Improving Resource Isolation of Critical Tasks in a Workload

Meghana Thiyyakat, Subramaniam Kalambur, and Dinkar Sitaram
meghanathiyyakat@pesu.pes.edu
{subramaniamkv,dinkars}@pes.edu

PES University, Bengaluru

Abstract. Typical cluster schedulers co-locate critical tasks and background batch tasks to improve the utilization of resources in the cluster. However, this leads to resource contention and interference between the diverse co-located tasks. To ensure guaranteed resource allocation and predictability, critical tasks are executed within containers as they provide resource isolation using container resource allocation mechanisms. Linux-based containers achieve resource allocation and isolation using a kernel feature known as Control Groups (cgroups). Cgroups allow the division of CPU time into shares which can be allocated to different groups of tasks. In our study, we run workloads on servers with different hardware configurations and measure the CPU time per second, or the *CPU bandwidth*, that the critical tasks in the workloads can consume. Our workloads have been generated using a cluster trace published by Google, and contain a mixture of critical and background tasks. The results of the experiments show that under high CPU load conditions, the CPU bandwidth consumed by the critical tasks is inadequate and unstable because of the poor resource isolation offered by cgroups. However, when these tasks are scheduled with the careful use of SCHED_DEADLINE policy, which is based on the Global Earliest Deadline First and Constant Bandwidth Server algorithms, they steadily consume their required CPU bandwidth irrespective of the load on the CPU. As a result, when critical tasks are scheduled using SCHED_DEADLINE, they experience 3x-40x smaller delays than under cgroups.

Keywords: Resource isolation · Cgroups · Containers · CPU Bandwidth

1 Introduction

As a result of the cost benefits of deploying multiple applications on shared infrastructure, companies are moving more and more of their computations to private and public clouds. The workloads running on these clouds comprise critical tasks and background tasks. Critical tasks need guaranteed resource allocations and/or have strict latency requirements. These are generally tasks of user-facing services such as web search that are deployed on stream processing frameworks like Storm[29] and Flink[7]. Background tasks comprise batch jobs from frameworks, such as Hadoop[30] which do not have stringent resource requirements

and may be latency-tolerant. To improve the utilization of servers, tasks with diverse characteristics and resource requirements are co-located. There have been numerous works that use heuristics to find optimal combinations of tasks to schedule on to systems such that the utilization of the infrastructure is maximized without violating service-level objectives[11, 21, 32, 10, 15, 22]. Despite the recognized benefits of shared infrastructure, it gives rise to two concerns: resource isolation [4] and security. Today, containers have emerged as a popular light-weight virtualization solution to address both these issues [23]. In Linux-based containers, two kernel features - control groups (cgroups) [8] and namespaces, are used to implement resource-isolation and security, respectively. The cgroups feature was introduced in Linux kernel version 2.6.24. It allows users to impose constraints on the resource utilization of groups of tasks. While cgroups allow users to manage multiple resources such as memory and network bandwidth, in this paper, we focus on the resource isolation offered by cgroups for CPU bandwidth.

SCHED_DEADLINE, originally designed for embedded systems [12, 5, 18], is a real-time CPU scheduling policy available in Linux kernel version 3.14 onward. The policy is based on 2 algorithms - Earliest Deadline First (EDF) and Constant Bandwidth Server (CBS). It schedules tasks based on their deadlines and performs admission control of tasks to prevent over-provisioning of the CPU bandwidth, thereby ensuring that all the deadlines can be met. While SCHED_DEADLINE has traditionally been used to guarantee temporal isolation of real-time tasks, that is, ensuring that deadlines of tasks are met, we show that it can also be used to secure the required CPU bandwidth for critical tasks, undeterred by the load on the CPU.

We use samples from the Google Cluster Trace [27] to emulate cluster workloads consisting of critical and background tasks. The results of our experiments show that when critical tasks are allocated CPU bandwidth using cgroups, they are unable to steadily consume their required bandwidth because of interference from the background tasks. When the same tasks are scheduled using SCHED_DEADLINE, they are able to consistently consume their required CPU bandwidth which results in smaller delays in their response times.

The contributions of the paper are a description of how we have used the Google Cluster Trace to emulate cluster workloads, a method to estimate the parameters required for scheduling with SCHED_DEADLINE, and a comparison of the resource isolation offered by cgroups and SCHED_DEADLINE.

The rest of the paper is organized as follows. Section 2 summarizes scheduling in the Linux operating system. Section 3 describes in detail the workloads used and the experimental setup. Section 4 presents the results of the study. Section 5 discusses related works and finally, Section 6 concludes the paper.

2 Background

Linux is a multitasking general-purpose operating system and must concurrently execute interactive as well as CPU bound jobs. Therefore, the scheduler needs to

give preference to interactive tasks to ensure quick responses, while ensuring that CPU-bound jobs do not starve. Linux also supports real-time scheduling policies to handle tasks with real-time constraints. SCHED_DEADLINE, SCHED_FIFO, and SCHED_RR are the real-time policies available in the Linux scheduler. The SCHED_DEADLINE policy, based on Global Earliest Deadline First [19] and Constant Bandwidth Server [2] algorithms, is used to grant temporal isolation to tasks and predictability in their execution. In our work, we show that SCHED_DEADLINE offers better resource isolation than cgroups.

2.1 Completely Fair Scheduler

The Completely Fair Scheduler [25] is Linux's default scheduling policy. Tasks scheduled under SCHED_OTHER, SCHED_IDLE and SCHED_BATCH policies are all handled by CFS. In our work, we only consider one of CFS's policies: SCHED_OTHER. Since CFS does not provide implicit resource isolation, we have used the results of critical tasks scheduled under CFS (SCHED_OTHER) as a baseline to compare the CPU bandwidth allocation with and without resource isolation. In all the scheduling scenarios in our study, the background tasks are scheduled using SCHED_OTHER only.

2.2 SCHED_DEADLINE

The SCHED_DEADLINE policy was added to the Linux kernel in 2014, to version 3.14 and is based on the Global Earliest Deadline First and Constant Bandwidth Server algorithms. Tasks scheduled under this policy are given the highest priority in the system and can preempt tasks of all other policies. Since the scheduling policy is based on CBS it ensures non-interference between tasks by throttling threads that try to consume more than their allotted share of the CPU. While CFS ensures maximum utilization of the CPU and that no process starves, SCHED_DEADLINE provides predictability to tasks that have strict deadlines and latency constraints. To schedule a task using SCHED_DEADLINE, 3 parameters are passed to the scheduler — `sched_runtime` (budget), `sched_period` (period) and `sched_deadline` (deadline). The budget denotes the amount of CPU time the task needs every period. Hence, the share of CPU time allotted to the task is `sched_runtime` ÷ `sched_period`. The sched_deadline parameter conveys to the scheduler that it must allocate `sched_runtime` seconds of CPU time to the task within `sched_deadline` seconds of each period. If a task tries to consume more CPU time than its budget, it is throttled until its next period. This property, combined with the fact that the tasks of this policy are assigned the highest priority, enforces resource isolation between the tasks. To ensure that deadlines can be met and that the required budget can be sanctioned to each task every period, the scheduler performs a schedulability test. The schedulability test only allows a new thread to enter the runnable thread pool if the sum of the CPU usage rates of the new thread and the existing threads in the pool

continues to be less than the number of processors. That is,

$$\sum_{i=1}^{n} \frac{sched\_runtime}{sched\_period} <= N \tag{1}$$

where N is the number of processors, and n is the runnable threads in the system, including the new task. However, to ensure that non-real-time tasks do not starve, an upper limit is enforced on the total CPU usage rate of real-time tasks. To reflect this the above equation is modified:

$$\sum_{i=1}^{n} \frac{sched\_runtime}{sched\_period} <= N \times rt\_quota\% \tag{2}$$

where rt_quota% is the cap on the aggregate CPU usage rate of real-time tasks. rt_quota is equal to 95% by default in Linux.

In our work, we make use of the resource isolation offered by SCHED_DEADLINE while scheduling critical tasks. However, since the tasks may be long-running, aperiodic, and have dynamic resource requirements, we use the SCHED_DEADLINE policy differently from its conventional usage for real-time tasks. In our work, we show that the policy can also be used to secure the CPU bandwidth necessary for the critical tasks to run without delays. Since the bandwidth requirements may vary with time, for every window of time during which the CPU bandwidth is constant, SCHED_DEADLINE is used to secure the needed CPU bandwidth. A description of this modified usage of SCHED_DEADLINE is given in Section 3.3. When scheduling critical tasks using SCHED_DEADLINE, if a task fails the schedulability test, we put the task to sleep for 0.1s, and then try to reschedule the task. We repeat this step until the task is successfully scheduled. The sleep duration was determined empirically based on the trade-off between the polling overhead and delay in scheduling the thread.

2.3 Control Groups

Control groups, or cgroups, were introduced as a part of the Linux kernel in version 2.6.24. They allow system administrators to make resource reservations and partitions for groups of tasks. The cgroups interfaces to resources such as CPU or memory are known as subsystems. In our work, we focus on the CPU only and have thus made use of the CPU subsystem. Linux-based containers use the CPU subsystem to enforce CPU bandwidth isolation with the help of shares. Shares are integers used to describe the relative share of total CPU bandwidth that a cgroup is assigned. That is, a cgroup's portion of the total CPU bandwidth is the number of shares assigned to the cgroup divided by the total number of shares available. If a task is not assigned explicitly to a cgroup, it comes under the root cgroup. In our study, we assign the critical tasks to a cgroup, group_p. We have allocated group_p with 95% in the initial runs, and then with 99% of the CPU shares. The number of shares assigned to group_p is calculated using the following formulae:

$$\frac{group\_p\_shares}{group\_p\_shares + root\_shares} = share\% \tag{3}$$

The root cgroup has 1024 shares by default. If the subsystem has only one explicitly defined group — group_p, it has to be assigned 19456 shares to give the tasks in group_p access to a minimum of 95% of the total CPU bandwidth (using Eq. 3).

Cgroups also allow users to enforce hard upper limits on the amount of CPU Time that can be consumed in a specified period. This is done by defining quotas. In our work, we have not used quotas for the background tasks to ensure optimum CPU utilization. When the critical tasks do not require the remaining CPU bandwidth, the background tasks scheduled using quotas cannot make use of this bandwidth, thereby reducing the overall utilization of the CPU and introducing unnecessary delays in the background tasks.

Scheduling of tasks under cgroups is undertaken by CFS by default. In our work, the critical tasks assigned to cgroups are scheduled using CFS (SCHED_OTHER).

3 Methodology

The objective of our study was to compare the resource isolation offered by cgroups and SCHED_DEADLINE policy. To do so, we measured the CPU bandwidth consumed by the critical tasks in the workloads under different scheduling scenarios. When resource isolation is not robust, interference from the co-located tasks does not allow critical tasks to consume the required CPU time at the rate needed by the tasks. This leads to delays in their response times. While robust resource isolation could be guaranteed to critical tasks by scheduling them on dedicated CPU cores, such course-grained allocation leads to severe underutilization of the CPU resources. Hence, we have used cgroups and SCHED_DEADLINE to allocate resources at a finer granularity.

In this paper, the terms *CPU usage rate*, *bandwidth*, *share* and *utilization*, all represent the amount of CPU Time that is consumed or required every second. While the terms all measure the same quantity, they have been used depending on the context of the measurement and cannot be used interchangeably. The quantity, *CPU usage rate*, used by Google in their cluster trace, is measured in core-second/second. We have used the term *CPU bandwidth* to refer to the ratio of the CPU Time consumed or required per second, to the total CPU Time available per second. We use the term "Total CPU bandwidth" to denote 100% CPU bandwidth. Therefore, if a task running on a 4-core machine has a CPU bandwidth requirement of 20%, it consumes 20% of the total CPU time available every second, that is, 20% of 4 core-seconds. Therefore, its CPU usage rate is 0.8 core-second/second. We have used the term *share*, as per the cgroups definition, for the relative amount of CPU Time allotted to a cgroup per second. The term *CPU utilization* has been used to denote the utilization of the entire CPU of the server when executing a workload or a part of it.

3.1 Workloads

The two workloads used in our experiments were created from the Google Cluster Trace [27]. The trace, published by Google in 2011, contains data about the different tasks that run on a 12k node cluster for 29 days. It includes measurements such as CPU and memory usage, task characteristics such as priorities and scheduling classes, and the attributes of the machines on which the tasks are deployed. All the dimensions in the trace have been normalized relative to the largest capacity of a resource.

3.2 Workload Generation

To generate the High CPU Utilization (HCU) workload, we first chose a machine from the trace that had experienced a high CPU load. To do so, we selected all machines from the machine_events table with CPU capacity 1 (maximum capacity, since all the resource dimensions are normalized). The utilization of the cluster is maximum during days 22 and 23 of the trace [28]. We found the average utilization of the selected machines on these days by joining tables task_usage and task_events. The tasks_events table contains information about the priority and scheduling class of each task and the task_usage table holds a log of the resource consumption of each task for every measurement window. The measurement window is typically 300s unless it is the beginning or end of a task's execution lifetime. From the top 10 machines with the highest average utilization, we chose a machine that had tasks with a diverse mixture of priorities and scheduling classes. For our second workload, the Low CPU Utilization (LCU) workload, we sampled the resource usage data for the same machine but for day 10 — during which the CPU utilization was relatively lower. The LCU workload, therefore, has a lower overall CPU utilization.

Since our study only concerns CPU time, each row of the resulting trace sample consisted of the following values:

1. Start time of the measurement period
2. End time of the measurement period
3. Mean CPU usage rate
4. Priority
5. Scheduling class

As all measurements in the trace are normalized, the "Mean CPU usage rate" in the sample, measured in core-second/second, ranged from 0-1. Google has chosen not to disclose their machine specifications to the extent we require, hence, to be able to denormalize the resource measurements, we made assumptions about the values of the CPU clock speed of the machine chosen. We used the parameters of a typical machine of that period to assume that the chosen machine, machine A, was a 4-core machine with a processing speed of 2.1 GHz. While generating the workload, we have made a simplifying assumption that the CPU clock speed is constant for the values reported in the trace. However, we have executed the workload on real systems with varying CPU speeds. The

assumption holds since we use the results to perform comparisons between different runs on a given system only, and not across systems. We calculated the following values for each measurement period in the trace as per the assumptions made:

$$ExecutionTime = Starttime - Endtime \tag{4}$$

$$CPURuntimeA = Normalized\_CPU\_Usage\_Rate \times 4 \times ExecutionTime \tag{5}$$

Here, *Starttime* and *Endtime* are the beginning and end of the measurement window in the trace. $CPURuntimeA$ is the runtime of the task on machine A and $Normalized\_CPU\_Usage\_Rate$ is the normalized mean CPU usage rate given in the trace for each measurement window. We multiply by 4 to undo the normalization based on the assumed number of cores. To run the workloads on the target machine, machine B, we scaled them appropriately by performing the following calculations:

$$ClockCycles = CPURuntimeA \times ClockSpeedA \tag{6}$$

$$= CPURuntimeB \times ClockSpeedB \tag{7}$$

$$\therefore CPURuntimeB = \frac{CPURuntimeA \times ClockSpeedA}{ClockSpeedB} \tag{8}$$

$CPURuntimeB$ is the denormalized CPU time the task is to consume on machine B, $ClockSpeedA$ and $ClockSpeedB$ are the CPU speeds of the chosen machine, machine A, and the target machine, machine B, respectively.

Each task in the workloads was executed as a separate multi-threaded process. When the CPU usage rate of a task was greater than 1 core-second/second, the rate was split equally among multiple threads and scheduled to run on multiple cores. To consume $CPURuntimeB$ seconds of the CPU, the task thread executed a few mathematical statements for $CPUSlice$ seconds and was then put to sleep for *SleepSlice* seconds. The task thread repeated these steps until the total CPU Time consumed by it equaled $CPURuntimeB$. The parameters were calculated as follows:

$$SleepTimeB = ExecutionTime - CPURuntimeB \tag{9}$$

$$NumSlices = \frac{ExecutionTime}{Slice} \tag{10}$$

$$CPUSlice = \frac{CPURuntimeB}{NumSlices} \tag{11}$$

$$SleepSlice = \frac{SleepTimeB}{NumSlices} \tag{12}$$

The value of *Slice* was set to 0.05. Algorithm 1 summarizes how the tasks were modeled and emulated based on the calculated values.

For each task, a logger thread maintained a log of the CPU usage rate of the task. The aggregate CPU usage rate of each task class was later found

Algorithm 1: Pseudo-code for running a task in the workloads.

```
Input: trace sample trace
begin
    Function RUN_ON_CPU(row)
    slice = 0.05 num_slices = row.window/slice
     cpu_slice = cpu_time/num_slices sleep_slice = sleep_time/num_slices
     current_cpu_time = 0
    while current_cpu_time < cpu_time do
        Run math operations on CPU till CPU time equal to cpu_slice is
         consumed;
        Sleep for sleep_slice seconds;
        current_cpu_time = current_cpu_time + cpu_slice;
    end
    Function Main(trace)
    foreach row ∈ trace do
        if row.cpu_time > row.window then
            num_threads = ceil(row.cpu_time/row.window)
             row.cpu_time = row.cpu_time/num_threads
            for i ← 0 to num_threads 1 do
                execute RUN_ON_CPU(row) on a new thread
            end
        else
            RUN_ON_CPU(row)
        end
    end
    return Task Completed
end
```

and plotted. The average delay in the tasks, as well as the delay observed in each measurement window, were plotted for each scheduling scenario. We also recorded the overall CPU utilization % of the workloads every second.

Only tasks with priority 9 or higher and scheduling class greater than 0 were considered as "critical". Other tasks were labeled as "background". We did so based on the analysis of the trace by Reiss et al. [28] and the description of the trace parameters published by Google [27]. Since there are no descriptions of the jobs and tasks in the trace, we assume that critical tasks are those tasks that have stringent-latency constraints and/or need strict resource allocation guarantees.

We created two different workloads to study resource isolation under different CPU load conditions. The HCU workload consisted of 7 critical tasks and 7 background tasks. A summary of the tasks in the workload scaled for the 4-core machine according to the methodology described in Section 3.2, can be found in Table 1. The LCU workload contained 9 critical tasks and 33 background tasks. Both workloads had a runtime of around 2 hours. The CPU bandwidth requirements of the two categories of tasks in the HCU and LCU workloads are shown in Fig. 1.

Table 1: Description of the HCU workload tasks on the 4-core machine.

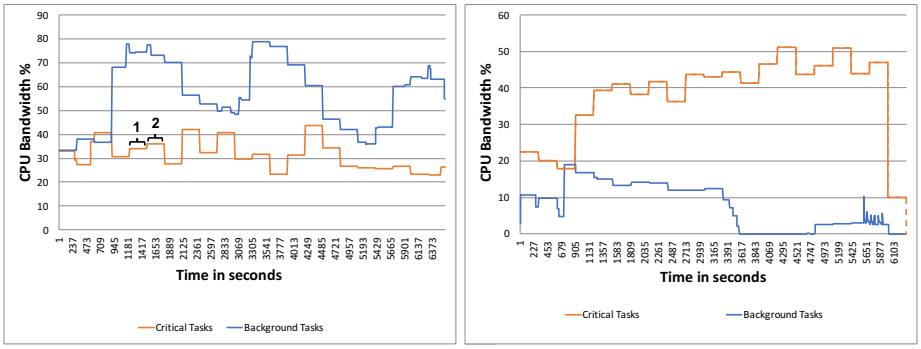
Task No	Type	Mean CPU Usage Rate [Std Dev] (in core-second/second)	Duration (in s)
1	critical	0.186 [0.024]	5700
2	critical	0.179 [0.018]	5700
3	critical	0.162 [0.019]	6300
4	critical	0.039 [0.042]	6600
5	critical	0.030 [0.025]	6600
6	critical	0.042 [0.040]	6600
7	critical	0.705 [0.154]	6600
8	background	1.398 [0.266]	6600
9	background	0.247 [0.085]	6600
10	background	0.239 [0.090]	6600
11	background	0.231 [0.120]	6600
12	background	0.261 [0.124]	5438
13	background	0.247 [0.094]	6600
14	background	0.001 [0.0]	246

The delay in the response time was calculated as follows:

$$ResponseTime = Task\_ETime - Task\_STime \tag{13}$$

$$Delay = ResponseTime - TaskExecutionTime \tag{14}$$

Where $TaskExecutionTime$ is defined as the duration between the start and end of the task as per the trace. $Task\_STime$ and $Task\_ETime$ are the actual wall clock times recorded at the start and end of the task thread's execution, respectively.



(a) (b)

Fig. 1: CPU bandwidth requirement of each task category in the a) HCU workload, and b) LCU workload.

3.3 Experimental Setup

To compare the resource isolation offered by the two resource allocation mechanisms, the upper limit on the CPU bandwidth allocated to critical tasks had to be set equally for both. In our experiments, we set the upper limit as 95% of the total CPU bandwidth. This is because, in Linux, the CPU bandwidth that real-time tasks are allowed to consume is capped, by default, at 95% of the total CPU bandwidth. The 2 system-wide settings: `sched_rt_period_us` and `sched_rt_runtime_us`, found in `/proc/sys/kernel/`, determine the CPU bandwidth that real-time tasks are allowed to consume [26] and have default values of 1s and 0.95s, respectively, restricting the CPU usage of real-time tasks to 95% of the total CPU bandwidth. Therefore, for a fair comparison of the two scheduling scenarios, we assigned 95% of the CPU shares to the cgroup with the critical tasks. While the critical tasks scheduled using SCHED_DEADLINE were allotted a *maximum* of 95% of total the CPU bandwidth, the tasks scheduled using cgroups were allotted a *minimum* of 95% of the total CPU bandwidth. However, since SCHED_DEADLINE tasks are always assigned the highest priority in the system, the tasks can consume up to 95% of the total CPU bandwidth without being throttled or having to wait for lower priority tasks to yield the CPU. Therefore, in both scenarios, 95% of the total CPU bandwidth was allocated to the running of the critical tasks. We ran the workloads under 3 scenarios: i) using neither cgroups or SCHED_DEADLINE (only SCHED_OTHER), ii) selectively assigning the critical tasks to a cgroup with 95% share of the total CPU bandwidth, and iii) selectively scheduling critical tasks using SCHED_DEADLINE. The results of scenario (i) were used as a baseline to compare the performance of tasks with and without resource isolation.

We ran the workloads on two Intel Xeon machines. Since some cloud-workloads comprise tasks deployed on containers which, in turn, run on Virtual Machines (VMs), we also conducted our experiments on two EC2 instances. The details of the servers are summarized in Table 2. In the table, HT stands for Hyperthreading.

Table 2: Server Configurations

Type	CPU Speed GHz	Logical Cores or vCPUs	Linux Kernel Version
EC2 Instance- c5n.xlarge	3.0	4	4.15
EC2 Instance- c4.2xlarge	2.6	8	4.15
Intel Xeon E3-1220	3.1	4	5.5
Intel Xeon E5-2683 v4 (with HT)	2.1	32	5.5

Cgroups: For the cgroups scenario, the critical tasks were scheduled to a cgroup `group_p`. Since the CPU bandwidth of real-time tasks is capped, by default, at

95% of the total CPU bandwidth, we assigned 95% of the total CPU shares to group_p to allow us to compare the two scheduling scenarios. The experiment was also repeated with 99% of the total CPU shares being allotted to `group_p`.

Dynamic Parameter Estimation for SCHED_DEADLINE: To schedule tasks using SCHED_DEADLINE, 3 parameters need to be set: `sched_runtime`, `sched_deadline` and `sched_period`. A task thus scheduled is given `sched_runtime` seconds of CPU Time every `sched_period` seconds with a relative deadline of `sched_deadline` seconds from when the task begins execution.

Since the value of `sched_runtime` required by a task is generally unknown before-hand, we use a dynamic profiling method to estimate the value of `sched_runtime`. We split the execution of every task in each measurement window of the trace into 2 stages: a short profiling stage at the beginning of the window, and an execution stage spanning the rest of the window. We utilised the resource isolation offered by SCHED_DEADLINE even during the profiling stage to get an accurate estimate of the CPU usage rate required by the task. During its short profiling stage, each thread in every runnable task was allotted an entire CPU core for the duration of the profiling stage. That is, if the duration of the stage is t seconds, `sched_runtime`, `sched_deadline` and `sched_period` are all set to t. This allowed the task to run unhampered at the CPU usage rate it required without being throttled or having to deal with interference from other tasks. After the short profiling stage, we measured the required CPU usage rate of the task as :

$$CPU\_usage\_rate = \frac{CPUTime}{ProfilingWindow}$$

Where $ProfilingWindow$ is the duration of the profiling stage, and $CPUTime$ is the amount of CPU time(on a single core) consumed by the task during the profiling stage. A representation of the stages can be found in Fig. 2. SCHED_DEADLINE performs a schedulability test (Eq. 2) before admitting tasks into its runnable pool. Hence, if the server has N cores, and t_n is the number of concurrent tasks that are executing in their profiling stage at an instant, $t_n <= N$. When the number of tasks that wish to enter their profiling stages, t_p, is greater than N, then at least t_p - N tasks fail the schedulability test and need to wait for at least one task to exit its profiling stage before they can enter their own. We empirically determined the size of the Profiling Window such that the wait time of the tasks did not cause significant delays in their response times. Once the CPU_usage_rate was found, the `sched_runtime` parameter was set to CPU_usage_rate seconds and the `sched_period` and `sched_deadline` parameters were set to 1 second. By doing so, we guaranteed that the task got its required CPU bandwidth every second. Therefore, instead of using SCHED_DEADLINE to guarantee a deadline was met, we used it to secure the CPU bandwidth required by the task.

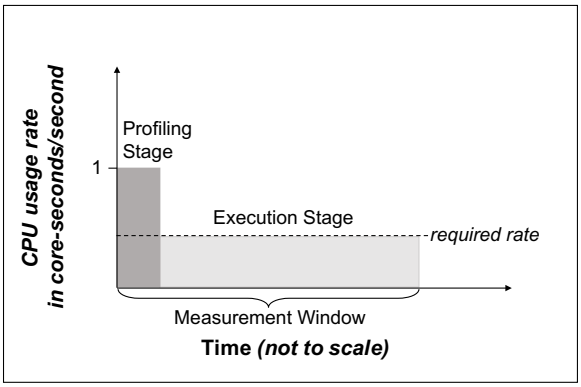


Fig. 2: Dynamic `sched_runtime` estimation

Since the Google Cluster trace only reports the mean CPU usage rate for every measurement window (usually 300 seconds long), we modeled the tasks such that they had a constant CPU usage rate during a measurement window — equal to the value given in the trace. After every window, we ran the profiling stage again to find the current required rate and rescheduled the tasks with the updated parameter values. In the future, we wish to extend our work to tasks with continuously varying CPU usage rates.

By default, Linux restricts the total usage rate of real-time tasks to 95% of the total CPU bandwidth to ensure that non-real-time tasks are not starved. We retained the default value while running our workloads under SCHED_DEADLINE.

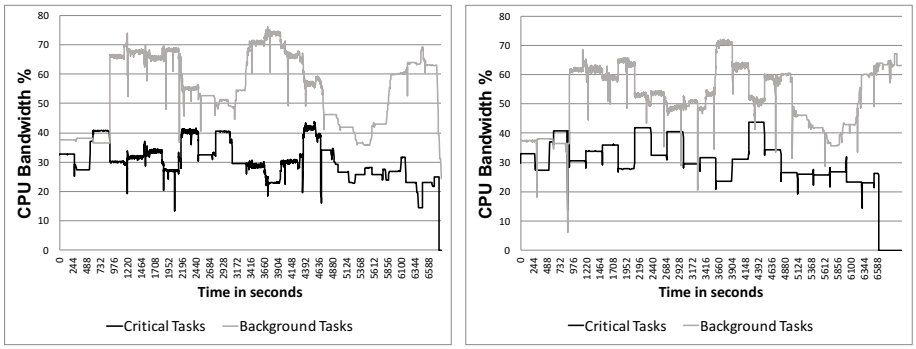
4 Results

In this section, we discuss the results of running the LCU and HCU workloads on the 4 servers under the different scheduling scenarios described in Section 3.

4.1 CPU bandwidth consumption

Fig. 3 shows the CPU bandwidth consumed by the critical and background tasks of the HCU workload on the 32-core physical machine. In Fig. 4 and Fig. 5, the CPU bandwidth consumed by the critical tasks in the HCU workload, scheduled using CFS, cgroups, and SCHED_DEADLINE, is plotted.

The results show that on all the servers, the bandwidth consumed by the critical tasks under CFS and cgroups was unsteady and lesser than the required CPU bandwidth for most of the run. However, the CPU bandwidth consumed by the critical tasks scheduled with SCHED_DEADLINE did not fluctuate as much and was equal to the required bandwidth for most of the run. This observation also validates our methodology used for the dynamic estimation of the `sched_`



(a) (b)

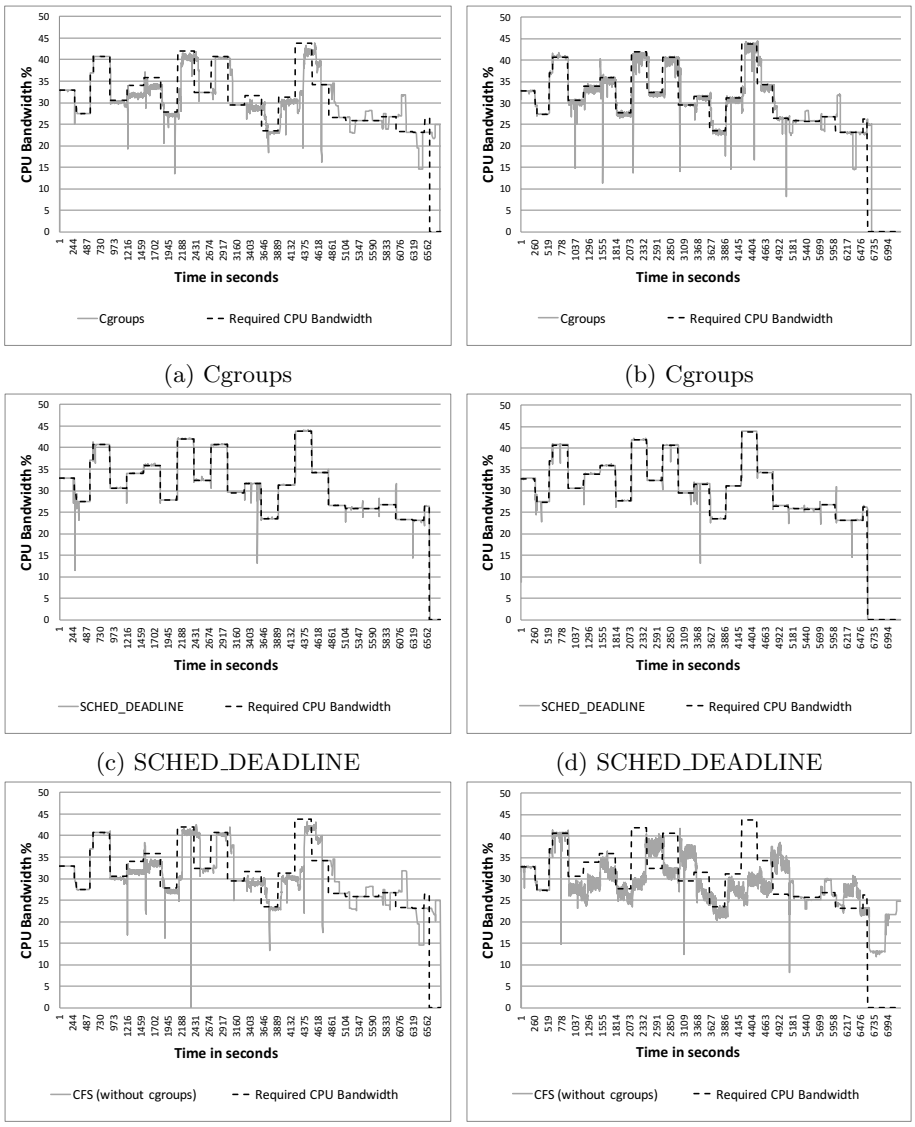
Fig. 3: CPU bandwidth consumed by each task category in the HCU workload when scheduled on the 32-core physical machine, using a) cgroups, and b) SCHED_DEADLINE.

`runtime` parameter for the critical tasks(Section. 3.3). The mean absolute percentage error (MAPE) and the standard error (SE) in the estimation of `sched_runtime` for the critical tasks in the HCU workload are recorded for each server in Table 3.

Table 3: Mean Absolute Percentage Error and Standard Error of the estimated `sched_runtime` parameter in the HCU workload

Server	Mean Absolute Percentage Error	Standard Error (in core-second/second)
EC2 Instance- c5n.xlarge	4.694%	0.043
EC2 Instance- c4.2xlarge	3.726	0.052
Intel Xeon E3-1220	5.349%	0.052
Intel Xeon E5-2683 v4	1.249%	0.030

The standard deviation of the CPU bandwidth consumed during the intervals of time indicated by the numbers 1 and 2 in Fig. 1a is tabulated in Table 4a and Table 4b. From the table it is observed that the standard deviation of the bandwidth consumption is consistently smaller in the SCHED_DEADLINE scenario, proving that the CPU bandwidth consumption under SCHED_DEADLINE is steadier. We have chosen the two intervals towards the beginning of the runs because, towards the latter part of the runs, the CPU bandwidth curve for the cgroups scenario lags behind due to delays arising from insufficient bandwidth allocation. During intervals 1 and 2, the transitions in the CPU bandwidth consumption in the two time-series are still aligned with respect to time, therefore, allowing us to compare them.

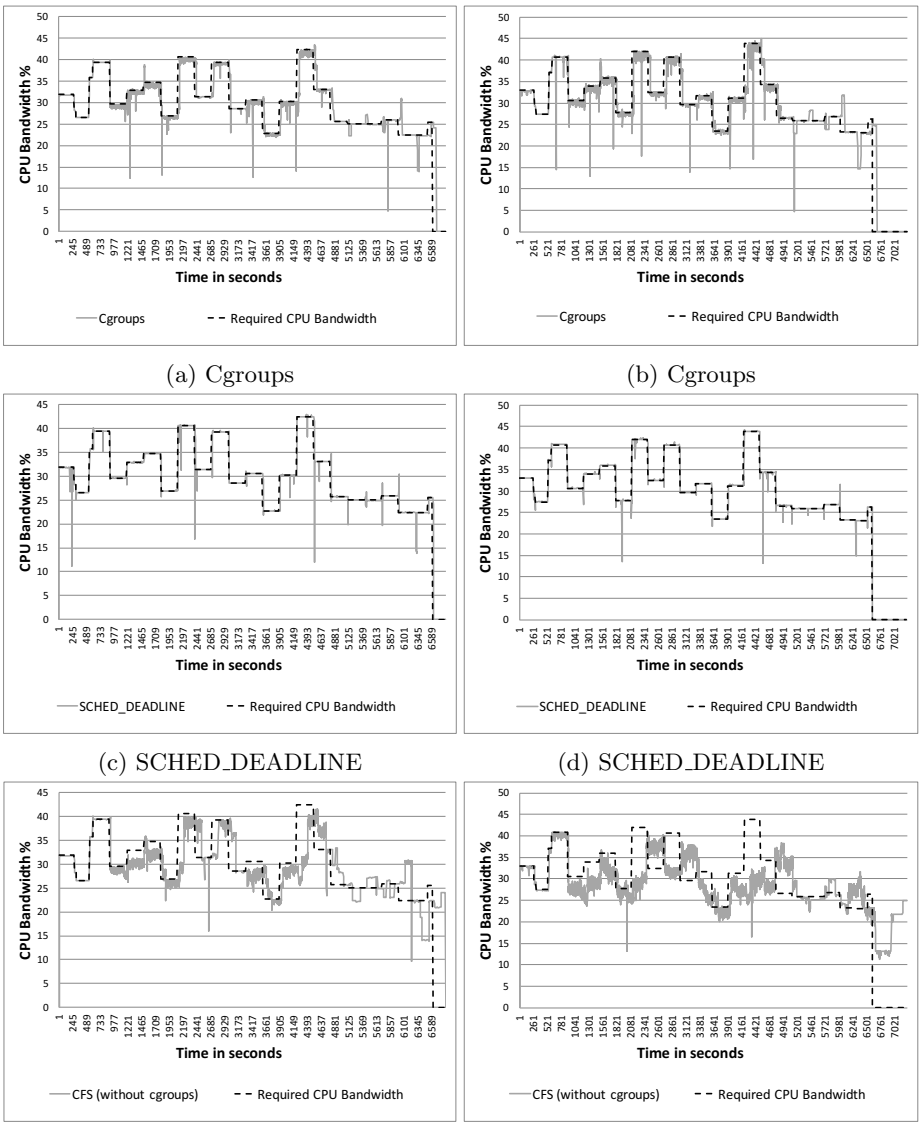


(a) Cgroups (b) Cgroups

(c) SCHED_DEADLINE (d) SCHED_DEADLINE

(e) CFS (f) CFS

Fig. 4: CPU bandwidth consumption of critical tasks under cgroups on the a) 32-core physical machine and b) 4-core physical machine, under SCHED_DEADLINE on the c) 32-core physical machine and d)4-core physical machine, and under CFS on the e) 32-core physical machine and f) 4-core physical machine



(a) Cgroups (b) Cgroups

(c) SCHED_DEADLINE (d) SCHED_DEADLINE

(e) CFS (f) CFS

Fig. 5: CPU bandwidth consumption of critical tasks under cgroups on the a) c4.2xlarge instance, and b) c5n.xlarge instance, under SCHED_DEADLINE on the c) c4.2xlarge instance and d) c5n.xlarge instance, and under CFS on the e) c4.2xlarge instance and f) c5n.xlarge instance

Table 4: Standard deviation of the CPU bandwidth % consumed, in a) Interval 1 and b) Interval 2, by critical tasks scheduled using SCHED_DEADLINE and cgroups

(a) Interval 1

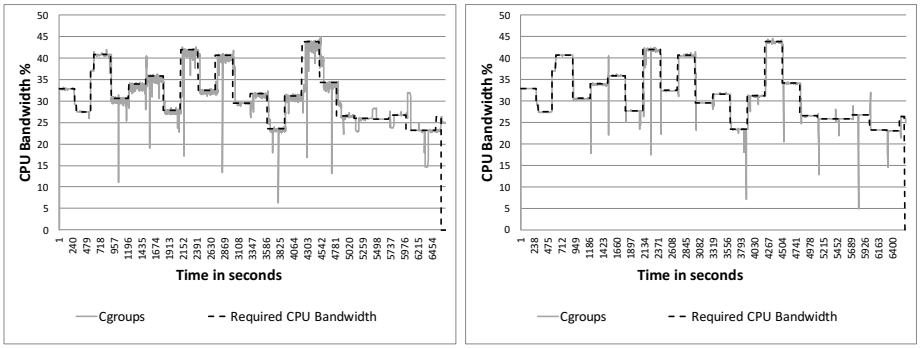
Server	SCHED_DEADLINE	Cgroups
4-core Physical Machine	0.032	0.422
32-core Physical Machine	0.031	1.235
EC2 Instance- c4.2xlarge	0.026	1.409
EC2 Instance- c5n.xlarge	0.049	1.518

(b) Interval 2

Server	SCHED_DEADLINE	Cgroups
4-core Physical Machine	0.044	1.745
32-core Physical Machine	0.054	0.578
EC2 Instance- c4.2xlarge	0.027	0.805
EC2 Instance- c5n.xlarge	0.028	1.120

A similar trend was found when the critical tasks were scheduled under a cgroup allotted with 99% of the CPU shares (Fig. 6). The standard deviation of the CPU bandwidth consumed in intervals 1 and 2 is given in Table 5.

The CPU bandwidth consumed by the critical tasks in the LCU workload, under the different scheduling scenarios, on the 32-core physical machine is reported in Fig. 7.



(a) (b)

Fig. 6: CPU bandwidth consumption of critical tasks under cgroups (99% share allotted) on the a) 4-core physical machine, and b) 32-core physical machine.

Table 5: Standard deviation of the CPU bandwidth % consumed, in intervals 1 and 2, by critical tasks scheduled under the cgroup with 99% of the CPU shares

	32-core physical machine	4-core physical machine
Interval 1	0.087	0.783
Interval 2	1.139	1.296

Since the background tasks in the LCU workload have a lower CPU utilization than those in the HCU workload, the interference posed by the tasks is bound to be lesser. Hence, it can be seen that the CPU bandwidth consumed under both scheduling scenarios was nearly equal to the bandwidth required. However, the CPU bandwidth consumed by critical tasks under SCHED_DEADLINE was still found to be marginally greater than that consumed under cgroups. Since the difference is not apparent from the plot, we have briefly summarized the distribution of the CPU bandwidth consumed by the critical tasks in Table 6.

4.2 Delay in Tasks

The maximum CPU utilization of the critical tasks in the workloads was found to be far less than 95% (e.g., only 43.8% on the 32-core physical machine). Therefore, it is expected that if the critical tasks are assigned to a cgroup with 95% of the CPU shares, the tasks must execute without considerable delay in

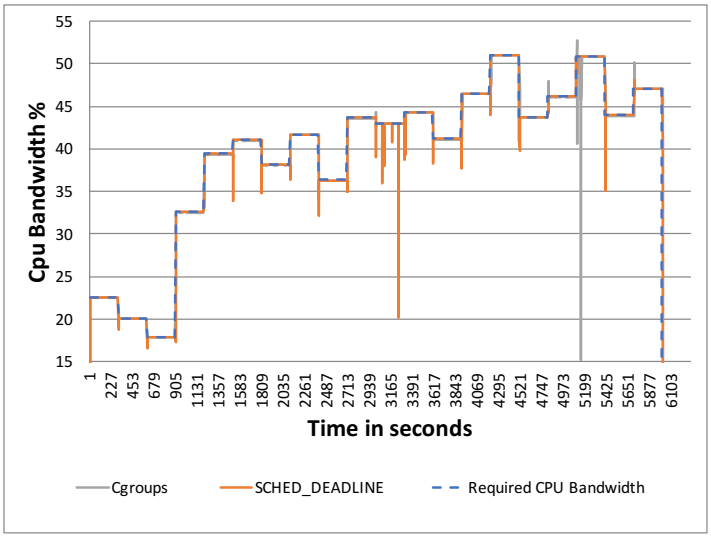


Fig. 7: CPU bandwidth consumption of the critical tasks in the LCU workload, under cgroups and SCHED_DEADLINE, on the 32-core physical machine.

Table 6: Distribution of the CPU bandwidth % consumed by the critical tasks in the LCU workload on the 32-core physical machine.

	SCHED_DEADLINE	Cgroups
1st Quartile	36.317	36.282
Median	41.6641	41.6143
3rd Quartile	44.283	44.246

their response times. The average delay in the response times of tasks in the HCU workload, for the 3 scenarios: CFS, selective scheduling of critical tasks using a cgroup with 95% of the CPU shares, and selective scheduling of critical tasks using SCHED_DEADLINE (with the default 95% cap on CPU usage rate), is reported for the 4 servers in Fig.8. The average delay for each task category has been calculated as the sum of the delays in each of its task, divided by the number of tasks in the category. The delay in each task was found using Eq. 14.

From the plots, it is evident that despite the cgroup having been allotted with more than the required number of CPU shares, the response times of the critical tasks assigned to it were very poor. The average delay of critical tasks when they were assigned to cgroups was 3 to 40 times larger than the average delay of the tasks under SCHED_DEADLINE. These results were found to be consistent across all 4 servers.

The average delay in the response times of the tasks in the LCU workload under the different scheduling scenarios is shown in Fig. 9. While the delays were smaller than those in the HCU workload, it can be seen that, here too, the delay in the critical tasks scheduled with SCHED_DEADLINE was smaller than the delay under cgroups. However, unlike the HCU workload, the average delay in the background tasks in the LCU workload is smaller than the average delay in the critical tasks. We suspect this is due to two reasons. First, the CPU bandwidth requirement of the critical tasks in the LCU workload is much higher than that of the background tasks. Second, the background tasks in the LCU workload are short-lived while the critical tasks are long-running tasks. Both these factors cause the critical tasks to be more adversely affected by resource contention leading to larger delays. In the case of the HCU workload, the background tasks had a higher CPU bandwidth requirement when compared to the critical task. Since the background tasks are scheduled using CFS which penalizes CPU-bound jobs, larger delays were found in the background tasks of the HCU workload. We also suspect that the reason behind the slightly higher average delay in the critical tasks of the LCU workload under cgroups, when compared to CFS, is due to contention within the cgroup itself.

To further study the resource isolation, the delays observed in the critical tasks at the end of each measurement period and the CPU utilization of the HCU workload were recorded throughout the run (Fig.10). It was found that under cgroups, the delays of the critical tasks in each measurement period increased considerably when the CPU utilization of the workload was high. Hence, it is

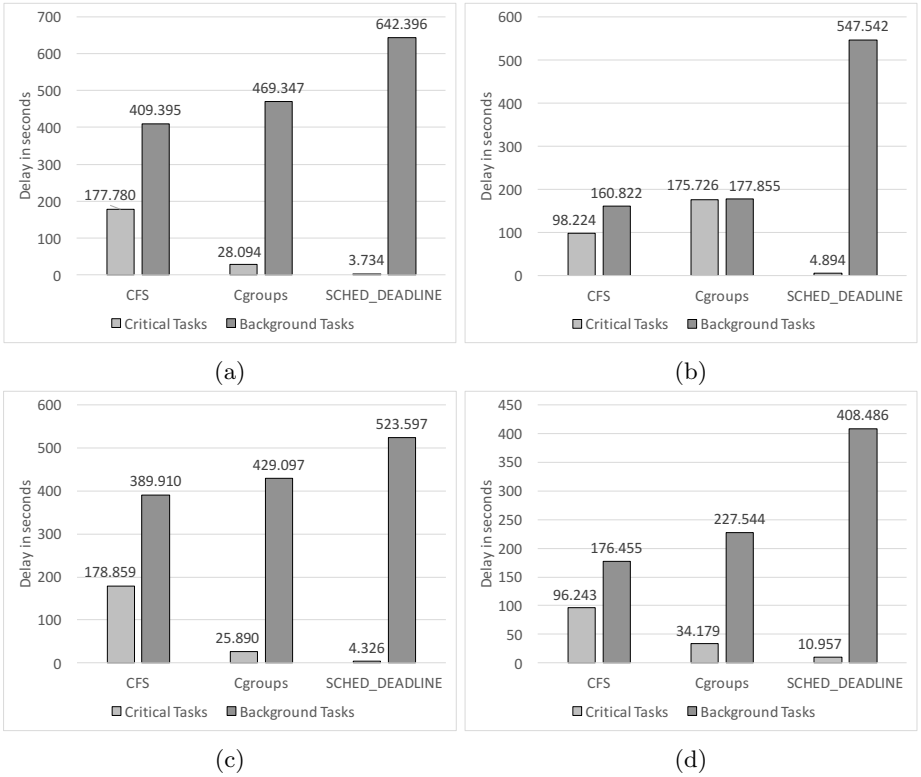


Fig. 8: Average delay in the response time of tasks of the HCU workload on the a) 4-core physical machine, b) 32-core physical machine, c) c5n.xlarge instance, and d) c4.2xlarge instance.

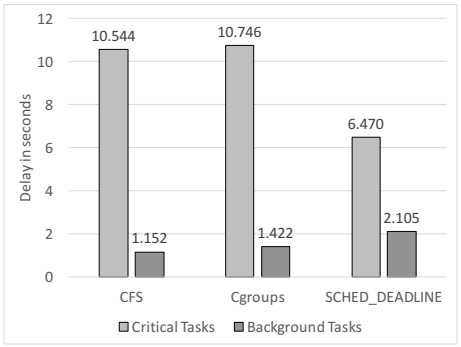


Fig. 9: Average delay in LCU workload tasks on the 32-core physical machine.

evident that the performance of the critical tasks under cgroups was not immune to the load on the CPU. However, under SCHED_DEADLINE, the magnitude

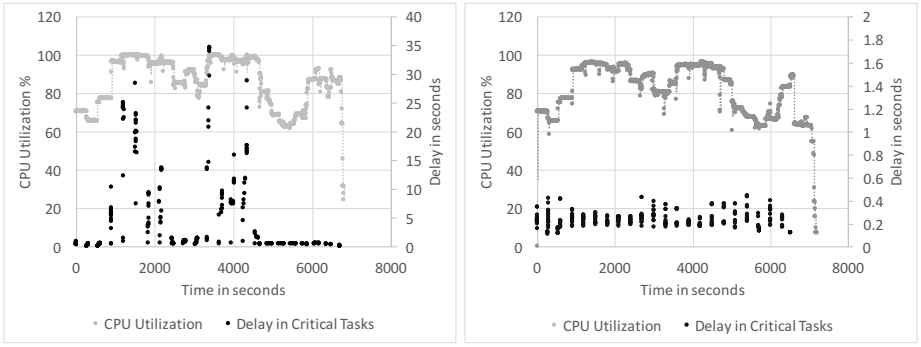


Fig. 10: Total CPU utilization of the 32-core physical machine while running the HCU workload and the corresponding delay in the critical tasks scheduled using a) cgroups and b) SCHED_DEADLINE.

of the delay in the critical tasks was not only considerably smaller but also remained nearly constant throughout the run, unaffected by the changes in the CPU load.

5 Related Work

In this section, we survey work that is related to resource isolation, cgroups, and the performance of scheduling policies such as SCHED_DEADLINE that enforce temporal isolation in the CPU.

Several enhancements have been proposed to the cgroups feature to improve its I/O resource management capabilities on NUMA machines with SSDs [24, 3]. Gao et al. [13], studied the different ways in which it is possible to circumvent the resource isolation enforced by cgroups and how an adversarial container can take advantage of the weaknesses inherent in cgroups to consume more resources than it is allowed to. From our work, too, it is evident that cgroups are unable to provide robust resource isolation for CPU time and background hogs can interfere with the tasks running in cgroups, thereby reducing their chances of getting the required share of CPU time.

PINE [20] is a performance isolation optimization strategy that dynamically adjusts disk space and I/O concurrency levels based on the performance requirements of tasks— either throughput or latency.

Vitucci et al. [31] compared the performance of SCHED_DEADLINE with non-resource isolating algorithms like SCHED_FIFO and SCHED_OTHER concluding that SCHED_DEADLINE provides an almost stable throughput compared to the other scheduling policies.

AIRS [16] is based on two algorithms — Flexible CBS and EDF with Window-constraint Migration (EDF-WM). The results show that AIRS is able to achieve

higher frame rates than SCHED_DEADLINE when running multiple movies due to the use of the EDF-WM algorithm. The work, however, does not touch on resource isolation or stability of CPU bandwidth allocation. It instead focuses on improving overall system utilization by allowing tasks to divide runtime among multiple cores and to share excess CPU bandwidth allocation.

In [9], the authors make use of the IRMOS real-time scheduler with KVM to provide stable performance to real-time applications running on VMs. The IRMOS scheduler allocates CPU time to a group of threads, such that the threads share the specified CPU time within a given period. Since our paper focuses specifically on the isolation of individual critical tasks, rather than VMs, we make use of the SCHED_DEADLINE policy without having to modify underlying system software. Our `sched_runtime` parameter estimation methodology is dynamic, and unlike the benchmarking in the aforementioned work, it does not require dedicated resources. This is because we use the SCHED_DEADLINE policy to provide the resource isolation necessary to estimate the CPU bandwidth requirement of critical tasks.

In [1], the authors present a hierarchical scheduler to execute the tasks in real-time control groups. The runqueues associated with the CPUs of a real-time cgroup are scheduled using the SCHED_DEADLINE policy. While the paper is related to cgroups and SCHED_DEADLINE, the objective of the work greatly differs from ours. Firstly, our work focuses on non-real-time workloads with unknown resource requirements and non-real-time cgroups which are scheduled using CFS. Secondly, the objective of our paper is to show that non-real-time cgroups, which are used as the default resource allocation mechanism in containers, provide poor resource isolation when compared to SCHED_DEADLINE.

PerfIso [14] is a performance isolation framework that uses a method known as CPU Blind Isolation that restricts the cores on which background tasks run to ensure that critical tasks always have some headroom. In our work, we use SCHED_DEADLINE to provide robust resource isolation to critical tasks such that the background tasks can utilize the remaining CPU bandwidth, without affecting the performance of the critical tasks. By allowing background tasks unrestricted access to the CPU bandwidth that has not been allocated to critical tasks, we improve the overall CPU utilization.

SCHED_DEADLINE is most effective when the CPU usage rate required by a task can be known or predicted accurately. Hence, our study complements other works [11, 6, 17], whose objective is to profile workloads and predict the dynamic resource requirements of tasks.

6 Conclusion

Cgroups are a kernel feature which enforce resource isolation between tasks in Linux-based containers. In this paper, we compare the resource isolation offered by cgroups and the SCHED_DEADLINE scheduling policy in the context of CPU bandwidth. While SCHED_DEADLINE has traditionally been used to guarantee temporal isolation of real-time tasks, we demonstrate that it can also

be used to secure guaranteed CPU bandwidth allocation for critical tasks, undeterred by the load on the CPU. Our experiments reveal that under cgroups, the critical tasks are unable to secure the required CPU bandwidth because of interference from co-located tasks. Since the tasks scheduled using cgroups do not get adequate CPU bandwidth in a timely manner, delays are introduced in the tasks' execution. However, when the critical tasks in the workloads are scheduled using SCHED_DEADLINE, the CPU bandwidth consumption of the tasks is nearly equal to the required CPU bandwidth, irrespective of the load on the CPU. As a result, the average delay of critical tasks scheduled using SCHED_DEADLINE was found to be 3x-40x smaller than cgroups. From this, we conclude that under high CPU load conditions, if critical tasks are scheduled using SCHED_DEADLINE rather than cgroups, they are assured better resource isolation, which results in stable resource allocations, smaller delays and predictable response times. We, therefore, recommend that existing execution environments for critical tasks be modified to exploit the resource isolation and the consequent benefits (such as dynamic estimation of CPU bandwidth requirement and reduction in task delays) offered by SCHED_DEADLINE.

7 Acknowledgement

We thank Akarsh Dsouza for his assistance with the implementation of the emulation and modelling of tasks in the cluster trace.

References

[1] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. "Container-based real-time scheduling in the linux kernel". In: *ACM SIGBED Review* 16.3 (2019), pp. 33–38.

[2] Luca Abeni and Giorgio Buttazzo. "Integrating multimedia applications in hard real-time systems". In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE. 1998, pp. 4–13.

[3] Sungyong Ahn, Kwanghyun La, and Jihong Kim. "Improving i/o resource sharing of linux cgroup for nvme ssds on multi-core systems". In: *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. 2016.

[4] Sean Kenneth Barker and Prashant Shenoy. "Empirical evaluation of latency-sensitive application performance in the cloud". In: *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. ACM. 2010, pp. 35–46.

[5] Enrico Bini et al. "Resource management on multicore systems: The ACTORS approach". In: *IEEE Micro* 31.3 (2011), pp. 72–81.

[6] Rodrigo N Calheiros et al. "Workload prediction using ARIMA model and its impact on cloud applications' QoS". In: *IEEE Transactions on Cloud Computing* 3.4 (2014), pp. 449–458.

[7] Paris Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[8] *Cgroups*. URL: `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`.

[9] Tommaso Cucinotta et al. “Providing performance guarantees to virtual machines using real-time scheduling”. In: *European Conference on Parallel Processing*. Springer. 2010, pp. 657–664.

[10] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware scheduling for heterogeneous datacenters”. In: *ACM SIGPLAN Notices*. Vol. 48. 4. ACM. 2013, pp. 77–88.

[11] Christina Delimitrou and Christos Kozyrakis. “Quasar: resource-efficient and QoS-aware cluster management”. In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 1. ACM. 2014, pp. 127–144.

[12] Dario Faggioli et al. “An EDF scheduling class for the Linux kernel”. In: *Proceedings of the 11th Real-Time Linux Workshop*. Citeseer. 2009, pp. 1–8.

[13] Xing Gao et al. “Houdini’s Escape: Breaking the Resource Rein of Linux Control Groups”. In: (2019).

[14] Călin Iorgulescu et al. “Perfiso: Performance isolation for commercial latency-sensitive services”. In: *2018 {USENIX} Annual Technical Conference*. 2018, pp. 519–532.

[15] Spencer Julian, Michael Shuey, and Seth Cook. “Containers in Research: Initial Experiences with Lightweight Infrastructure”. In: *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*. XSEDE16. Miami, USA: ACM, 2016, 25:1–25:6. ISBN: 978-1-4503-4755-6. DOI: `10.1145/2949550.2949562`. URL: `http://doi.acm.org/10.1145/2949550.2949562`.

[16] Shinpei Kato, Ragunathan Rajkumar, and Yutaka Ishikawa. “Airs: Supporting interactive real-time applications on multicore platforms”. In: *2010 22nd Euromicro Conference on Real-Time Systems*. IEEE. 2010, pp. 47–56.

[17] Yasaswi Kishore et al. “Qos aware resource management for apache cassandra”. In: *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*. IEEE. 2016, pp. 3–10.

[18] Juri Lelli et al. “Deadline scheduling in the Linux kernel”. In: *Software: Practice and Experience* 46.6 (2016), pp. 821–839.

[19] Jing Li et al. “Global EDF scheduling for parallel real-time tasks”. In: *Real-Time Systems* 51.4 (2015), pp. 395–439.

[20] Youhuizi Li et al. “PINE: Optimizing performance isolation in container environments”. In: *IEEE Access* 7 (2019), pp. 30410–30422.

[21] David Lo et al. “Heracles: Improving resource efficiency at scale”. In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 450–462.

[22] Jason Mars et al. “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations”. In: *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 248–259.

[23] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux Journal* 2014.239 (2014), p. 2.

[24] Junghi Min et al. “Cgroup++: Enhancing I/O Resource Management of Linux Cgroup on NUMA Systems with NVMe SSDs”. In: *Proceedings of the Posters and Demos Session of the 16th International Middleware Conference*. ACM. 2015, p. 7.

[25] Chandandeep Singh Pabla. “Completely fair scheduler”. In: *Linux Journal* 2009.184 (2009), p. 4.

[26] *Real-time group scheduling*. URL: `https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt` (visited on 04/12/2017).

[27] Charles Reiss, John Wilkes, and Joseph L Hellerstein. “Google cluster-usage traces: format+ schema”. In: *Google Inc., White Paper* (2011), pp. 1–14.

[28] Charles Reiss et al. “Heterogeneity and dynamicity of clouds at scale: Google trace analysis”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012, p. 7.

[29] Ankit Toshniwal et al. “Storm@ twitter”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156.

[30] Vinod Kumar Vavilapalli et al. “Apache hadoop yarn: Yet another resource negotiator”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[31] Carlo Vitucci et al. “A Linux-based Virtualized Solution Providing Computing Quality of Service to SDN-NFV Telecommunication Applications”. In: *Proceedings of the 16th Real Time Linux Workshop (RTLWS 2014)*. 2014, pp. 12–13.

[32] Ran Xu et al. “Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-located Workloads”. In: *Proceedings of the 19th International Middleware Conference*. ACM. 2018, pp. 146–160.