Mitigating Unnecessary Throttling in Linux CFS Bandwidth Control

Odin Ugedal
NTNU, Norway
odin@ugedal.com

Rakesh Kumar
NTNU, Norway
rakesh.kumar@ntnu.no

***Abstract*—An operating system needs to fairly allocate shared hardware resources among different applications, and Linux uses Completely Fair Scheduler (CFS) to achieve this goal. To ensure fairness, CFS implements *bandwidth control* that sets the maximum limit on the resources that a process can use. Setting the upper limit helps CFS to discover badly behaving applications and hard cap them to limit the overall damage to the system and other applications, thereby improving fairness.**

We observe that, in an effort to ensure fairness, the bandwidth control can unnecessarily throttle processes which results in poor application performance. We investigate the root cause of this limitation and discover that the CPU runtime accounting mechanism of bandwidth control, which tracks if a process has reached it maximum allocated limit, is responsible for it. We also find that the overhead of fair resource allocation in bandwidth control can become significantly high due to the way it is implemented. We propose mechanisms to reduce throttling as well as the overhead. Our experimental results show that the proposed techniques are able to eliminate nearly all throttling, thus providing up to 12% performance gain. Also, our approach reduces the bandwidth control overhead by up to 24x.

***Index Terms*—System Software, Operating Systems, Linux, Process Scheduling, Process Throttling**

I. Introduction

One of the most fundamental tasks of an operating system is to arbitrate among applications, or processes, to provide them with access to shared hardware resources. To do so, it employs a scheduler that selects a subset of ready processes for execution on hardware based on a predetermined policy. Different scheduler policies target different metrics and are aimed at different environments. For example, a scheduling policy focusing on fairness aims to evenly distribute CPU time among all competing processes. A priority based scheduling policy, in contrast, prioritizes processes, say, based on their criticality, for example, in a real time system where a process might need to finish before a deadline.

Linux is one of the most popular operating systems that runs, in one form or other, on a wide variety of hardware ranging from warehouse scale computers to small embedded devices. Linux supports six scheduling policies with the default scheduler, called Completely Fair Scheduler (CFS), implementing a fairness oriented policy called SCHED_NORMAL. Historically, to ensure fairness, the hardware resources were distributed among processes using one of the four different metrics: weights, limits, protections, and allocations. However, irrespective of the metric used, the key idea in resource allocation was to set a minimum bound on resources that each process or a set of processes should get.

About a decade ago, Google proposed CFS Bandwidth Control [1] that changed the way the resources are allocated among processes. Instead of setting the *minimum* amount, CFS Bandwidth Control sets the *maximum* limit on resources that a process can use. Setting the maximum limit helps discover badly behaving applications and hard cap them to limit the overall damage to the system and other applications, thereby improving overall fairness. As a result, bandwidth control has not only become an integral part of all Linux Container runtimes but also of container orchestration tool Kubernetes [2], [3], Google's internal container orchestrator Borg [4], a well known container runtime Docker [5], etc.

Though CFS bandwidth control is excellent at providing fairness among processes, we observe that it might unnecessarily throttle processes which not only increases process execution time but also reduces overall throughput of the system. We find that the root cause of this throttling is the CPU runtime accounting mechanism of bandwidth control which is responsible for tracking if a process has reached its maximum allocated limit. Concretely, the bandwidth control allocates resources, including CPU runtime, in fixed time chunks, called *periods*. Due to the way runtime accounting is done, it is possible for a process to run slightly longer than its allocated runtime quota. However, when a process requests more runtime after its quota expires, any extra time that the process has already run for is adjusted before allocating more runtime to the process. We discover that if the extra runtime adjustment is not done during the same *period* in which the extra runtime was consumed, rather in the next *period*, the bandwidth control throttles the processes during the next *period* before they reach their allocated quota limit. Such throttling results in underutilization of resources and lowers the performance and throughput.

In addition to throttling, we also discover that the overhead of fair resource allocation in CFS bandwidth control can become significantly high. This overhead mainly stems from the spinlock used to guard the runtime available within a period. The overhead can be especially high in a system running large number of threads as the waiting time to acquire the spinlock increases with the number of waiting threads.

Based on our analysis of the root causes of the limitations of bandwidth control, we propose mechanisms to mitigate these bottlenecks while still preserving the fairness. To avoid throttling, we propose a mechanism that gives an illusion as if the extra runtime used by a process is accounted for in the same period in which it was consumed even if it is actually accounted for in the next period. As a result, the bandwidth control does not need to throttle the processes in the next period. To reduce the overhead of bandwidth control, we propose to use atomic variables to guard the runtime available in a period instead of spinlocks that are used in existing implementation. Overall, this paper makes the following key contributions:

- Identifying that the CFS Bandwidth Control can unnecessarily throttle processes, thus hurting performance.
- Identifying the root cause of throttling and bandwidth control overhead.
- Proposing mechanism to mitigate throttling and bandwidth control overhead.
- Evaluating our proposals and showing that the proposed throttling mitigation technique eliminates nearly all throttling, thus providing up to 12% performance benefit. Our proposal also reduces the bandwidth control overhead by up to 24x.

II. Background and Motivation

Over the last three decades, the Linux kernel has evolved from a small university hobby project to 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 datacenters. All of the world's top 500 supercomputers are also running Linux [6]. All this while still being a fully open source and community driven project where everyone can contribute.

A. Linux Scheduler

Like any operating system, the central piece of Linux that decides what process to execute is called a process scheduler. Linux supports six different scheduling policies, listed in Table I, 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 referred to as the *"Scheduler"* of a given system.

Linux scheduler uses a mechanism, called cgroup (control group), to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner. cgroup is largely composed of two parts - the core and the controllers. *cgroup core* is primarily responsible for hierarchically organizing processes, whereas *cgroup controller* is usually responsible for distributing a specific type of system resource along the hierarchy although there are utility controllers which serve purposes other than resource distribution.

[1]Named `SCHED_OTHER` in user space.

TABLE I: The different scheduling policies supported by Linux

Policy Name	Class	Description
SCHED_NORMAL[1]	fair	Normal time sharing scheduling
SCHED_FIFO	rt	First in-first out scheduling
SCHED_RR	rt	Round-robin scheduling
SCHED_BATCH	fair	SCHED_NORMAL, less preemption
SCHED_IDLE	idle	Low priority tasks
SCHED_DEADLINE	dl	Deadline scheduling

Resource Distribution models: Depending on the resources a cgroup 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.
- **Limits**: Sets the maximum amount of a given resource that a control the group can use. The limit is not tied to the actual availability of the resource, thus allowing 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 that only have a limited amount available, and cannot be overcommitted.

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

B. CFS Bandwidth Control

Bandwidth Control is a mechanism used by Linux CFS (Completely Fair Scheduler) to limit the amount of CPU time that a set of processes can use. It was designed by an engineering team at Google [1] to discover badly behaving applications and hard cap them to limit the overall damage to the system and other applications. The key difference with prior mechanisms is that while prior mechanisms set the *minimum* amount of CPU resources a cgroup should get, bandwidth control sets the *maximum* limit. Bandwidth control is also a part of the cpu cgroup controller.

Since its introduction in 2010, Bandwidth control has become an integral part of all Linux Container runtimes as it plays a vital role in controlling how much CPU time a container or application can use at a maximum. It is also used as the tool for limiting CPU usage by the container orchestration tool Kubernetes [3], and it is used directly by mapping container CPU limits to an equivalent CFS Bandwidth configuration. The same applies to the well known

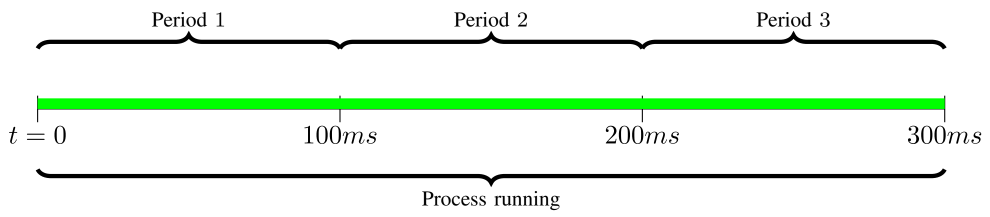
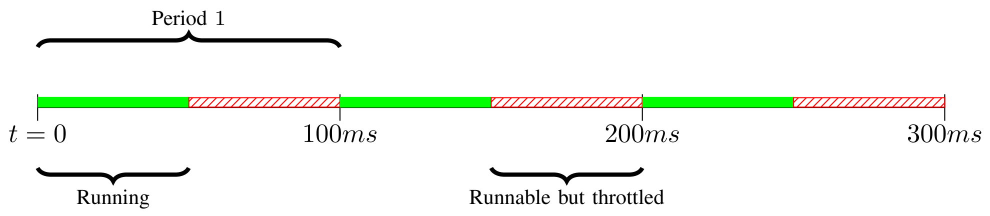
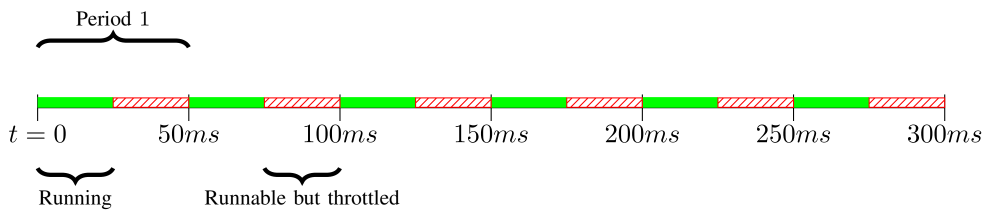


Fig. 1: Timeline of a cpu bound process running with a period of $100ms$ and a quota of $100ms$. The quota is the total CPU time that can be used per period.



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



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

Fig. 2: 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.

container runtime *Docker* [5]. CFS Bandwidth control is also mentioned as a critical aspect of Borg [4], the internal container orchestrator at Google.

1) Bandwidth Control API: The bandwidth control exposes two configuration parameter per cgroup;

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

An example with 100ms *quota* and 100ms *period* is shown in Figure 1. The ratio between the *quota* and the *period* can be seen as the number of logical CPUs a control group can continuously use without being throttled. For example, a period of, say, $100ms$ and a quota of $300ms$ 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, for example, a period of $100ms$ and a quota of $50ms$ can be viewed as the equivalent of half of a logical Linux CPU, as shown in Figure 2a. Another example for a half logical Linux CPU equivalence would be a control group with a period of $50ms$ and a quota of $25ms$, as seen in Figure 2b.

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 the periods where the control group has been throttled, i.e., 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 in throttled, i.e., stopped from executing due to using too much CPU time, summed across all logical Linux CPUs.

2) Enforcement: The Linux scheduler itself enforces the CFS Bandwidth control. 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, an integer protected by a spinlock, holds the runtime available for use. It 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. The global pools exist per control group.
- **The local pool**: Each logical Linux CPU has a local pool that is used to account for the runtime available and used. Such local pools exist per logical Linux CPU in each control group.

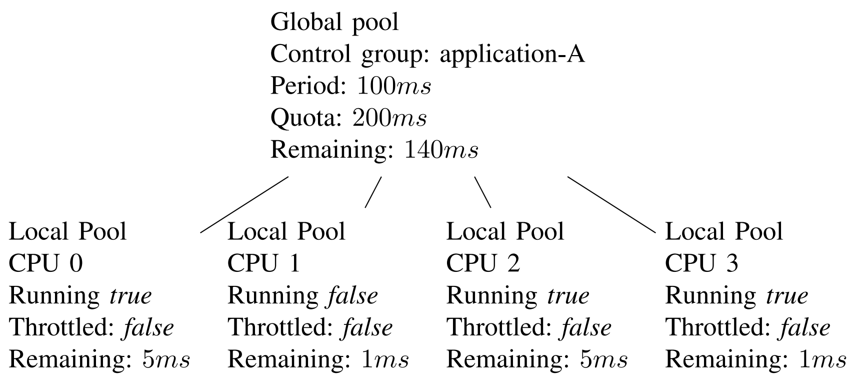


Fig. 3: Control group with the equivalent of 2 logical Linux CPUs set via CFS bandwidth, running on a system with 4 logical Linux CPUs.

When a process in a Bandwidth Control enabled cgroup has been executed for a given amount of time, that time is removed from the corresponding CPU local pool. This accounting happens either i) when the process is swapped out or ii) on scheduler ticks while the process is running. For example, Figure 3 shows the Global and Local Pool runtimes of a cgroup. Local Pool for logical Linux CPU 0 and CPU 2 have 5ms remaining while only 1 ms remains for CPU 1 and CPU 3. If a local pool runs out of its allocated runtime, the scheduler needs to acquire more runtime. To do so, it will try to acquire the lock protecting the global pool and then take an amount from it 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, depending on use, and defaults to $5ms$. Given this default slice value, the *Remaining* time in Figure 3 suggests that, the processes in CPU 0 and CPU 2 have not executed after the last refill while processes in CPU 1 and CPU 3 have executed for 4ms each.

When the local pool of a logical Linux CPU does not have any runtime remaining and it is unable to refill from the global pool, the Linux CPU will be *throttled*. The scheduler will then mark all the scheduling entities in the hierarchy under that logical CPU 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. In 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 their processes as ready to schedule again.

III. Limitations of Linux CFS Bandwidth Control

We observe that the Linux CFS Bandwidth Control mechanism has two major limitations: i) Unnecessary throttling and ii) high overhead.

A. Unnecessary Throttling

As seen in section II-B, as long as a local pool has a positive amount of runtime remaining, it is allowed to continue running one of its child processes. However, we observe that, due to the runtime accounting mechanism of Linux Bandwidth Control, local pools often end up with substantial amount of *negative* runtime available. Especially for highly threaded applications running simultaneously on multiple logical Linux CPUs, the accumulated negative runtime can become very significant compared to the total quota per period.

A local pool with negative runtime signifies that a logical Linux CPU has run for more time than it was originally allocated. When such a local pool refills from the global pool, it has to pay back its negative runtime to compensate for the extra runtime that is has already used. For example, with a $5ms$ slice, when a local pool with $-1ms$ runtime is refilled from the global pool, the $1ms$ is first detected from the global pool before refilling the local pool with $5ms$. If this refill happens during the same *period* in which the extra $1ms$ runtime was consumed, then the negative runtime is compensated correctly and no unnecessary throttling happens. However, if we have already entered into a new *period* at the point of this refill, then the debt is paid back using the quota from the new period, even though the extra runtime was used in the previous period. Consequently, the new period sees a reduced quota which often leads to processes being throttled at the end of the period, even though they have used less runtime in the new period than the quota.

Next, we discuss the sources of negative runtimes and how they lead to unnecessary throttling, in details.

1) Unnecessary throttling due to negative runtimes: The two most important parameters for runtime accounting, and responsible for negative runtimes, are: i) scheduler tick interval and ii) slice length, i.e., maximum runtime in local pool. The scheduler tick interval, also called a *jiffy*, is important because the runtime accounting happens only on scheduler ticks, in addition to process swapping. Linux supports $1ms$, $3.33ms$, $4ms$, and $10ms$ scheduler tick intervals. If a local pool has less runtime remaining than the scheduler tick interval, running a process from such a pool can result in negative runtime in the local pool. For example, consider a $4ms$ jiffy and a local pool with $1ms$ remaining runtime. Running a process from

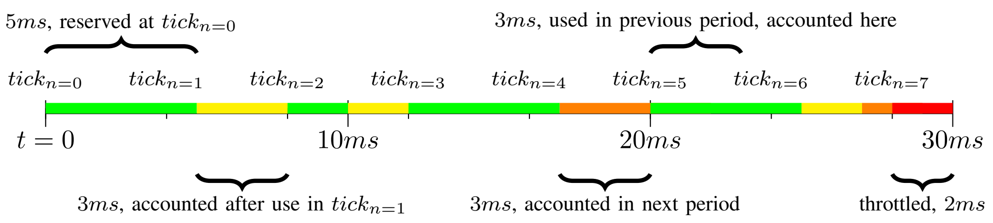

$5ms$, reserved at $tick_{n=0}$
$3ms$, used in previous period, accounted here
$tick_{n=0}$
$tick_{n=1}$
$tick_{n=2}$
$tick_{n=3}$
$tick_{n=4}$
$tick_{n=5}$
$tick_{n=6}$
$tick_{n=7}$
$t = 0$
$10ms$
$20ms$
$30ms$
$3ms$, accounted after use in $tick_{n=1}$
$3ms$, accounted in next period
throttled, $2ms$

Fig. 4: Timeline of runtime accounting on a cpu bound process, with a bandwidth period of $10ms$ and quota of $10ms$, with slice length of $5ms$ and jiffy length of $4ms$. The yellow portion of the timeline represents negative runtime compensated in the same period, orange represents negative runtime compensated in the next period, while red represents throttled period.

this local pool can result in negative runtime depending on when the last scheduler tick arrived. If the process is scheduled at tick, the next tick will arrive in $4ms$ and that is when the bandwidth control will check if there is still runtime remaining in the local pool to continue executing the process. However, at the next tick, the process would already have executed for $3ms$ extra time as the pool has only $1ms$ remaining. Consequently, the local pool will be left with $-3ms$.

The other essential parameter for runtime accounting is slice length, i.e., the amount of runtime a local pool tries to get when refilling from the global pool. The slice length is a compromise between increased accounting overhead and fine-grained consumption [7]. A large slice length reduces communication between the local and global pools because local pools would need fewer refills, thereby reducing accounting overhead. More importantly, fewer refills result in smaller accumulated negative runtime because a local pool's remaining runtime can become negative only close to refills when the remaining runtime is less than scheduler tick interval. On the other hand, large slice lengths also increase the likelihood 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 in Linux is $5ms$.

To better understand how negative runtimes lead to throttling, Figure 4 presents an example of runtime accounting of one process executing over three periods. The example uses a default slice length of $5ms$ and a scheduler tick interval of 4ms. Further, both period and global quota are $10ms$ each, i.e., after a period of $10ms$, the global pool is refilled with $10ms$ runtime. As the figure shows, the first scheduler tick, $tick_{n=0}$, arrives at t=0. At this point, the local pool gets $5ms$, i.e. slice length, runtime from the global pool which is also left with $5ms$ of remaining runtime. At the next scheduler tick, $tick_{n=1}$ at t=4ms, the local pool still has 1ms remaining runtime, so the bandwidth control keeps executing the process. However, as the next tick, $tick_{n=2}$, comes at t=8ms, the local pool at this point has accumulated $-3ms$ of runtime as it has run for $4ms$ since the last tick whereas it had only $1ms$ of runtime available. The local pool is then refilled from the global pool. However, before the refill, the $-3ms$ of runtime is first compensated and the global pool is left with $2ms$. As a result, the local pool gets refilled with only $2ms$.

At $10ms$ mark, the second period starts and the global pool is refilled with 10ms; however, the local pool remains unchanged at this point. The next scheduler tick, $tick_{n=3}$, arrives at t=12ms and the local pool has accumulated $-2ms$ runtime at this point as it was refilled with only $2ms$ at the last scheduler tick. Consequently, the $-2ms$ runtime is compensated from the global pool before filling the local pool with $5ms$ runtime, thus leaving the global pool with 3ms remaining runtime. At the next scheduler tick, $tick_{n=4}$ at t=16ms, the local pool still has $1ms$ runtime reaming, so the process keeps executing until the next tick at t=20ms and the local pool accumulates $-3ms$ by then. At t=20ms, the third period starts, the global pool is refilled with $10ms$, and the $-3ms$ of the local pool are compensated. Notice that, in the prior two cases of negative runtime compensation, the negative runtime was compensated from the global pool of the same period in which the negative runtime was accumulated. However, at t=20ms, the negative runtime was accumulated in second period whereas it is compensated from the global pool of the third period. Therefore, it leads to throttling towards the end of the period.

After compensating the $-3ms$ runtime at t=20ms, the local pool is refilled with $5ms$, thus the global pool is left with $2ms$ runtime. At the next scheduler tick, $tick_{n=6}$, the local pool still has $1ms$ remaining runtime, thus the process continues to execute until $tick_{n=7}$. At this point, the local pool has again accumulated $-3ms$ runtime. However, the global pool has only $2ms$ runtime remaining. Therefore, the local pool cannot be refilled and the process is throttled until the end of this period. Notice that the process has run only for $8ms$ in the third period instead of full $10ms$ of global quota. However, it has been throttled for $2ms$ because the negative runtime accumulated in the second period is compensated from the global quota of the third period.

In this simple example, the negative runtimes might be avoided by making the slice length a multiple of scheduler tick interval. However, in practice, the scheduling environment is much more complex and such a solution does not always work. For example process terminations and new process starts are not always aligned with scheduler ticks. Therefore, if a process starts between scheduler ticks, it's local pool will still end up accumulating negative runtime despite the slice length being a multiple of scheduler tick interval. The other reasons include

TABLE II: Baseline results for Sysbench CPU Benchmark with 300 000 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

each local pool potentially having a hierarchy of processes to schedule instead of just one, the quota to period ratio being fractional, and so on. Also, the runtime accounting is not the only factor in choosing scheduler tick frequency. As a result, making slice length a multiple of scheduler tick interval is not sufficient to eliminate negative runtimes.

2) Unnecessary throttling due to updating Bandwidth Control configuration: Another source of throttling is when the bandwidth control is configured from userspace. When a userspace application updates the CFS bandwidth configuration, the controller will reset all local pools to zero and refill the global pool even though the new values are the same as the old values. 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.

3) Performance loss due to throttling: Each time a local pool gets throttled, its processes will have to wait before starting executing again. This effectively means that if a process gets throttled for, say, $30ms$, it will take at least $30ms$ longer to finish. To investigate the amount of throttling in CPU intensive applications, i.e., how many times one or more local pools are throttled and for how long, we use the *Sysbench CPU benchmark* [8].

Table II shows the number of periods where processes were throttled and the total time spent while throttled for different values of period. As the table shows, the number of throttled periods and the throttling times increase as we reduce the bandwidth control period while maintaining the CFS bandwidth quota to period ratio, the number of running processes, and logical Linux CPUs. With unlimited quota, i.e. when bandwidth control is disabled, we do not see any throttling as expected. With a 100ms period, we observe throttling in about 6.8% of periods. As we reduce the interval to 10ms, we observe throttling in about 49% of periods. To better understand the impact of throttling on performance, we also evaluate the amount of time spend throttled in addition to the number of periods experiencing throttling. As the table shows, the time spent throttled increases from 94ms with 100ms period to about 14 seconds with 10ms period. These results show that throttling is a severe performance bottleneck as about 18.6% of the execution time is spent throttled with 10ms period. Therefore, any technique to reduce this throttling can provide significant performance benefits.

B. Scheduler Overhead

Linux bandwidth control uses a global spinlock to protect the global pool. This spinlock is one of the most significant factors that affects the overhead introduced by bandwidth control. This is because the spinlock has to be acquired each time the global pool is modified, for example when the global pool is refilled at the beginning of a period or when local pools are refilled from the global pool. The overhead increases with the number of logical Linux CPUs running processes below the same control group with bandwidth control enabled.

```
void refill_global_pool(cfs_global_pool global_pool)
{
    global_pool->runtime = global_pool->quota;
}
```

Listing 1: Original global pool refill logic.

Although spinlocks are quite frequently used in kernel code, they have some overhead. However, as long as there are no (or few) other users waiting for the spinlock, the overhead is negligible. The major bottleneck occurs when there are many users, i.e. logical Linux CPUs in our case, trying to acquire the same lock simultaneously; something potentially happening multiple thousand times per second in Linux bandwidth control.

IV. Mitigating the Limitations

This section proposes different solutions to mitigate the bandwidth control limitations discussed in the previous section. We first introduce two mechanisms to avoid throttling and then discuss a technique to reduce the bandwidth control overhead.

A. Mitigating unnecessary throttling

We propose the following two mechanisms to mitigate the unnecessary throttling:

1) Slush fund for compensating negative runtimes: As discussed in Section III-A1, throttling occurs when the negative runtime of a local pool in one period is compensated from the global pool of the next period. We observe that in the majority of cases the negative runtime can simply be compensated from the global pool of the period in which the negative runtime is accumulated as the global pool has sufficient runtime available at the end of the period. The reason it is compensated from the global pool of the next period is that once we enter the next period we lose the information of whether any runtime was left in the previous period or not. This is because, as the code listing 1 shows, the remaining runtime quota from the previous period is discarded during a refill. We define this runtime as the runtime *lost* on quota refilling.

```
void refill_global_pool(cfs_global_pool global_pool)
{
    global_pool->slush_fund = global_pool->runtime;
    global_pool->runtime = global_pool->quota;
    global_pool->period_nr += 1;
}
```

Listing 2: Proposed global pool refill logic.

```
int refill_local_pool_with_slush(cfs_global_pool
    global_pool, cfs_local_pool local_pool)
{
 if (global_pool->period_nr > local_pool->period_nr){
     int refill = min(global_pool->slush_fund,
                   -local_pool->remaining_runtime);
     global_pool->slush_fund -= refill;
     local_pool->remaining_runtime += refill;

     local_pool->period_nr = global_pool->period_nr;
 }
 return refill_local_pool(global_pool, local_pool);
}
```

Listing 3: Local pool refill using slush fund.

To mitigate this issue, we propose to account for negative accumulated runtime in local pools from the global pool of the same period in which negative runtime was accumulated, if possible. To do so, we introduce a new attribute to the global pools, called `slush fund`. During refill of the global pool, we save the remaining global pool runtime of previous period to the `slush_fund`, before refilling the global pool again, as seen in listing 2.

We also update the local pool refill mechanism accordingly. On a local pool refill, we use this slush fund to account for the negative accumulated runtime we might see on the first refill in a new period, as shown in listing 3.

As this technique compensates the negative runtime using slush fund instead of the global pool of the new period, the new period does not *lose* it's runtime and, hence, does not need to throttle the processes.

2) Modified CFS Bandwidth Update Logic: As discussed in Section III-A2, updates to bandwidth control configuration can also lead to throttling. To avoid such throttling, we propose to modify the configuration update logic to ensure that no configuration updates are done unless strictly necessary. In order to keep this simple, we first verify whether or not the old and new configuration parameters are same. And if they turn out to be the same we convert the update to a *no operation* so that it doesn't interfere with the normal operation.

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 are implemented 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 for the quota will result in a quota refill with the new quota and

```
int refill_local_pool(cfs_global_pool global_pool,
    cfs_local_pool local_pool)
{
    int target_runtime = slice_length;
    int amount = (target_runtime
                 - cfs_rq->remaining_runtime);
    int old = atomic_fetch_and_subtract(amount,
                        &global_pool->runtime);

    if (old < amount){
        if (old <= 0){
            amount = 0;
        } else {
            amount = old;
        }
    }
    local_pool->remaining_runtime += amount;
    return local_pool->remaining_runtime > 0;
}
```

Listing 4: Atomic implementation refill_local_pool.

a reset of all the local pools.

B. Reducing bandwidth control overhead

As seen in section III-B, the global spinlock used to protect the global pool is one of the most significant factors contributing to the bandwidth control overhead. We observe that most processor 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 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 spinlocks as shown in listing 4.

Linux has a wrapper for atomic variables via the *atomic_t* and *atomic64_t* types [9], together with implementations for each supported architecture. Architectures without native support for atomic variables fall back to transparently using a spinlock, i.e., there is no performance penalties for systems without such instructions. In order to use these types, we introduce an atomic 64 bits integer on the global pool. This replaces the old non-atomic version.

V. Evaluation

To evaluate our proposals, we use a desktop machine running Arch Linux [10] on an Intel® Core™i5-4670K with 4 non-SMT cores clocked at 4.0GHz, with a total of 24GB of memory. The kernel version used is a self compiled Linux kernel, with version v5.12.0. All experiments are performed on an otherwise idle system and on a set of exclusive logical Linux CPUs using the cpuset cgroup controller [11]. The experiments for evaluating the bandwidth control overhead are performed on a server with two Intel Xeon Silver 4114, each with a total of 20 SMT threads, clocked at 2.2GHz, and with a total of 128GB of memory.

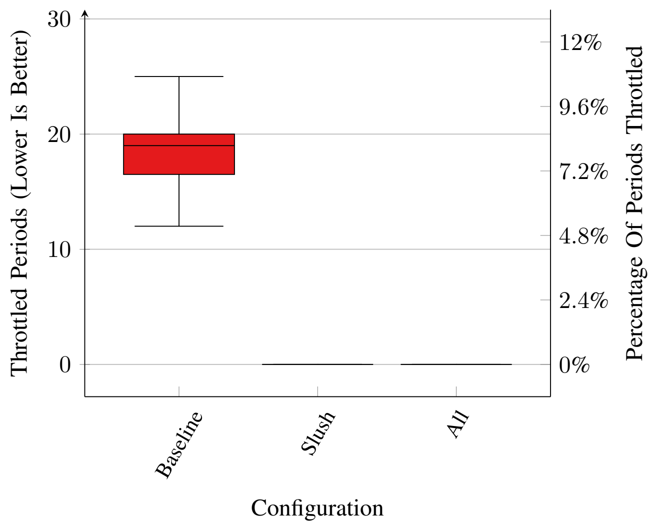


Fig. 5: Sysbench CPU Benchmark result, showing periods throttled with the given changes. All results with a 100ms CFS Period and 300ms quota, and three threads. Values based on 30 consecutive runs.

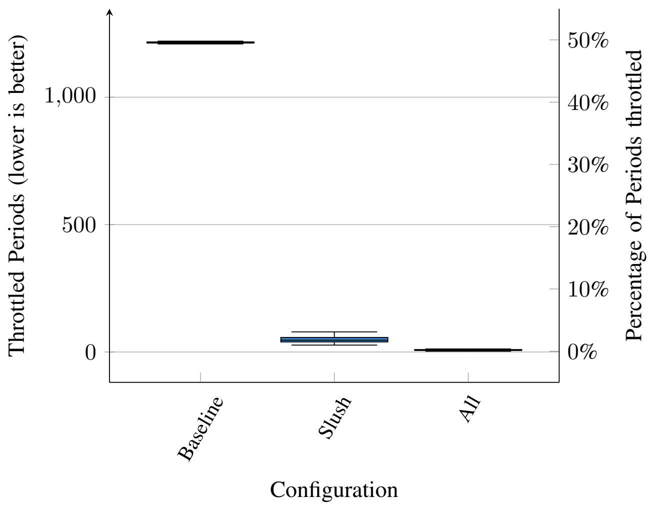


Fig. 6: Sysbench CPU Benchmark result, showing periods throttled with the given changes. All results with a 10ms CFS Period and 30ms quota, and three threads. Values based on 30 consecutive runs.

A. *Evaluating throttling mitigation*

To assess the effectiveness of the proposed techniques in mitigating throttling, we use the *Sysbench CPU benchmark [8]*. It consists of a CPU bound algorithm that calculates prime numbers, and is used as a long lived CPU bound benchmark. The benchmark takes number of threads and number of events (default 1000) as input and it outputs the time spent doing all the computations. We evaluate the following three designs:

- **Baseline**: The unmodified bandwidth control in Linux Kernel.
- **Slush**: Using slush fund to account for previous period's negative runtimes, as discussed in section IV-A1.
- **All**: In addition to slush fund, this desing also includes the modified bandwidth update logic, as discussed in section IV-A2 .

We first evaluate the reduction in the number of periods experiencing throttling and then the reduction in execution time.

1) Reduction in throttled periods: Figure 5 shows the throttled periods, absolute and percentage, experienced by different mechanisms with a 100ms period and 300ms quota, and collected over 30 runs. The figure shows that the baseline experiences a substantial amount of throttling as it throttles in about 5%-10% of the periods. In contrast, *slush* is able eliminate this throttling by using the slush fund to account for the negative runtimes of the previous period. As *slush* itself is very effective, applying the modified bandwidth update logic, i.e. *All* in Figure 5, does not provide any additional benefit as there is no opportunity left for it.

As discussed in section III-A, the throttling increases as we reduce the CFS bandwidth control period. Figure 6 shows the throttled periods, absolute and percentage, experienced by different mechanisms with a 10ms period and 30ms quota, and collected over 30 runs. As the figure shows, the baseline experiences throttling in substantially higher number of periods than it did with 100ms period in Figure 5. However, *slush* is able to significantly reduce the number of throttled periods. Even though it does not entirely eliminate throttled periods, as was the case with$100ms$ period, it does reduce the number of periods where throttling occurs from about 50% to less than 1%. Furthermore, when we apply the modified bandwidth update logic on top of slush, i.e. *All* in Figure 6, we see a further reduction in throttled periods. In addition, we observe that *All* delivers a much lower variance compared to *Slush*.

2) Runtime impact: To assess the performance improvements brought by the reduced throttling, we present the total execution time in Figure 7 for 100ms period and in Figure 8 for 10ms period. As the figures show, the baseline bandwidth control with 10ms period requires more execution time than with 100ms period because of the higher throttling at 10ms as presented in Figure 5, Figure 6 and Table II. Figures 7 and 8 also show that our proposed mechanism, *All*, significantly reduces the execution time compared to baseline, concretely about 5% reduction with 100ms period and about 12% reduction with 10ms period. These results show the effectiveness of the proposed mechanism in reducing the execution time by avoiding throttling.

B. *Evaluating bandwidth control overhead*

To assess the effectiveness of the proposed techniques in reducing bandwidth control overhead, we use the *Sysbench Thread benchmark [8]*. It is a thread-based scheduler benchmark comprised 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.

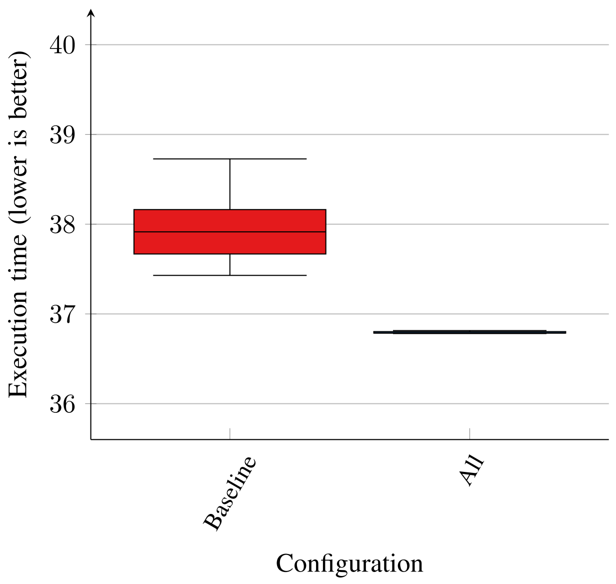


Fig. 7: Sysbench CPU Benchmark result showing execution time with 100ms period. Values based on 30 consecutive runs, all using two threads.

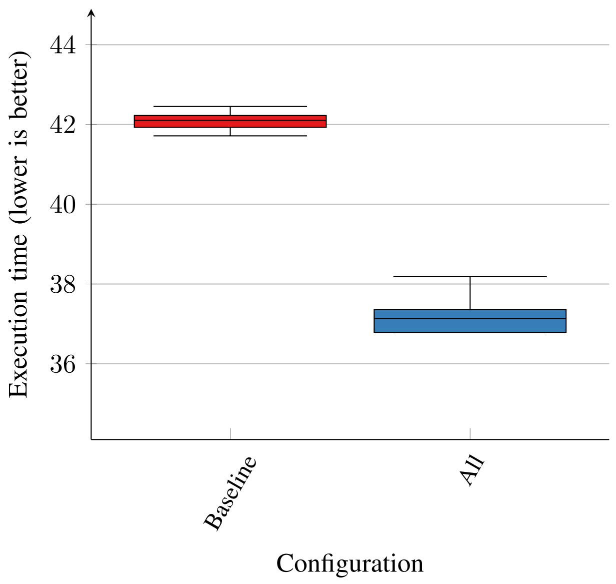


Fig. 8: Sysbench CPU Benchmark result showing execution time with 10ms period. Values based on 30 consecutive runs, all using two threads.

It is a commonly used benchmark for measuring scheduler overhead. We use a $100ms$ period, together with a quota that will never be reached which ensures that we are never throttled. This makes sure that we measure only the scheduler overhead without any interference from throttling. We evaluate the following three different slice lengths:

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

The $1\mu s$ slice length is especially interesting as it will significantly increase the lock congestion allowing us clearly differentiate the overheads of the standard spinlock based approach and our proposed atomic variable approach.

Figure 9 shows the overall execution time for spinlock and atomic variable based approaches with different slice lengths. In addition, it also shows the execution time without bandwidth control, i.e., when there is no overhead. As the figure shows the difference between spinlock and atomic variable based approaches is clearly visible at 1 μs slice length where our proposed atomic variable approach brings down the execution time from 210 seconds to 28 seconds, i.e. 7.5x performance improvement over spinlock based approach. By defining the overhead as the time spent over *No bandwidth control* (as it does not incur any overhead), we observe that for $1\mu s$ slice length, the overhead is 190 seconds for the spinlock implementation and 8 seconds for the atomic variable implementation, i.e., atomic implementation reduces the overhead by nearly 24x.

At slice lengths other than $1\mu s$, both spinlock and atomic variable based approaches perform similar. This is because we used only 38 threads in our experiments which are not enough to sufficiently stress the bandwidth control at these

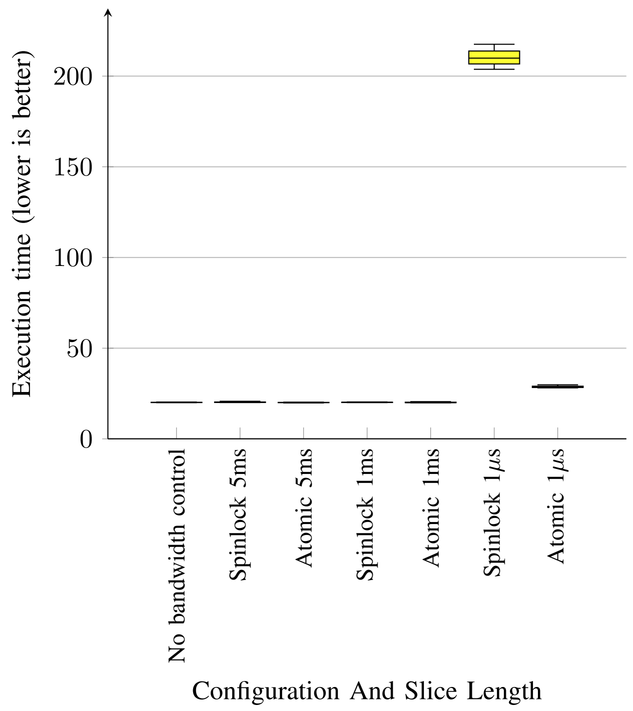


Fig. 9: Sysbench Threads Benchmark results. 38 threads on 38 logical Linux CPUs, 500 000 events and 10 locks. Values based on 30 consecutive runs.

slice lengths, hence a spinlock based approach works just fine. However, with 1 μs slice length, the waiting time to acquire the spinlock becomes very high even with 38 threads. Our atomic variable approach gracefully handles this pressure and keeps the overhead to minimal as is suggested by the results in Figure 9. As $1\mu s$ slice length is unlikely to be commonly used in practice, we observe that as the number of threads increases the pressure on spinlock starts to increase at higher slice lengths as well, thus increasing the overhead. Hence, the atomic variable based approach is likely to perform better than spinlocks in heavily loaded systems even at higher slice lengths.

VI. Conclusion

This work investigated the bandwidth control mechanism of Linux Completely Fair Scheduler (CFS). CFS is an integral component of Linux process scheduling and is key to ensuring fair allocation of hardware resources among completing processes. CFS bandwidth control sets the maximum limit on the resources that a process can use. Setting the upper limit helps CFS to discover badly behaving applications and hard cap them to limit the overall damage to the system and other applications, thereby improving overall fairness.

Despite providing excellent fairness, our analysis revealed that the bandwidth control has a major limitation that it can unnecessarily throttle processes which results in poor application performance. Further investigation revealed that the root cause of this limitation is the CPU runtime accounting mechanism of bandwidth control. This mechanism is responsible for tracking if a process has reached it maximum allocated limit. In addition to unnecessary throttling, we also found that the overhead of bandwidth control can become significantly high because of the use of spinlocks.

To mitigate the unnecessary throttling, we proposed to use *slush fund* to compensate for the negative accumulated runtime of a cgroup in a period. Since the negative runtime is compensated from the slush fund, the global quota of the new period remains intact. As a result, the bandwidth control does not need to throttle the processes. Our results show that we are able to eliminate nearly all throttling, thus providing up to 12% performance improvement. To reduce the bandwidth control overhead, we proposed to use atomic variables instead of spinlocks. Our results show that the atomic variable based approach reduces the overhead by up to 24x compared to spinlocks.

References

[1] P. Turner, B. B. Rao, and N. Rao, "Cpu bandwidth control for cfs," in *Proceedings of the Linux Symposium*, pp. 245–254, 2010.

[2] "Kubernetes - production-grade container orchestration." https://kubernetes.io/.

[3] "Assign cpu resources to containers and pods — kubernetes." https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/#cpu-units.

[4] 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*, EuroSys '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[5] "Runtime options with memory, cpus, and gpus — docker documentation." https://docs.docker.com/config/containers/resource_constraints/#configure-the-default-cfs-scheduler.

[6] "Operating system family / linux — top500." https://www.top500.org/statistics/details/osfam/1/.

[7] "Cfs bandwidth control — the linux kernel documentation." https://www.kernel.org/doc/html/v5.11/scheduler/sched-bwc.html.

[8] "akopytov/sysbench: Scriptable database and system performance benchmark." https://github.com/akopytov/sysbench.

[9] "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.

[10] "Arch linux." https://archlinux.org/.

[11] "Control group v2, cpuset controller — the linux kernel documentation." https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#cpuset.