$CGROUP/cg-2" 10000 "stress --cpu 1"

read
echo $$ | tee "$CGROUP_OLD"/cgroup.procs
killall stress
sleep .1
rmdir /sys/fs/cgroup/slice/{cg-1{/sub,},cg-2{/sub,},} /sys/fs/cgroup/tmp/
```

B.2 CPU Affinity Triggered Process Move While Throttled

Script for reproducing *Process Moved Into A New Control Group*. Also shared on the Linux kernel mailing lists [27].

Code listing B.2: Reproduction script for CPU affinity triggered process move while throttled

```
CGROUP=/sys/fs/cgroup/slice
TMP_CG=/sys/fs/cgroup/tmp
OLD_CG=/sys/fs/cgroup"$(cat /proc/self/cgroup | cut -c4-)"
function run_sandbox {
  local CG="$1"
  local LCPU="$2"
  local SHARES="$3"
  local CMD="$4"

  local PIPE="$(mktemp -u)"
  mkfifo "$PIPE"
  sh -c "read <$PIPE ;exec $CMD" &
  local TASK="$!"
  mkdir -p "$CG/sub"
  tee "$CG"/cgroup.subtree_control <<< "+cpuset +cpu"
  tee "$CG"/sub/cpuset.cpus <<< "$LCPU"
  tee "$CG"/sub/cgroup.procs <<< "$TASK"
  tee "$CG"/sub/cpu.weight <<< "$SHARES"
```

```
19
20    sleep .01
21    tee "$PIPE" <<< sandox_done
22    rm "$PIPE"
23  }
24
25  mkdir -p "$CGROUP"
26  mkdir -p "$TMP_CG"
27  tee "$CGROUP"/cgroup.subtree_control <<< "+cpuset +cpu"
28
29  echo $$ | tee "$TMP_CG"/cgroup.procs
30  tee "$TMP_CG"/cpuset.cpus <<< "0"
31  sleep .1
32
33  tee "$CGROUP"/cpu.max <<< "1000 4000"
34
35  run_sandbox "$CGROUP/cg-0" "0" 10000 "stress --cpu 1"
36  run_sandbox "$CGROUP/cg-3" "3" 1 "stress --cpu 1"
37
38  sleep 2
39  tee "$CGROUP"/cg-0/sub/cpuset.cpus <<< "3"
40
41  tee "$CGROUP"/cpu.max <<< "max"
42
43  read
44  killall stress
45  sleep .2
46  echo $$ | tee "$OLD_CG"/cgroup.procs
47  rmdir "$TMP_CG" /sys/fs/cgroup/slice/{cg-{0,3}{/sub,},}
```

B.3 Load Not Properly Decayed

Script for reproducing *Process Moved Into A New Control Group*. Also shared on the Linux kernel mailing lists [27].

Code listing B.3: Reproduction script for load not properly decayed

```
1   CGROUP=/sys/fs/cgroup/slice
2
3   function run_sandbox {
4     local CG="$1"
5     local LCPU="$2"
6     local SHARES="$3"
7     local CMD="$4"
8
9     local PIPE="$(mktemp -u)"
10    mkfifo "$PIPE"
11    sh -c "read < $PIPE ; exec $CMD" &
12    local TASK="$!"
13    mkdir -p "$CG/sub"
14    tee "$CG"/cgroup.subtree_control <<< "+cpuset +cpu"
15    tee "$CG"/sub/cgroup.procs <<< "$TASK"
16    tee "$CG"/sub/cpuset.cpus <<< "$LCPU"
17    tee "$CG"/sub/cpu.weight <<< "$SHARES"
18    tee "$CG"/cpu.max <<< "10000 100000"
19
20    sleep .1
```

```
  tee "$PIPE" <<< sandox_done
  rm "$PIPE"
}

mkdir -p "$CGROUP"
tee "$CGROUP"/cgroup.subtree_control <<< "+cpuset␣+cpu"

run_sandbox "$CGROUP/cg-1" "0" 100 "stress␣--cpu␣1"
run_sandbox "$CGROUP/cg-2" "3" 100 "stress␣--cpu␣1"
sleep 1.02
tee "$CGROUP"/cg-1/sub/cpuset.cpus <<< "1"
sleep 1.05
tee "$CGROUP"/cg-1/sub/cpuset.cpus <<< "2"
sleep 1.07
tee "$CGROUP"/cg-1/sub/cpuset.cpus <<< "3"

sleep 2

tee "$CGROUP"/cg-1/cpu.max <<< "max"
tee "$CGROUP"/cg-2/cpu.max <<< "max"

read # click enter to cleanup
killall stress
sleep .2
rmdir /sys/fs/cgroup/slice/{cg-{1,2}{/sub,},}
```

Appendix C

Benchmarks

In this appendix, we describe all the benchmarks used in this research.

All tests have been executed on a desktop machine running Arch Linux [28], containing an Intel® Core™i5-4670K with 4 non-SMT cores clocked at 4.0GHz, with a total of 24GiB of memory. The kernel version used is a self compiled Linux Kernel v5.12.0. All tests have been running with the kernel configuration parameter `CONFIG_HZ=250`.

Tests with 38 logical Linux CPUs have been executed on a server with two Intel Xeon Silver 4114 chips, each with a total of 20 SMT threads clocked at 2.2GHz, with a total of 128GiB of memory.

All benchmarks have also been executed on an otherwise idle system, unless otherwise specified, and on a set of exclusive logical Linux CPUs using the cpu-set control group controller [29]. During all our testing, except for testing the fairness issues themselves, we have used the patches in Appendix A.3, described in Chapter 6. We have done this in order to get proper results without fairness skews.

Together with the output of the benchmarks, we also inspect the CFS bandwidth metrics described in Section 2.5.1.

C.1 Webserver Latency Benchmark

For testing latency of a real world workload we use a simple webserver written in go, that does a finite amount of CPU intensive work during each request.

Code listing C.1: Simple go webserver

```
1 package main
2
3 import (
4     "fmt"
```

```
5     "log"
6     "net/http"
7 )
8
9
10 func main() {
11     maxLoad := 1000000
12
13     http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
14         for i := 1; i < maxLoad; i++ {
15             _ = i * i
16         }
17     })
18
19     fmt.Printf("Starting␣server␣at␣port␣8080\n")
20     if err := http.ListenAndServe(":8080", nil); err != nil {
21         log.Fatal(err)
22     }
23 }
```

For load generation, we use the tool *vegeta* [30], since it allows for setting a constant rate of requests sent to the webserver each seconds.

C.1.1 Test Parameters

- *Number of processes* : The number of OS level processes to use.
  - Each request will execute in a separate goroutine, a user level thread implementation. Go has an environment variable *GOMAXPROCS* that can control the amount of operating system threads those gorutines run. Defaults to the amount of available Logical Linux CPUs.
- *Request rate*: The number of requests per second
- *Duration*: The duration to run the test for

C.1.2 Test Output

The output of the *vegeta* tool is a latency histogram that can be visualized with HdrHistograms [31].

C.2 Sysbench CPU Benchmark

The Sysbench [32] CPU test is a simple CPU benchmark. It consists of a CPU bound algorithm that calculates prime numbers, and is used as a long lived CPU bound benchmark.

C.2.1 Test Parameters

- *Threads*: The number of threads to run on
- *Events*: The number of events to run, defaults to 10000.

C.2.2 Test Output

The output is the time spent doing all the computations.

C.3 Sysbench Threads Benchmark

The Sysbench [32] threads test is a thread-based scheduler benchmark. It consists of a set of locks that is locked and unlocked multiple time at each iteration, and results in a high amount of short program executions and communication between threads. The test is used as a benchmark for measuring scheduler overhead.

C.3.1 Test Parameters

- *Threads*: The number of threads to run on
- *Events*: The number of events to run, defaults to 10000.
- *Locks*: The number locks to take each event, defaults to 8.

C.3.2 Test Output

The output is the time spent doing all the computations.

C.4 Synthetic background load

> 'stress' imposes certain types of compute stress on your system
>
> $ stress –help

For creating synthetic background load, we use the *stress* [33], a simple workload generator. Stress spawns as many threads as requested, with each just running a simple busy loop calculating the square root of a randomly created integers.

Odin Ugedal

Bandwidth Control And Fairness In The Linux Scheduler

NTNU
Norwegian University of
Science and Technology