Design and Development of Real-Time Multi-Processor Bandwidth Control Mechanisms in General Purpose Operating Systems

Youcheng Sun

September 4, 2012

Abstract

Attentions have been being paid to extend general purpose operating systems with real time functionalities. In this thesis, a scheduling framework, which can control Central Processing Unit (CPU) bandwidth distribution on multi-processor platforms in a real-time way, is proposed in Linux. Under the framework, cpu bandwidths from different processors are reserved according to Constant Bandwidth Server (CBS) rules. Yet as for how to utilize the reserved bandwidths to schedule tasks in Linux, this is not the interest of the framework. They can be scheduled by policies used in the Linux system or scheduling algorithms that are implemented for a specific purpose. A subset of these tasks can get a portion of the reservation using the same rule. Furthermore, under the framework, scheduling policies can be applied in a controlled scale instead of the whole system.

In current implementation, both normal tasks and real-time tasks in Linux can work under the framework. Experimental results are not available now ...

Contents

Introduction

Linux is the most widely deployed open source general purpose Operating System. Linux flavors are used in a very wide range of systems and application area, from servers for cloud computing, to desktop computing, to tablet and smart phones, to industrial embedded systems.

One critical part of every Operating System is the task scheduler. For Linux, the scheduling system is also very critical given the wide range of application scenario in which this system is used. In facts, every scenario has different requirements and there is no single scheduler that can optimally satisfy them all.

Traditionally, one of the aim of Linux scheduling is to distribute CPU cycles fairly among tasks and task groups according to their relative importance. Another important requirement is to make *interactive applications* reactive to user input. There is no precise definition of an *interactive task*: the kernel uses heuristics to identify interactive tasks from batch processing tasks, in order to give higher priority to interactive tasks, thus reducing their response time.

One important class of applications consists of hard real-time or soft real-time systems. In order to work well, these applications have requirements in terms of the CPU cycles they receive, or on the time they need to react to external events. Hard and soft real-time applications needs to be guaranteed prior to their execution that they will be able to meet all their timing requirements (if hard real-time) or most of their timing requirements (if soft real-time). Unfortunately, such timing guarantee (or *real-time guarantee*) cannot be provided just by using an heuristic scheduler.

Examples of soft real-time applications are multi-media applications.

0.1 CPU reservations

In many situations, people prefer to distributing cpu power in a privileged and predictable way. For instance, to give 10% of the total CPU cycles to a set of tasks; furthermore, take half of this 10% and assign it to a subset of the tasks.

0.2 Multi-core systems

Nowadays, multi-core architectures are successfully used to boost computing capability for computing devices. Common computing systems with multiple processors are becoming mainstream: it is no surprise to see an embedded device with more than one CPU inside. Yet high speed processors and more cores rather than solving the CPU timing guarantee problem do bring more challenges for the developer. Also, no matter how powerful the platform is, there is always time when people need more. So, to fully utilize multi-core platform is also a good reason to manage computing power in a real-time way.

0.3 The Linux scheduler

In mainline Linux kernel, there are two so called real time scheduling policies: SCHED_FIFO and SCHED_RR. Tasks scheduled by them are called "real time(rt) tasks". They are required in POSIX standard for POSIX-compliant operating systems like Linux. Unfortunately, despite of the name, these two policies can only provide real time gurantee in very limited conditions. In mainline Linux, there exist two non real time mechanisms to control CPU bandwidth distribution: real time (rt) throttling and complete fairness scheduler (CFS) bandwidth control. In principle, they are the same technique working for different types of tasks.

0.4 Contributions of this thesis

The objective of our framework is to develope a multi-processor reservation mechanism in Linux that does not constraint with specific scheduling algorithms implemented in the kernel.

Our framework has two features:

- It can predictably distribute the cpu cycles of a multi-processor platform to a set of tasks and its subsets, without requirements for details how they are scheduled.
- Under the framework, scheduling policies can be applied in a fine-grained way.

In Linux, there is a scheduling system on each CPU. Different such per CPU scheduling systems construct the system level scheduling by task migration mechanisms among different CPUs. For the first time in Linux, our framework provides the opportunity to build extra scheduling systems besides these destinated with each CPU. We call the framework Open-Extension Container (OXC) scheduling framework. Open-Extension (OX) container is a new data structure in Linux that is introduced by our work. It is the fundamental element in the framework. Based on ox container structure, the concept "per oxc scheduling system", whose behaviour is the same as "per CPU scheduling system" in Linux, is given. Several per oxc scheduling systems cooperate and work as the "pseudo (Linux) system level scheduling".

In oxc scheduling framework, each ox-container can reserve an amount of bandwidth from a CPU through CBS rules [1]. The per oxc scheduling system based on it utilize this computing power to shcedule tasks as if working on a less powerful cpu. This is how oxc framework distribute reserved CPU cycles to tasks under it. Because the per oxc scheduling has the same behaviour as per CPU scheduling in Linux, general types of tasks can run using the reserved bandwidth. On multiple processor platforms, different OXCs can inpdependently reserve bandwidths from a subset of total CPUs and scheduling systems above them work together to imitate the behaviour of the Linux system level scheduling. The basic unit to apply a scheduling policy under OXC framework is an OX container.

0.5 State of the Art

There is work that extends Linux with real time capabilities to fullfill the timing guarantee requirement: RTAI[2], AQuoSA[3], `sched-deadline` patch[4], IRMOS real-time framework, RESCH and so on. Instead of modifying the system directly, RT-Xen tries to apply real time mechanisms in the hypervisor level(Xen).

Each work has its emphasis. Initially, our work is motivated by IRMOS.

Chapter 1

Background

1.1 The Constant Bandwidth Server theory

A Constant Bandwidth Server(CBS) is characterised by a budget c_s and an pair (Q_s, T_s), where Q_s is the maximum budget and T_s is the period of the server. $U_s = Q_s/T_s$ is called the server bandwidth (or utilisation). Such a server can be utilised to serve a set of tasks, which can be tasks with soft, hard or non real-time guarantee requirements. The original CBS algorithm

defines rules to reserve bandwidth on a single processor. In our work, we use a hard version of CBS.

For a specific server S, at any instant, a fixed deadline $d_{s,k}$ is associated with the server. A CBS is said to be active at time t if there are pending tasks and c_s is not 0; otherwise it is called idle. At any time, among all active servers, the one with earliest deadline is chosen. Then a served task of this server is picked to execute. CBS does not restrict the rule to pick up a particular task. For example, first in first out (fifo), rate monotonic scheduling and any user defined rule can be used. As the picked task executes, the server budget c_s is decreased by the same amount. When budget c_s reaches 0, the server become inactive. At each deadline point, the c_s will be recharged to Q_s and a new server deadline will be generated as $d_{s,k+1} = d_{s,k} + T_k$. Initially, $c_s = Q_s$ and $d_{s,0} = 0$. When a task arrives at time t and the server is idle, if $c_s \geq (d_{s,k} - t)U_s$, the server updates its deadline as

$d_{s,k+1} = t + T_s$ and c_s is recharged to maximum value Q_s.

Given a set of servers $\{S_0, S_1, ..., S_n\}$, if

$$\sum_{i=0}^{n} U_i \leq 1$$

then, every T_i time units, a server S_i can obtain Q_s time units to serve its tasks. In other words, U_i is the bandwidth a server S_i reserves from a cpu.

1.2 The Linux Scheduler

A scheduler is responsible for distributing CPU cycles to tasks in the system according to some scheduling algorithm. In Linux, tasks refer to a process or a thread and correspond to the data structure `struct task_struct`. The emphasis in this section is to clarify the relationships and connections among different scheduling components. As for how each scheduling algorithm in Linux is implemented, it's neither the interest of this section or oxc framework. To understand the Linux scheduling architecture is the first step to explore the oxc framework. For details about how linux schedulers work, people can read corresponding chapters in [5] [6].

1.2.1 Scheduling classes

Linux scheduling system adapts a modular design, and the basic modularity is a scheduling class, which is an instance of `struct sched_class`[1]. Scheduling algorithms are implemented as scheduling classes and a scheduling class is a modular scheduler(or simply called a scheduler). All modular schedulers composes the generic scheduler in Linux. The `struct sched_class` defines a set of interfaces which need to be realized in order to implement a scheduler in Linux. These methods are all scheduling operations a scheduler in Linux can perform. Each scheduler fullfills details behind the interface and carries out its specific scheduling behaviour. There are three scheduling

[1]Defined in include/linux/sched.h

classes in mainline Linux: rt_sched_class, cfs_sched_class and idle_sched_class[2]. Each scheduling class is responsible for scheduling a type of tasks. Tasks scheduled `cfs_sched_class` are called normal tasks. Tasks scheduled by `rt_sched_class` are called rt tasks. `idle_sched_class` deals with special idles tasks which does nothing and occupies the CPU when no rt or normal tasks need a CPU. Now, it's time to see the semantics of scheduling operations for a scheduler.

Listing 1.1: **Shceduling operations for a scheduler**

```
struct sched_class {
  const struct sched_class *next;
  void (*enqueue_task) (struct rq *rq, struct task_struct *p,
     int flags);
  void (*dequeue_task) (struct rq *rq, struct task_struct *p,
     int flags);
  void (*check_preempt_curr) (struct rq *rq, struct task_struct
     *p, int flags);
  struct task_struct * (*pick_next_task) (struct rq *rq);
  void (*put_prev_task) (struct rq *rq, struct task_struct *p);
        void (*set_curr_task) (struct rq *rq);
  void (*task_tick) (struct rq *rq, struct task_struct *p, int
     queued);
  ...
};
```

- `next`**:** Scheduling classes are linked in a chain, as shown in 1.1. Whenever a task is needed, the scheduler from the beginning to the end of the chain is checked and corresponding scheduling methods are called until a task is found. So, schedulers in front have higher priority to execute their tasks.

- `enqueue_task`**:** Called when a task enters a runnable state. The task is then enqueued into a runqueue, which is an instance of `struct rq`.

- `dequeue_task`**:** When a task is no longer runnable, this function is called to move corresponding task from a runqueue.

[2]Defined in rt.c, fair.c, and idle.c under kernel/sched directory respectively

- `check_preempt_curr`: This function checks if a task that entered the runnable state should preempt the currently running task.
- `pick_next_task`: This function chooses the task to run next. The newly picked up one can be the one currently occupting the CPU; in this case, no context switches are needed.
- `put_prev_task`: This is the last scheduling activity for a task before it gives up the executing opportunity on a CPU. In fact, it can happen that after this operation, the same task still occupies the CPU, as it is picked up again through `pick_next_task`.
- `set_curr_task`: This is the first scheduling operation for a task after a task is chosen to occupy the CPU.
- `task_tick`: This function is the most frequently called scheduling function. It is a good point to update the scheduling information, and it might lead to task switch.

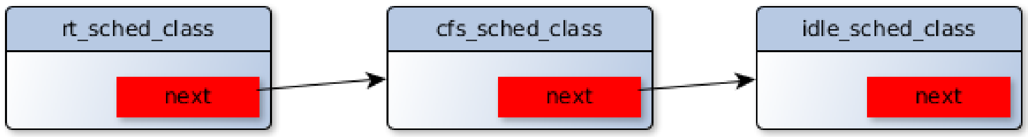


Figure 1.1: Scheduling classes in Linux

The basic scheduling unit in Linux is scheduling entity, which can represent both tasks and task groups. There are two kinds of scheduling entities [3]: cfs (scheduling) entities and rt (scheduling) entities. They are separately defined by `struct sched_entity` and `struct sched_rt_entity`. When we say a task is enqueued in a runqueue, more precisely, it is the task's (cfs or rt) scheduling entity is enqueued.

Listing 1.2: A task embeds scheduling entities

```
struct task_struct {
```

[3]Both are defined in include/linux/sched.h

```
  ...
  struct sched_entity se;
  struct sched_rt_entity rt;
  ...
};
```

Both cfs and rt entities are embedded in `struct task_struct`. For any task, its status can switch between a cfs task and a rt task through the system call `sched_setscheduler`.

When `CONFIG_FAIR_GROUP_SCHED` is set, cfs task grouping is enabled. And `CONFIG_RT_GROUP_SCHEED` is the kernel configuration for rt task group scheduling. A task group can contains both rt tasks and normal tasks, as shown in listing 1.3. For each CPU, a task group uses a rt entity and a cfs entity to represent its rt tasks and normal tasks. Each type of tasks inside a task group is scheduled independently by its own scheduling class.

Listing 1.3: **`A task group`**

```
struct task_group {
#ifdef CONFIG_FAIR_GROUP_SCHED
  /* sched_entity of this group on each cpu */
  struct sched_entity **se;
  ...
#endif
#ifdef CONFIG_RT_GROUP_SCHED
  /* sched_rt_entity of this group on each cpu */
  struct sched_entity **rt_se;
  ...
#endif
  ...
};
```

1.2.2 Runqueue centered scheduling

Every hook in `struct sched_class` deals with the data structure `struct rq`[4], which is called runqueue in Linux. We say that Linux scheduling is run-

[4]Defined in kernel/sched/core.c

queue centered. In Linux, the `struct rq` is a per CPU data structure; each cpu is associated with a runqueue. Although the name indicates, `struct rq` is not a queue. The `struct rq` contains a large amount of information. Its partial contents that are necessary for understanding this article is listed.

Listing 1.4: **The runqueue structure**

```
struct rq {
  ...
  unsigned long nr_running;
  struct cfs_rq cfs;
  struct rt_rq rt;
  struct task_struct *curr, *idle;
  u64 clock;
  u64 clock_task;
#ifdef CONFIG_SMP
  int cpu;
#endif
  ...
};
```

- `nr_running` specifies the number of runnable tasks having been enqueued in the runqueue.

- `cfs and rt` are two specific runqueues for `cfs_sched_class` and `rt_sched_class` respectively. In order to handle specific type of tasks, different schedulers define new type of runqueue data structures. When we say a task is enqueued into a runqueue, it is finally into its corresponding specific runqueue.Each task group has one cfs runqueue and rt runqueue per CPU to enqueue the tasks and sub task groups it contains in that CPU. Figure 1.5 adds this new information to the knowledge just introduced for a task group, as in figure 1.3. The default task group in the system points their cfs runqueue and rt runqueue to fields contained in the per CPU runqueue directly.

- `curr` points to the task currently running in this runqueue.

- `idle` points to a special idle task. This is the task occupying the CPU when no other tasks are runnable.
- `clock and clock_task` are time information kept by the runqueue. They updated by `update_rq_clock` method and some scheduling operation can rely them as time source.
- `cpu` tells the CPU of this runqueue.

Listing 1.5: **`Specific runqueue information within a task group`**

```
struct task_group {
#ifdef CONFIG_FAIR_GROUP_SCHED
  /* sched_entity of this group on each cpu */
  struct sched_entity **se;
  /* runqueue "owned" by this group on each cpu */
  struct cfs_rq **cfs_rq;
  ...
#endif
#ifdef CONFIG_RT_GROUP_SCHED
  /* sched_rt_entity of this group on each cpu */
  struct sched_entity **rt_se;
  struct rt_rq **rt_rq;
  ...
#endif
  ...
};
```

1.2.3 Completely Fair scheduler

Completely fair scheduler is implemented in `fair_sched_class`. Most tasks inside Linux are scheduled by completely fair scheduling class and are normal tasks, which can be further divided into three sub types given scheduling policies (`SCHED_NORMAL`, `SCHED_BATCH` and `SCHED_IDLE`[5]).

[5]This SCHED_IDLE policy is not related to idle_sched_class which aims to handle a special idle task.

CFS tries to distribute CPU cycles fairly to tasks and task groups according to their *weight*. A specific runqueue structure `struct cfs_rq` is provided to deal with normal tasks. Recall that an instance of such cfs runqueue is embedded in the per CPU runqueue and each task group holds a pointer to cfs runqueue on each CPU to store cfs tasks belonging to it. A little more details on cfs runqueue and cfs scheduling entity are followed.

Listing 1.6: **The cfs runqueue**

```
struct cfs_rq {
        unsigned long nr_running;
        struct rb_root tasks_timeline;
#ifdef CONFIG_FAIR_GROUP_SCHED
        struct rq *rq;
  struct task_group *tg;
#endif
  ...
};
```

- `nr_running` is the number of cfs tasks(entities) in this cfs runqueue.
- `tasks_timeline` is the root of the red-black tree [7] where all cfs entities enqued into this cfs runqueue is stored. This article will not go into details of the red-black tree mechanism, people only need to know it is an efficient way to sort and access data elements.
- `rq` is the per CPU runqueue that the task group `tg` is finally enqueued.
- `tg` is the task group that owns this cfs runqueue.

Listing 1.7: **The cfs scheduling entity**

```
struct sched_entity {
  ...
  struct cfs_rq *cfs_rq;
#ifdef CONFIG_FAIR_GROUP_SCHED
  struct cfs_rq *my_q;
#endif
  ...
};
```

- `cfs_rq` is where this entity is to be queued.
- `my_rq` is the cfs runqueue owned by this entity(group). Remember that a scheduling entity can also represent a task group.

Now there is enough information to show how different scheduling components (`sched_entity`, `task_struct`, `task_group` and `struct cfs_rq`) are related in completely fair scheduler.

In this case that cfs task group scheduling is enabled. the cfs scheduling scheme is shown in figure1.2. This is not a complete scheme: 1) Under a task group there could be sub groups, which behave as the task in the figure 2) In the system, there is a top group, which includes all tasks in the system by default; tasks in this group are enqueued in the cfs runqueue embedded in the per CPU runqueue directly.

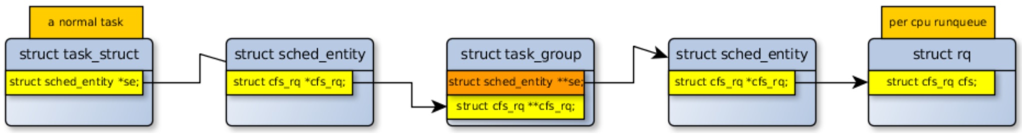


Figure 1.2: CFS scheduling when cfs group scheduling is enabled.

If cfs task group scheduling is not enabled, a task is directed to its per CPU runqueue by a *task_rq* marco. *task_rq* also works for rt scheduling when rt task group scheduling is not enabled. In fact, this `task_rq` can be used even in case rt and cfs task group scheduling are enabled. It just that when such task group schedling are set, normally the information in each element in the scheduling route is more important than simply returning a runqueue.

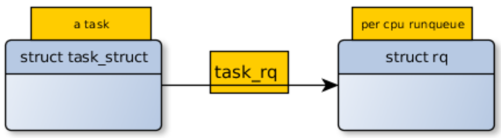


Figure 1.3: Scheduling scheme without group scheduling

We call the scheme in figure 1.2 and 1.3 *scheduling routes* in cfs scheduling. The route source is a task and the destination is a runqueue. The feature of a scheduling route is that if one component in the route is known, then other scheduling components in the direction towards the destination can be tracked. The concept of scheduling route is first invented in our work. Later you will see the theory behind oxc framework explores scheduling routes in Linux extensivly. We believe a new concept is deserved in formalizing the work for oxc framework.

1.2.4 Real time scheduler

Tasks with POSIX real time policies `SCHED_FIFO` and `SCHED_RR` are scheduled by the real time scheduling class `rt_sched_class` and are called rt tasks. Given figure1.1, rt tasks are always schedueld over normal tasks.

`SCHED_FIFO` implements a simple first-in, first-out scheduling algorithm. A running `SCHED_FIFO` task can only be preempted by a higher priority rt task. `SCHED_RR` is `SCHED_FIFO` with timeslices — it is a round robin algrithm. When a `SCHED_RR` task exhausts its timeslice, another `SCHED_RR` task of the same priprity is picked to run a timeslice, and so on. In either case, a rt task cannot be preempted by a lower priority task.

The rt scheduling class provides with a sub runqueue structure `struct rt_rq` to deal with rt tasks.

Listing 1.8: **The rt runqueue**

```
struct rt_rq {
  struct rt_prio_array active;
        unsigned long rt_nr_running;
#ifdef CONFIG_RT_GROUP_SCHED
        struct rq *rq;
        struct task_group *tg;
#endif
  ...
};

struct rt_prio_array {
  DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1);
  struct list_head queue[MAX_RT_PRIO];
};
```

All rt tasks with the same priority, let's say *prio*, are kept in a linked list headed by *active.queue*[*prio*]. If there is a task in the list, the corresponding bit in *active.bitmap* is set. All other fields have the same meaning as in cfs runqueue. Compare with the cfs scheduling entity, the following `struct sched_rt_entity` is self explanatory enough.

Listing 1.9: **The rt scheduling entity**

```
struct sched_rt_entity {
  ...
  struct rt_rq *rt_rq;
#ifdef CONFIG_RT_GROUP_SCHED
  struct rt_rq *my_q;
#endif
  ...
};
```

When `CONFIG_RT_GROUP_SCHED` is set, figure1.4 shows the scheduling route for rt scheduling. If rt task group scheduling is not enabled, still *task_rq* marco will be used.

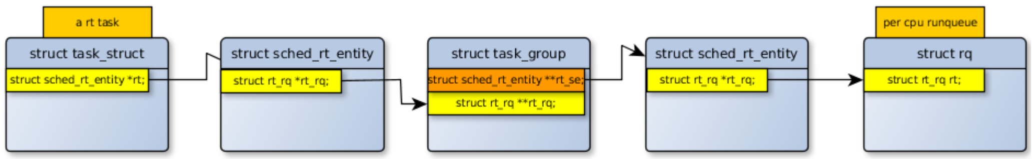


Figure 1.4: RT scheduling when rt group scheduling is enabled.

1.3 Related work

1.3.1 RT throttling

Enabling `CONFIG_RT_GROUP_SCHED` lets users explicitly allocate CPU bandiwidth to rt tasks in task groups. It uses the *control group* (cgroup) virtual file system. Each cgroup associates a set of tasks with a set of resources, called *subsystems*. For example *cpuset* subsystem is responsible for assigning a set of CPUs and Memory Nodes to tasks in a cgroup. Such tasks and resources can be further distributed in sub cgroups. Each cgroup is represented by a directory in the cgroup file system and a hierarchy of cgroups maps to a hierarchy of directories. In the directory, each mounted subsystem provides a list of files that are used as interfaces to control the allocation of a resource. Through mounting the *cpu* subsystem, two interfaces *cpu.rt_period_us* and *cpu.rt_runtime_us* are used to control the CPU bandwidth for rt tasks in each cgroup. That is, the total execution time of rt tasks in a cgroup on each CPU in time length *rt_period_us* cannot exceed *rt_runtime_us*. If this constraint is met, rt tasks would not be choosen to run on that CPU until a new period; we call such tasks be throttled.

No matter `CONFIG_RT_GROUP_SCHED` is set or not, in order to avoid rt tasks forever occupy thc CPU, there is a system wide setting that constraints rt tasks' execution through the /proc virtual file system :

/proc/sys/kernel/sched_rt_period_us

/proc/sys/kernel/sched_rt_runtime_us

This applies to all rt tasks in a system.

1.3.2 CFS bandwidth control

Basically, CFS bandwidth control is the same technique as RT throttling applying on normal tasks. It is a `CONFIG_FAIR_GROUP_SCHED` extension which allows the specification of the maximum CPU bandwidth available to normal tasks in a cgroup or cgroup hierarchy. The bandwidth allowed to a cgroup is specified using a quota(`cpu.cfs_quota_us`) and a period(`cpu.cfs_period_us`). By specifying this, normal tasks in a cgroup will be limited to `cfs_quota_us` units of CPU time within the period of `cfs_period_us`. Recall that in RT throttling 1.3.1, the reserved bandwidth through cgroup interfaces are applied in each CPU individually.

1.3.3 AQuoSA

The Adaptive Quality of Service Architecture composes two parts: a resource reservation scheduler an a feedback-based control mechanism. The scheduler uses CBS rules to rserve CPU bandwidth for a task, which is a rt task with

SCHED_RR policy in its Linux implementation. Given the error between the reserved computation and the amount of CPU cycles really consumed, the feedback controller adapts CBS reservation paramters to provide quality of service CPU allocation in the system. The control mechanism depends on CBS performance, not the scheduling details. That is, such a control mechanism can be applied to general CBS based scheduling. AQuoSA lacks considerations on multi-processor platform.

1.3.4 Schedule-deadline patch

The schedule-deadline patch for Linux kernel is being developed to extend current mainline Linux with a deadline-based scheduling method. In schedule-deadline, a new scheduling class (scheduler) is implemented and has highest priority among all scheduling classes. Tasks scheduled by this scheduling class are called sched tasks. A sched task is assigned deadlines according to CBS rules and scheduled in "earliest deadline first (EDF)" way.

1.3.5 IRMOS real-time framework

The rague name IRMOS comes from the European project "Interactive Real-time Multimedia Applications on Service Oriented Infrastructures". The IRMOS framework replace rt throttling mechanism in mainline Linux with real time CPU reservation(still CBS), and reuses the existing interfaces. So, users configure the cgroup interface as what we saw in rt throttling(1.3.1), the difference is that this time the CPU bandwidth is allocated in a guaranteed way. Also, new cgroup interfaces are added to assist reserved CPU power distribution in the cgroup hierarchy.

Chapter 2

Design of OXC Scheduling

2.1 Open-Extension Container Structure

Linux scheduling is runqueue, `struct rq`, centered and each scheduling class implements a set of interfaces to deal with the runqueue structure. In mainline Linux, `struct rq` is a per CPU structure. Each scheduling class defines its scheduling operations (enqueue, dequeue, etc.) with this per CPU runqueue. On Multiple processor platforms, tasks can migrate among different runqueues, also depending on behaviours defined in specific scheduler. In other words, on each CPU, a scheduling system is built up based on the associated runqueue. Different per CPU scheduling systems cooperate with each other by task migrating operations defiend by specific scheduling classes and construct the system level scheduling. On question raised here could be what if there are extra runqueues and how they can be utilized. Ideally, suppose there is one extra runqueue, each scheduler can still use it as scheduling parameter and a scheduling system can be built around it. If there are more than one extra runqueues, they can produce a pseudo system level scheduling system.

Extended from the above idea, a data structure named Open-Extension Container(OXC) is proposed in Linux kernel, shown in figure2.1. The ox container is designed as an abstract data structure; that is, any data structure contains a `struct rq` runqueue inside can be called the ox container. After

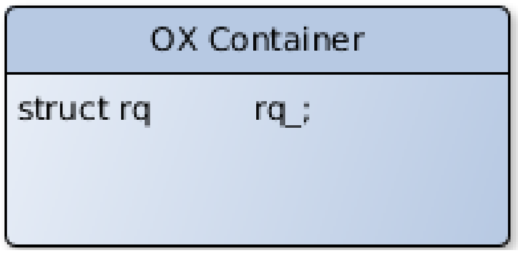


Figure 2.1: Open-Extension Container

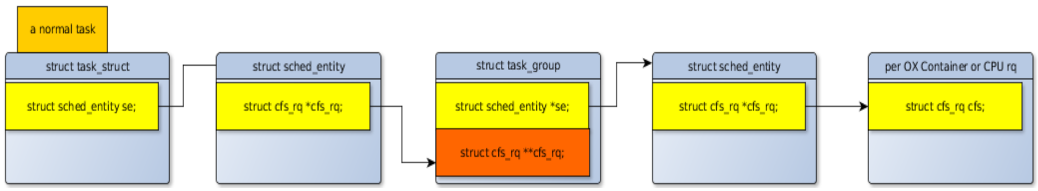
bringing this new data structure into the kernel, there is not only per CPU runqueues in teh system now, but also per oxc runqueues.

2.2 The oxc scheduling

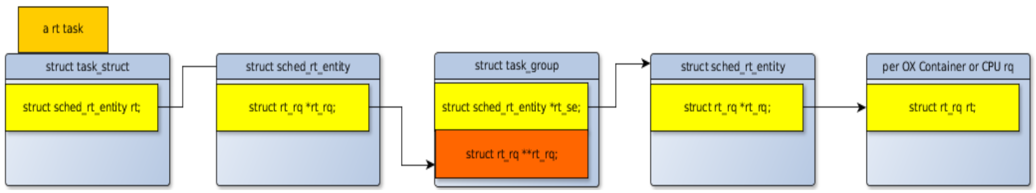
As there are extra per oxc runqueues besides per CPU ones in a system, they can be candidates passed as parameters to scheduling operations. From the standpoint of a scheduler, it manages a task over the runqueue according to implementation details in the scheduling class, and as long as a runqueue parameter is provided for its scheduling operations, the scheduler does not care whether it is associated with a CPU or from an ox container. So, as a oxc local runqueue is passed to hooks of scheduling classes, **tasks would enqueue, operate and dequeue on a per container runqueue**. This is called **the oxc scheduling**. The task enqueued in an ox container's local runqueue is called an oxc task. Of course, an oxc task can be a normal task or a rt task.

For tasks and scheduling classes, there is no difference between a per CPU and per container runqueue. This will be clearly shown when we see the scheduling route inside an ox container. Recall the path which relates each task with the per CPU runqueue it runs above. Figure 1.2 is the scheduling route for a task under cfs scheduling when fair task group scheduling is enabled; figure 1.4 is the shceduling route under rt scheduling when rt task group scheduling is enabled; figure 1.3 is the scheduling route for a task without task group scheduling. Now we are going to extend the scheduling routes in Linux for the oxc scheduling.

It's very natural to merge oxc scheduling in Linux scheduling routes. Figure 2.2(a) shows the extended sheduling route under cfs scheduling with CONFIG_FAIR_GROUP_SCHED enabled. The only difference for an oxc task happens in the terminal, where a per ox container runqueue replaces the per CPU counterpart. Figure 2.2(b) shows the scheduling route under rt scheduling with CONFIG_RT_GROUP_SCHED enabled; the situation is similar to cfs scheduling case we just saw. In fact, the scheduling route previous route can also lead a task to its per container runqueue. The same codes can still be used to find both kinds of runqueues.



(a) The route under cfs scheduling



(b) The route under rt scheduling

Figure 2.2: Scheduling routes for a task with task group scheduling enabled

In section 1.2.3, we introduce that when task group scheduling is not enabled, a macro task_rq is used to associate a task to its runqueue. The task_rq is defined as follows:

```
#define task_rq(p)                  cpu_rq(task_cpu(p))
```

The macro returns the associated rq for the CPU where the task is currently running on. So, when task group scheduling is not enabled, in order to merge oxc scheduling in the system, a new path leading a task to a runqueue is needed, shown in figure 2.3. Actually, this path already exists in Linux kernel. Just because in mainline Linux, there is no runqueue other than per CPU ones and people ignore to exploit it.

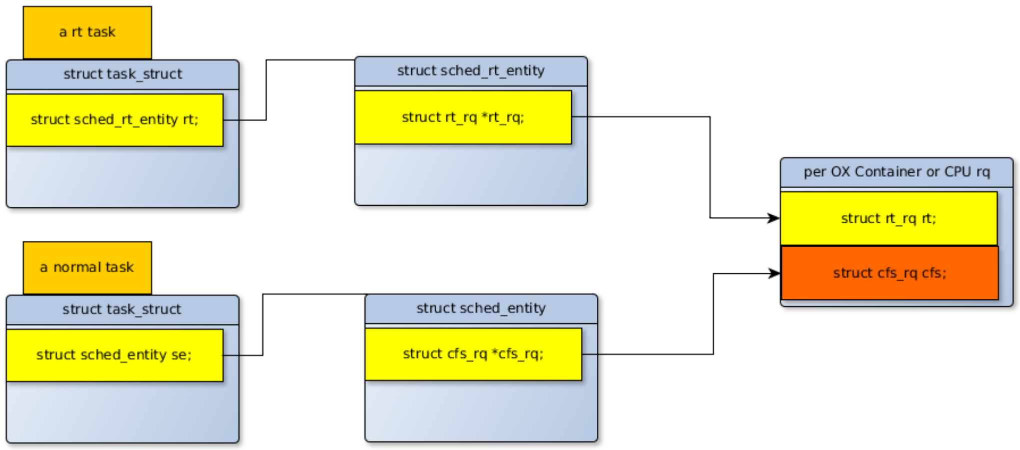


Figure 2.3: New scheduling route for tasks when task group scheduling is not enabled

The first feature of oxc scheduling is that it is compatibe with Linux original scheduling design. Tasks dealt by rt scheduler or cfs can naturally work under the oxc scheduling system. When there is new scheduling algorithm implemented in Linux kernel, like the *sched deadline* patch we mentioned before, the new scheduler has to fullfill details behind scheduling interfaces in `struct sched_class`. Again, for each shceduling class, they do not care the interface is passed a per CPU or per container runqueue as the parameter, and the new scheduling class can also work under oxc scheduling. So, the ox container structure is open to extension; this is where the name is from.

Based on per CPU scheduling, each scheduling operation defined in `struct sched_class` will affect all tasks in the CPU. The oxc scheduling provides another opportunity to apply a scheduling class in a fine grained scale. Now, the scale to apply a scheduler can be controlled in the unit of an ox container.

2.3 Movtivation for oxc scheduling framework

In section 1.3, we see that, based on Linux, CPU bandwidth control can be applied in the level of single tasks or task groups which are shceudled by some policies. Such controls can be real-time or non real-time. One obervation is that there does not exist a mechanism that can control CPU bandwidth for all kinds of tasks as a whole without requirement of a task' scheduling details.

Suppose there is an ox container, different schedulers can use it to enqueue, operate, and dequeue its tasks. If a fraction of CPU bandwidth can be assigned to this ox container, all kinds of tasks will use it as running on a less powerful machine. This is the oxc solution to the CPU bandwidth control for general types of tasks.

Based on this idea, we develop an oxc scheduling framework in Linux that can realize multi processor CPU reservation for tasks without requiring scheduling policies. In the following chapter, details on how to implement this oxc scheduling framework in Linux kernel are described.

Chapter 3

Development of OXC Framework

The oxc framework is still ongoing. Latest codes can be found in github[1]. The oxc framework is not a scheduler. Although it cooperates with different scheduling classes, its pure responsibility is managing the distribution of CPU power, which depends on modular schedulers to use it for scheduling tasks. Under oxc framework, we also call oxc scheduling oxc control since it is utilized to control CPU bandwidth reservation.

3.1 Implementation of ox container structure

An ox container `struct oxc_rq`, list 3.1, is defined in Linux kernel. In `struct oxc_rq` there are fields for reserving bandwidth from a CPU using CBS rules. The CPU reservation for oxc runqueue follows the implementation of CBS reservation for a rt runqueue in IRMOS framework. In the following context, an `struct oxc_rq` instance is also called an ox container, container, or oxc runqueue for the same meaning. An oxc runqueue also corresponds to a constant bandwidth server in CBS theory.

Listing 3.1: `struct oxc_rq`

```
struct oxc_rq {
```

[1]https://github.com/YIYAYIYAYOUCHENG/linux

```
    unsigned long oxc_nr_running;
    inx oxc_throttled;
    u64 oxc_deadline;
    u64 oxc_time;
    u64 oxc_runtime;
    ktime_t oxc_period;
    struct hrtimer oxc_period_timer;
    raw_spin_lock oxc_runtime_lock;
    struct rq rq_;
    struct rq *rq;
    struct rb_node rb_node;
};
```

- `oxc_nr_running` is the number of oxc tasks enqueued in the container's local runqueue. We say these tasks work in the ox container.
- `oxc_throttled` is set when an ox container runs out of its budget in a period.
- `oxc_deadline` is current deadline of this ox container, which is a server in CBS theory.
- `oxc_time` is currently consumed budget in a period.
- `oxc_runtime` and `oxc_period` are CBS parameters: `oxc_runtime` is maximum budget and `oxc_period` is the period.
- `oxc_period_timer` is timer which will activate at recharging points. If at some point, the value of `oxc_time` is larger than the value of `oxc_runtime`, then `oxc_throttled` should be set until `oxc_period_timer` fires to recharge the container.
- `oxc_runtime_lock` guarantees that the timing information of the oxc is updated in a consistant way.
- `rq_` is the local runqueue of the ox container.
- `rq` points to a per CPU runqueue and its CPU is where the ox container reserves bandwidth from.

- `rb_node` is used to put an ox runquneue in a red black tree. All oxc runqueues reserve bandwidth from the same CPU are sorted in a red-black tree. In this tree, an ox container's `oxc_deadline` value is used to order nodes.

For each CPU, there is a red-black tree which stores all oxc runeueues that reserve bandwidths in this CPU and orders them with their current deadline. The oxc runqueue with earliest deadline is dtored in the leftmost node. This tree is called the edf tree and is defined in `struct oxc_edf_tree`.

Listing 3.2: The EDF tree

```
struct oxc_edf_tree {
  struct rb_root rb_root;
  struct rb_node *rb_leftmost;
};
```

The pointer `rb_leftmost` helps fast access to the earliest deadline ox container in a CPU.

An ox container is responsible for reserving bandwidth from a CPU. Another data structure `struct hyper_oxc_rq` is defined to reserve bandwidth from multiple CPUs. A `struct hyper_oxc_rq` instance is called a hyper ox container.

Listing 3.3: The hyper ox container

```
struct hyper_oxc_rq {
  cpumask_var_t cpus_allowed;
  struct oxc_rq ** oxc_rq;
};
```

- `cpus_allowed` specifies the CPUs that are used to reserve bandwidth.
- `oxc_rq` is an array of ox containers to reserve bandwidth from CPUs specified in `cpus_allowed`.

3.2 Extensions on original data structures

Several new data structures have been imported in the kernel, yet the interfaces defined in `struct sched_class` does not change. Extensions are added in some original data structures in order to merge newly defined data structures in the system. Such extensions are not complex.

Listing 3.4: Extensions in `struct rq`

```
struct rq {
  ...
  int in_oxc;
  struct oxc_edf_tree oxc_edf_tree;
};
```

Two fileds are added in runqueue structure. The `in_oxc` is used to distinguish per CPU and ox container runqueue. As the name says, for an ox container's local runqueue, its `in_oxc` field is set. And `oxc_edf_tree` in a per CPU runqueue is the edf tree for a CPU to keep and sort ox containers.

The kernel configuration option `CONFIG_CGROUP_SCHED` is required by the oxc framework. This option allows to create arbitrary task groups using the "cgroup" pseudo filesystem. In current implementation of oxc framework, reservation is made for tasks in a control group. Tasks in a cgroup is represented in the `struct task_group` structure. So extensions also happen inside it.

Listing 3.5: Extensions in `struct task_group`

```
struct task_group {
  ...
  struct hyper_oxc_rq *hyper_oxc_rq;
  int oxc_label;
};
```

Because tasks in a cgroup can span multiple CPUs, `struct task_group` is a good place to put the hyper ox container. If a task group runs inside a hyper oxc, its `hyper_oxc_rq` points to that hyper container; otherwise, this field is NULL. As before, we call a task group inside a hyper oxc an oxc task group. There are several types of oxc task group: an oxc group whose father is not an oxc task group; an oxc task group whose father is an oxc group with different hyper ox container; an oxc task group with the same hyper ox container as its father. The `oxc_label` field is used to differ them. For non oxc task group, this field is not set.

When oxc scheduling is added in the kernel, there are two kinds of tasks in the system: oxc tasks and non oxc tasks. To difference between them is that oxc tasks work enqueued in an ox container's local runqueue and non oxc tasks work in a per CPU runqueue. So, from a task, its associated runqueue can be tracked according to the scheduling route in figure 2.2 or 2.3. As long as the runqueue is found, given its `in_oxc` field, the status of this runqueue and the task can both be fixed. Consider that such "is that an oxc task?" is often used in the framework, a `is_oxc_task` field is added in `struct task_struct` for efficient reason.

Listing 3.6: `is_oxc_task` field in `struct task_struct`

```
struct task_struct {
  int is_oxc_task;
  ...
};
```

When a task runs in an ox container, this new field is set.

3.3 To direct a task to a per ox container runqueue

In section 2.2, we show the scheduling routes when there exists oxc scheduling in a system. This section will introduce the details on how to build these scheduling routes.

3.3.1 To build the scheduling route in mainline Linux

In orde to schedule a task in a per oxc runqueue, the first thing is to associate this task with the local runqueue of an oxc. To understand this, let's first see how the system associate a task to a runqueue in mainline Linux, where there is only per CPU runqueues. This is done through the method `set_task_rq`.

Listing 3.7: To associate tasks with a per CPU runqueue in mainline Linux

```
void set_task_rq(struct task_struct *p, unsigned int cpu)
{
#ifdef CONFIG_FAIR_GROUP_SCHED
        p->se.cfs_rq = task_group(p)->cfs_rq[cpu];
        p->se.parent = task_group(p)->se[cpu];
#endif

#ifdef CONFIG_RT_GROUP_SCHED
        p->rt.rt_rq  = task_group(p)->rt_rq[cpu];
#endif
}
```

As demonstrated in list 3.7, codes inside `set_task_rq` build up the first part of the scheduling route when `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED` are set. When rt or cfs task group scheduling is enabled, each task is then directed to its task group. In mainline Linux, the second part of a scheduling route only directs a task group to the per CPU runqueues. Such paths are connected when the task group is created. Figure 3.1 shows the hint how a task group joins the scheduling route during its creation. In addition, now we know that a scheduling route is built backwards. In case that task group

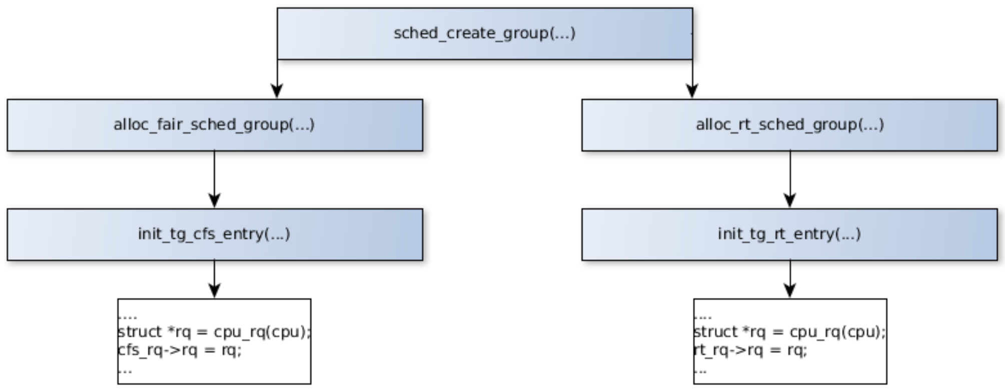


Figure 3.1: The creation of a task group in original Linux

scheduling is not enabled, recall the scheduling path where the `task_rq` leads a task to its per CPU runqueue directly.

3.3.2 To build the scheduling route in oxc enabled Linux

Now, we have seen the point when a task or task group joins the scheduling route in mainline Linux. These points are still time to fill elements in scheduling routes, figure ?? and 2.3, after oxc runqueues are imported in Linux.

Previously, the `set_task_rq` does not deal with the task that is not in a rt or cfs task group. This is because for tasks without group scheduling, the scheduling route `task_rq` is utilized. However, `task_rq` does not work for an oxc task to locate the per container runqueue. The functionality of `set_task_rq` is then extended to care about tasks with group scheduling.

Listing 3.8: The extended **`set_task_rq`**

```
void set_task_rq(struct task_struct *p, unsigned int cpu)
{
        struct task_group *tg = task_group(p);

#ifdef CONFIG_FAIR_GROUP_SCHED
        p->se.cfs_rq = tg->cfs_rq[cpu];
        p->se.parent = tg->se[cpu];
#else
        if(!tg->hyper_oxc_rq)
                p->se.cfs_rq = &cpu_rq(cpu)->cfs;
        else
                p->se.cfs_rq = &tg->hyper_oxc_rq->oxc_rq[cpu]->
                    rq_.cfs;
#endif

#ifdef CONFIG_RT_GROUP_SCHED
        p->rt.rt_rq  = tg->rt_rq[cpu];
        p->rt.parent = tg->rt_se[cpu];
#else
        if(!tg->hyper_oxc_rq)
                p->rt.rt_rq = &cpu_rq(cpu)->rt;
        else
```

```
                p->rt.rt_rq = &tg->hyper_oxc_rq->oxc_rq[cpu]->
                    rq_.rt;
#endif
        if (task_rq_oxc(p)->in_oxc == 1)
                p->is_oxc_task = 100;
        else
                p->is_oxc_task = 0;
}
```

When task group scheduling is enabled, there is no difference for setting a task's runqueue in both the mainline Linux and oxc enabled Linux. The interesting part happens when task group scheduling is not set. This time, given the task group is associated with a hyper oxc or not, the task is drected to a per CPU or oxc runqueue. This corresponds the first part of the scheduling route in figure 2.3. In the end of the method, `is_oxc_task` is configured. `task_rq_oxc` tracks the shceduling route to find the runqueue that a task is just associated. After this function call, the whole scheduling route without group scheduling and the first part of the scheduling route under group scheduling are built up.

In the mainline Linux, the end part of a scheduling route is built up when a task group is created. Within the oxc enabled kernel, things wil be a little more complex. In the oxc applied Linux, a task group can be associated with a hyper oxc and the contained runqueues in three cases:1. if its parent is associated with a hyper oxc, then when it is created it will inherit its parent's hyper ox container 2. the task group is explicitly attached to a hyper oxc 3. when the group's one ascendant task group is attached to a hyper oxc, its `hyper_oxc_rq` field will point to that hyper oxc too.

Corresponding to case 1, now when a task group is created, there will be a initilization routine for oxc scheduling. A sketch is shown in figure 3.2. The details of this routinue `alloc_oxc_sched_group` is shown below. Case 2 and 3 actually happen at the same time. To explicitly direct a task group to hyper ox containers, two methods `init_tg_cfs_entry_oxc` and `init_tg_rt_entry_oxc` will be used

Listing 3.9: OXC scheduling related initilization during task group creation

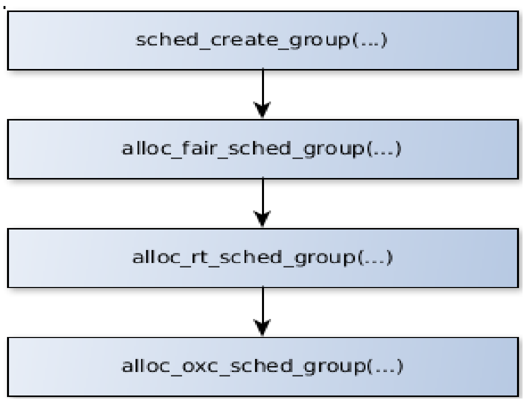

sched_create_group(...)
alloc_fair_sched_group(...)
alloc_rt_sched_group(...)
alloc_oxc_sched_group(...)

Figure 3.2: The creation of a task group in oxc enabled Linux

```
int alloc_oxc_sched_group(struct task_group *tg, struct
    task_group *parent)
{
        int i;

        tg->hyper_oxc_rq = parent->hyper_oxc_rq;
        if( parent->hyper_oxc_rq) {
                for_each_possible_cpu(i) {
#ifdef CONFIG_FAIR_GROUP_SCHED
                        tg->cfs_rq[i]->rq =
                                &tg->hyper_oxc_rq->oxc_rq[i]->
                                    rq_;
                        if( !parent->se[i] && tg->se[i])
                                tg->se[i]->cfs_rq =
                                      &tg->hyper_oxc_rq->oxc_rq[
                                          i]->rq_.cfs;
#endif
#ifdef CONFIG_RT_GROUP_SCHED
                        tg->rt_rq[i]->rq =
                                &tg->hyper_oxc_rq->oxc_rq[i]->
                                    rq_;
                        if( !parent->rt_se[i] && tg->rt_se[i])
                                tg->rt_se[i]->rt_rq =
                                       &tg->hyper_oxc_rq->oxc_rq
                                           [i]->rq_.rt;
#endif
                }
                tg->oxc_label = 100;
        }
        else
                tg->oxc_label = 0;

        return 1;
}
```

The alloc_oxc_sched_group handles oxc related initialization when a new task group is created. At first, the a newly created task group will inherit its parent task group's hyper container. If the parent is an oxc task group, the newly created task group will be directed to per oxc runqueues contained in the hyper oxc, which corresponds to the end part of the scheduling route. And the oxc_label for such child oxc task group is 100.

A task group can direct to a oxc local runqueue explicitly through init_tg_cfs_entry_oxc and init_tg_rt_entry_oxc. The structure in the two methods is silimar. Here we will have a look at init_tg_cfs_entry_oxc as an example.

Listing 3.10: To explicitlt direct a task group to an OXC local runqueue

```
static void init_tg_cfs_entry_oxc(struct task_group *tg,
          struct cfs_rq *cfs_rq,
          struct sched_entity *se, int cpu,
          struct sched_entity *parent,
          struct oxc_rq *oxc_rq)
{
  struct rq *rq = rq_of_oxc_rq(oxc_rq);
  init_tg_cfs_entry(tg, cfs_rq, se, cpu, parent);
  tg->cfs_rq[cpu]->rq = rq;
  if( !parent && se)
    se->cfs_rq = &rq->cfs;
}
```

Brief explanation on the parameters:

- tg is the task group to be dealt with.
- cfs_rq is the cfs runqueue where the cfs entity of this task group is enqueued.

- `se` is the cfs entity that represents `tg`.
- `cpu` specifies the cfs_rq pointer inside `tg` that will be redirected. This method redirects one cfs runqueue inside a `tg` to an oxc runqueue `oxc_rq` in each call. A hyper ox container can have more than one oxc runqueues iside and to associate a task group with such hyper ox container may require the `init_g_cfs_entry_oxc` be called multiple times.
- `parent` points to the parent cfs scheduling entity.
- `oxc_rq` contains the destinated runqueue.

As for codes, first `rq_of_oxc_rq` returns the oxc local runqueue. `init_tg_cfs_entry` initialize CFS related work for `tg`. Then, `tg` directs its local `cfs_rq` on `cpu` to the per oxc runqueue just got. After this function is invoked for every ox container in the hyper oxc, cfs tasks and task groups under `tg` work in the scheduling route which will lead them to an oxc local runqueue. When a task group is explicitly directed to a hyper ox container, the whole hierarchy of task groups under it will also be associated with this hyper ox container. This task group will be the top of this hierarchy, and it will be enqueud in the per oxc runqueue's embedded cfs runqueue directly. This is what the last `if` condition means. One amazing feature of this procedure to direct a task group to a per oxc is that tasks under a group is untouched.

3.4 Run tasks under OXC scheduling framework

As long as per oxc runqueue joins the scheduling route, the scheduling of tasks is compatible with modular schedulers in Linux. For an oxc task, just pass the task itself and its oxc local runqueue, instead of per CPU runqueue, to its corresponding scheduling class. And the scheduler will behave as usual. That is, this oxc scheduling framework is transparent to both schedulers and tasks.

Because we consider reserving CPU bandwidth for an ox container, there are scheduling operations that before or after passing parameters to them, the reservation information should be updated. For these kinds of scheduling operations, we adapt a relaying mechanism. The parameter is first passed to another function and after necessary actions, the scheduling operation is called inside this function. We say that such scheduling operations are encapsulated in oxc (scheduling) functions. Still scheduling details for a task are not the framework's work.

In order to fullfill real time guarantee, oxc tasks are always privileged to non oxc tasks. Among oxc tasks inside a container, the priority relation is the same as in Linux. The ox container itself can be considered as a virtual Linux system.

For each scheduling operation defined in `struct sched_class`, there are three situations for them to work under oxc scheduling framework: some are encapsulated inside oxc functions; some can work under the framework without directly; and others are not supported. The table 3.1 displays the three classes of scheduling operations. The naming convention for oxc functions which encapsulate a scheduling operation inside is appending the original name with `_oxc`. For example, the scheduling operation `task_tick` is called inside in `task_tick_oxc`. The enqueue and dequeue are two exceptions, they are enclosed in `enqueue_task_oxc` and `dequeue_task_oxc`.

3.4.1 To obtain the runqueue of a task

A method `tas_rq_oxc` is used to obtain the runqueue of a task. The runqueue retuned can be oxc local runqueue or per CPU one depending that whether the task is inside a container.

```
struct rq* rq_of_task(struct task_struct *p)
{
  struct rq *rq;

#ifdef CONFIG_FAIR_GROUP_SCHED
  rq = p->se.cfs_rq->rq;
```

Work inside an oxc function	Work without encapsulation	Unsupported
check_preempt_curr	yield_task	Others
pick_next_task	yield_to_task	
put_prev_task	task_waking	
set_curr_task	task_woken	
task_tick	set_cpus_allowed	
enqueue_task_rq	task_fork	
dequeue_task_rq	switched_from	
	switched_to	
	prio_changed	
	get_rt_interval	
	task_move_group	

Table 3.1: The way to handle a scheduling operation under the oxc framework

```
#else
  rq = task_rq_fair_oxc(p);
#endif
  return rq;
}
```

For any task, it has both cfs scheduling entity and rt entity. So the rt and cfs scheduling routes both exists for a task. Here, we utilize the cfs scheduling route. Given `CONFIG_FAIR_GROUP_SCHED` is set or not, the corresponding scheduling routes are explored to track the runqueue.

3.4.2 To enqueue an oxc task

When an oxc task arrives, besides enqueue it in the oxc local runqueue, the ox container information may be updated if necesary.

```
void enqueue_task_oxc(struct rq *rq, struct task_struct *p, int
    flags)
{
  struct oxc_rq *oxc_rq = oxc_rq_of_task(p);
  struct rq *rq_ = rq_of_oxc_rq(oxc_rq);

  /* Update the local runqueue' clock. */
  update_rq_clock(rq_);
```

```
    /*
    * Enqueue the task into the local runqueue
    * by its scheduling class.
    */
    p->sched_class->enqueue_task(rq_, p, flags);

    inc_oxc_tasks(p, oxc_rq);
    enqueue_oxc_rq(oxc_rq);
}
```

oxc_rq_of_task tracks the scheduling route and returns the ox container of an oxc task. Then local runqueue's time information is updated. Although the ox container does not care about scheduling details of tasks inside it, tasks are enqueued in its local runqueue and may rely the runqueue's time information. Then we see all scheduling details are dealt by a task's scheduling class as the enqueue_task of the scheduling class is called with the task and local runqueue. The inc_oxc_tasks method is simple.

```
static inline void inc_oxc_tasks(struct task_struct *p, struct
    oxc_rq *oxc_rq)
{
        oxc_rq->oxc_nr_running ++;
}
```

Until now, the oxc task has been put in the local runqueue. The coming of an oxc task in a container may change the relation among ox containers in the same edf tree. This is the work of enqueue_oxc_rq method.

```
static void enqueue_oxc_rq(struct oxc_rq *oxc_rq)
{
        int on_rq;

        on_rq = oxc_rq_on_rq(oxc_rq);

        BUG_ON(!oxc_rq->oxc_nr_running);
        BUG_ON(on_rq && oxc_rq_throttled(oxc_rq));

        if( on_rq) {
                /* Already queued properly. */
                return;
        }
        /* We do not put a throttled oxc_rq in the edf tree. */
        if( oxc_rq_throttled(oxc_rq))
                return;

        oxc_rq_update_deadline(oxc_rq);
        __enqueue_oxc_rq(oxc_rq);
}
```

on_rq tells if the container oxc_rq is in a edf tree. BUG_ON is a Linux kernel macro. If the condition it checks is true, then the kernel will crash! Because we just put a task in the local runqueue, so the first BUG_ON should be passed. When an ox container runs out of its budget, it should be moved from the edf tree, this is what the second BUG_ON checks. Now we pass the two BUG_ONs. If the oxc_rq is already on edf tree or moved from the tree because of exhausting budget, nothing to be done. There are two conditions for an ox container outside an edf tree: it is throttled or it is empty. This means a task just joins an empty container. Recall the CBS rules ”when a task arrives and the server is idle, update the deadline if necessary”. This is what oxc_rq_update_deadline does. When it comes to CPU reservation, an ox container corresponds to a constant bandwidth server in CBS theory. texttt_enqueue_oxc_rq is quite a mechanical procesure to put an oxc runqueue in an edf tree.

3.4.3 To dequeue an oxc task

dequeue_task_oxc is the opposite method of enqueue_oxc_rq.

```
static void dequeue_task_oxc(struct rq *rq, struct task_struct *
    p, int flags)
{
        struct rq *rq_ = task_rq_oxc(p);
        struct oxc_rq *oxc_rq = container_of(rq_, struct oxc_rq,
             rq_);

        /* Update the local runqueue. */
        update_rq_clock(rq_);
        /*
```

```
         * Dequeue the task from the local runqueue
         * by its scheduling class.
         */
        p->sched_class->dequeue_task(rq_, p, flags);

        dec_oxc_tasks(p, oxc_rq);
        dequeue_oxc_rq(oxc_rq);
}
```

The structure of dequeue_task_oxc are the same as enqueue_task_oxc: local runqueue's time information is updated, local runqueue and task is relayed to the corresponding scheduler, and the task number is decreased. When an oxc task leaves the container, it is the time to check if the oxc is empty or not, which is done in dequeue_oxc_rq.

```
static void dequeue_oxc_rq(struct oxc_rq *oxc_rq)
{
        int on_rq;

        on_rq = oxc_rq_on_rq(oxc_rq);
        /*
         * Here we do not expect throttled oxc_rq to be in the
             edf tree.
         * Note that when an oxc_rq exceeds its maximum budget,
         * it is dequeued via sched_oxc_rq_dequeue().
         */
        BUG_ON(on_rq && oxc_rq_throttled(oxc_rq));
        /*
         * If an oxc_rq is not in the edf tree, it should be
             throttled or
         * have no tasks enqueued.
         */
        BUG_ON(!on_rq && !oxc_rq_throttled(oxc_rq) && !oxc_rq->
            oxc_nr_running);

        if( on_rq && !oxc_rq->oxc_nr_running) {
                /* Dequeue the oxc_rq if it has no more tasks.
                    */
                __dequeue_oxc_rq(oxc_rq);
                return;
        }
}
```

The comments are explainable enough. `__dequeue_oxc_rq` removes a oxc runqueue from the edf tree.

3.4.4 To check the preemption

When a task wakes up from sleeping or is created, the scheduler will check if it can preempt current running task in the same CPU. If the current task is an oxc task and the waking task is not an oxc task, the later cannot preempt the current one. If both are non oxc tasks, Linux already has methods to check. So here we only interest in the case when the waking task is an oxc task.

```
static inline int
check_preempt_oxc_rq(struct task_struct *curr, struct
   task_struct *p, int flags)
{
        struct oxc_rq *oxc_rq = oxc_rq_of_task(p);
        struct oxc_rq *oxc_rq_curr = oxc_rq_of_task(curr);
        const struct sched_class *class;

  if(oxc_rq_throttled(oxc_rq)
    return 0;
        /*
         * Tasks from a unthrottled oxc_rq always has a higher
            priority
         * than non oxc tasks.
         */
        if( !oxc_rq_curr)
                return 1;

        /* Both p and current task are in the same oxc_rq. */
        if( oxc_rq_curr == oxc_rq) {
                if( p->sched_class == curr->sched_class)
                        curr->sched_class->check_preempt_curr(
                                      &oxc_rq->rq_, p, flags);
                else {
                        for_each_class(class) {
                                if( class == curr->sched_class)
                                        break;
                                if( class == p->sched_class) {
                                        resched_task(curr);
                                        break;
                                }
                        }
                }

                return 0;
        }

        /* p and current tasks are oxc tasks from different ox
           containers. */
        return oxc_rq_before(oxc_rq, oxc_rq_curr);
}
```

`curr` and `p` are the current task and wakeing task respectively. If task `p`'s container is throttled, it cannot preempt currently running task. Otherwise, if `curr` is not an oxc task, `p` has higher pripriy and preempt `curr`. When two tasks are contained in the same oxc runqueue: if they are even in the same scheduling class, it's the modular scheduler's responsibility to decide given; otherwise, the task whose scheduling class has higher priority in the scheduler chain is chosen to run. In the last case, they two are from different containers. Now, `oxc_rq_before` checks if a contaienr's deadline is before another's deadline. The one with smaller deadline will run.

3.4.5 To pick up an oxc task

When to pick a most eligible task to run, oxc tasks should be checked first. If there is no eligible oxc tasks, then non oxc tasks are considered. `pick_next_task_oxc` is responsible for choosing the most eligible oxc task in a CPU.

```
static struct task_struct* pick_next_task_oxc(struct rq *rq)
{
        struct oxc_rq *oxc_rq;
        struct rq *rq_;
        struct task_struct *p, *curr;
        const struct sched_class *class;
        /* This clock update is necessary! */
        update_rq_clock(rq);
        update_curr_oxc_rq(rq);
        oxc_rq = pick_next_oxc_rq(rq);
        if( !oxc_rq)
                return NULL;

        rq_ = rq_of_oxc_rq(oxc_rq);

        update_rq_clock(rq_);

        for_each_class(class) {
                if( class != &idle_sched_class) {
                        p = class->pick_next_task(rq_);
                        if( p) {
                                rq_->curr = p;
                                return p;
                        }
                }
        }

        return NULL;
}
```

There are one thing to note here: not only local runqueue's clock is updated, but also the per CPU runqueue's clock is updated here. This is because the reservation time of an ox container is counted using the per CPU runqueue's clock and to keep the clock on time for the container's use, here it is updated. `pick_next_oxc_rq` picks the ox runqueue with latest deadline in a CPU. And along the scheduling class chain, each scheduler uses its own way trying to find the most eligible task in the local runqueue. Another import method here is `update_curr_oxc_rq`. The budget comsumption really happens here.

```
static void update_curr_oxc_rq(struct rq *rq)
```

```
{
        struct task_struct *curr = rq->curr;
        struct oxc_rq *oxc_rq = oxc_rq_of_task(curr);
        u64 delta_exec;
        /*
         * If current task is not oxc task, simply return.
         */
        if( !oxc_rq)
                return;

        delta_exec = rq->clock - oxc_rq->oxc_start_time;
        oxc_rq->oxc_start_time = rq->clock;
        if( unlikely((s64)delta_exec < 0))
                delta_exec = 0;

        raw_spin_lock(&oxc_rq->oxc_runtime_lock);

        oxc_rq->oxc_time += delta_exec;
        if( sched_oxc_rq_runtime_exceeded(oxc_rq)) {
                resched_task(curr);
        }

        raw_spin_unlock(&oxc_rq->oxc_runtime_lock);
}
```

`update_curr_oxc_rq` updates the runtime information of an oxc runqueue. If the budget in current period is exhausted, the current task needs to be resched. the local spinlock `oxc_runtime_lock` protects the update of runtime from interleave.

```
static int sched_oxc_rq_runtime_exceeded(struct oxc_rq *oxc_rq)
{
        u64 runtime = sched_oxc_rq_runtime(oxc_rq);
        u64 period = sched_oxc_rq_period(oxc_rq);

        /*
         * If the runtime is set as 'RUNTIME_INF',
         * the ox container can run without throttling.
         */
        if( runtime == RUNTIME_INF)
```

```
                return 0;

        /*
         * If the runtime to be larger the the period,
         * the ox container can run without throttling.
         */
        if( runtime >=period)
                return 0;

        /* There is still budget left. */
        if( oxc_rq->oxc_time < runtime)
                return 0;
        /*
         * The reservation in a period has been exhausted,
         * to set the throttling label, remove the oxc_rq
         * from the edf tree and start the recharging timer.
         */
        else {
                oxc_rq->oxc_throttled = 1;
                sched_oxc_rq_dequeue(oxc_rq);
                start_oxc_period_timer(oxc_rq);

                return 1;
        }
}
```

Inside `sched_oxc_rq_runtime_exceeded`, at first a series of non exceeded conditions are checked, which is easy to understand. The last *else* statement deals with the case tha tthe container is throttled: the `oxc_throttled` label is set, the oxc runqueue is removed from the edf tree and the timer is set and will fire at the next deadline to recharge the budget.

3.4.6 put_prev_task operation

The scheduling operation is called when the currently running task is possible to be replaced. It performs some conclusion work for the task. Although the currently running task may keep running without being preempted. If currently running task is an oxc task, when `put_prev_task` is called, this is also a point to update the ox container information.

```
static void put_prev_task_oxc(struct rq* rq, struct task_struct
    *p)
{
        struct rq *rq_ = task_rq_oxc(p);

  update_rq_clock(rq_);
        update_curr_oxc_rq(rq);

        p->sched_class->put_prev_task(rq_, p);
}
```

Now, when a put_prev_task operation is needed and the currently running task is an oxc task, instead of calling the put_prev_task defined in a scheduling class directly, our put_prev_task_oxc encapsulation will be called first then ox container local runqueue and current task will be relayed to the corresponding scheduler.

3.4.7 set_curr_task operation

The above put_prev_task is the last scheduling operation before a task gives up a CPU (of course, if it is chosen again immediately, it can still occupy the CPU). The set_curr_task is the first scheduling operation a task performs when it is chosen to use the CPU. If the newly chosen task is an oxc task, it is also the point to prepare the ox container for time updating.

```
static void set_curr_task_oxc(struct rq *rq)
{
        struct task_struct *curr = rq->curr;
        struct rq *rq_ = task_rq_oxc(curr);
        struct oxc_rq *oxc_rq;

        oxc_rq = container_of(rq_, struct oxc_rq, rq_);

        oxc_rq->oxc_start_time = oxc_rq->rq->clock;

        update_rq_clock(rq_);
        curr->sched_class->set_curr_task(rq);
}
```

One thing to note is that inside this encapsulation, the per CPU runqueue parameter is passed to the task's corresponding scheduling class. This is because `set_curr_task` operation updates the infomation in the scheduling route except for the runqueue itself. So, there is no difference to pass which runqueue. The real reason is that, there is a possible insconsistent state. Initially, the current task in ox container local runqueue is NULL[2], which is not a special idle task from scheduling class `sched_idle`. Because `set_curr_task` is called even before the task is enqueued in the runqueue(maybe oxc local one). And this will cause problems.

3.4.8 task_tick operation

The `task_tick` operation is the most frequently called scheduling operation and is used to update the task's timing information. So, if the task parameter in this method is oxc type, this is the point to update the confainer's information and local runqueue's clock.

```
static void task_tick_oxc(struct rq *rq, struct task_struct *p,
    int queued)
{
        struct rq *rq_ = task_rq_oxc(p);

        update_curr_oxc_rq(rq);
        update_rq_clock(rq_);

        p->sched_class->task_tick(rq_, p, queued);
}
```

3.5 SMP support in oxc framework

An ox container based scheduling system on a CPU has no difference with the per CPU based scheduling system. Each ox container works independently and they are arranged in one hyper container. For a hyper container, tasks

[2]This will be fixed in future work

are partitioned to each ox container and task migration or load balancing between different ox contaienrs in a hyper container are not realized now.

3.6 User interfaces provided with OXC framework

Currently, oxc framework provides users with cgroup interfaces. The CPU reservation functionality is realized through *cpu* cgroup subsystem. This *cpu* cgroup subsystem is for CPU bandwidth control in Linux. RT throttling, fairness group scheduling and CFS bandwidth control are all realized through it. The following describes how to use the OXC framework and the CPU number is assumed to be 2.

To mount the *cpu* cgroup subsystem(in directory /cgroup):

```
#mount -t cgroup -ocpu none /cgroup
```

To create a cgroup for CPU reservation :

```
#mkdir -p /cgroup/cg
```

Observe the files inside `/cgroup/cg` directory, there are one new file `cpu.oxc_control` which is the interface to control CPU bandwidth in oxcframework. However, if one tries to see the content of this file:

```
#cat /cgroup/cg/cpu.oxc_control
```

It will display nothing. This is because by default the oxc reservation is disabled initially until the reservation is triggered for the first time. The reservation is triggerred by setting reservation parameters. To reserve bandwidth in a CPU, three parameters should be specified: CPU number, maximum budget and period. For example:

```
#echo 0 100000/1000000 1 20000/500000 > cg/cpu.oxc_control
```

This command reserve 100ms every 1s on CPU 0 and 20ms every 500ms on CPU 1 to cgroup `cg`. 0 and 1 are CPU numbers. 100000 and 20000 are maximum budgets and 1000000 and 500000 are periods. The unit for budget and period value is millisecond. This follows the convention in *cpu* cgroup subsystem. So tasks inside this cgroup and its further sub cgroups will run using the above reserved CPU bandwidth. Now we can say `cg` is contained in a hyper ox container.

To "cat" the content of the `cpu.oxc_control` interface under directory `/cgroup/cg`:

```
#cat /cgroup/cg/cpu.oxc_control
```

The above parameters set will be displayed:

```
0 100000/1000000 1 20000/500000
```

Reservation parameters can be configured in the same way. Furthermore, there is no need to configure reservation parameters for two CPUs at the same time. Suppose in some point, users decide to decrease the reservation from CPU 1, they can simply use the following command.

```
#echo 1 20000/1000000 > cg/cpu.oxc_control
```

The reservation on CPU 1 is dereased to 20ms every 1s and the reservation on another CPU is not interfered.

To create a sub cgroup for cgroup `cg`:

```
#mkdir -p /cgroup/cg/cg_0
```

The cgroup `cg_0` is contained in the same hyper container as its parent. Try to "cat" the `/cgroup/cg/cg_0/cpu.oxc_control`:

```
#cat /cgroup/cg/cg\_0/cpu.oxc\_control}
```

An error message will be returned. This is because for a cgroup family contained in a hyper container, only the top cgroup has the right to browse and modify reservation parameters.

People can move tasks to cgroup `cg` and texttt`cg_0`. For example

```
#echo 1982 > cg/tasks
#echo 1983 > cg\_0/tasks
...
```

This moves task with `pid` 1982 and 1983 to cgroup `cg` and `cg_0` respectively. Tasks can be rt tasks or normal tasks. All tasks inside an ox container behave as working on a virtual Linux system and utilize the reserved bandwidth.

Note here we use ”ox container”, not ”hyper ox container”, because in temporary iplementation, tasks inside a hyper ox container are partitioned into each CPU.

The oxc tasks can move between different cgroups contained in the same hyper container. They can move between ox containers and hyper containers. They can also leave an ox container and return to a non oxc task.

Until now, how to reserve CPU bandwidth under oxc framework is introduced. Now let's see how to distribute the reserved CPU power. ALthough users cannot browse reservation parameters in cgroup `cg_0`, they can indeed set reservation parameters for `cg_0`, which will trigger reserved power redistribution.

```
#echo 0 100000/2000000 1 20000/1000000 > cg/cpu.oxc_control
```

After this, a new hyper container with reservation parameters 1000000/1500000 on CPU 0 and 20000/1000000 on CPU 1 will be created and `cg_0` and its decendant cgroups will be associated with it. Now, although `cg` and `cg_0` are still in the same hierarchy in the cgroup directory observation, they are indeeded contained in two different hyper containers. The ideal effct of the above command should also include that the reserved bandwidth by `cg` need to decrease the same vaue as distributed to `tg_0`. This behaviour is still missed in current prorotype implementation of oxc framework. Yet this is indeed implementable. Also, the total reserved bandwidth in the system should not be more one in each CPU; this condition test is not realised either.

3.7 Cooperation with scheduling mechanisms inside Linux

When `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED` are not set; that is, task grouping is not enabled. Tasks work in an ox container just as the oxc scheduling theory explains. In fact, in this case, oxc framework can work as RT throttling and CFS bandwidth control in a real time behaviour. The result of IRMOS real time scheduler can also be achieved by our framework. Consider future scheduling classes that are possible to be merged in Linux kernel. For example deadline tasks in `schedule_deadline` patch, they do not have task group scheme. The oxc framework provide them a real time way to group their tasks and because of the "open to extension" feature for ox container, to merge them with oxc framework is natural.

When fairness task group shceduling is enabled, task groups under the same hyper ox container follow the rules of fairness task grouping and share the reserved CPU power. Fairness task group scheduling is applied in different areas independently: each hyper ox container is an area, outside ox containers there is the other area. Inside one hyper ox container, bandwidths are reserved and task groups inside this hyper ox container share the reserved computation power arrording to the fairness task group scheduling rules.

When `CONFIG_RT_GROUP_SCHED` is set, RT throttling is enabled. Let's first analyze the possible result when RT throttling is applied in a hyper ox container. Suppose inside an container, Q/T is the bandwidth reserved from the CPU and RT throttling sets parameters as $Q^{'}/T^{'}$ and $Q^{'}/T^{'} \leq Q/T$. The ideal behaviour for such an ox container would be : the container distributes reserved CPU power to tasks inside it; and rt tasks inside a group will be throttled when $Q^{'}$ units of CPU cycles are exhausted in period $T^{'}$, then non rt tasks can run. However, there are also other possibilities. For example, $Q^{'}/T^{'} = 1/10$ and $Q/T = 10/50$. Suppose there are other higher priority containers in the same CPU and in one period the example container get the right to use the CPU on the last 10 units of CPU cycles in its period. RT tasks inside thsi container immediately, yet after one unit, they are throttled and has to give up. So, during the whole period, rt tasks only run 1 unit over 50 unit of CPU cycles. The RT throttling result inside a hyper ox container is not stable. Even we set the period parameter in RT throttling the same as its container's, because of RT throttling using different *hrtimer* from oxc reservation, the unsyhcronization between can also cause even more complex situation.

In a word, to merge RT throttling directly inside container is not efficient. One possible solution is to count the time consumption in rt throttling using the same timer, this basically means to implement a copy of RT throttling in oxc framework itself. In such case, if some constraints are put, like the RT throttling period should equal to its container's period, we can expect predictable behaviour. Another solution is simply disable RT throttling inside container, because oxc scheduling framework itself can perform the same result in a real time way as RT throttling. In current implementation, solution one is used when `CONFIG_RT_GROUP_SCHED` is set, and result is not satisfiable.

The behaviour when CFS bandwidth control is enabled. We predict the behaviour should be similiar to what we see in RT throttling case.

Among cgroup subsystems, there is one `cpuset` also effecting scheduling behaviour in a cgroup. After `cpuset` cgroup subsystem is mounted in a cgroup, there is an interface `cpuset.cpus` appearring in the dorectory. Which can control which CPU the tasks inside this cgroup can use. For example,

```
#echo 1 > cpuset.cpus
```

This will result tasks inside this cgroup can only run on CPU 1. In oxc scheduling framework, we have the concept of hyper ox container, which control which CPUs tasks inside hyper container can run. So, this idea is compatible with `cpuset` cgroup subsystem. However, until now the two work independently; future work to bridge the two will make the system more efficient.

Chapter 4

Experiments

The overhead introduced by oxc framework includes three parts:

- The time required to execute codes brought oxc functions.
- The context switches introduced by the oxc framework.
- The degradation of modular schedulers' performance under oxc framework.

The third item is caused by implementation limitation can be minimized or removed by improving implementation details. For example, to access the per CPU runqueue in Linux is an optimized operation. However, the access to a per ox container is not so efficient, this will give a penalty when scheduler works inside an ox container.

There are two experiments carried out to evaluate the overhead in the oxc framework. In experiment A, the code execution time of frequently invoked oxc functions is measured. In experiment B, the overall overhead of oxc control is estimated through comparisons with rt throttling and cfs bandwidth control.

Inside each oxc function, there is a scheduling operation inside and codes to regulate bandwidth reservation. The cost of these functions is the main interest in experiment A. Current oxc framework implementation is still a prototype. Some kernel features are not considered under the framework yet.

For instance, the `priority inheritance`, which is important for the kernel's real-time performance and will influence number of context switches. So, instead of counting and analyzing context switches directly, in experiment B the overhead of scheduling inside an ox container is approximately evaluated by a relative way. As for the context switches caused by importng CBS based scheduling in the kernel, people can refer to [3] for more iformation; yet these results do not straightly apply to oxc work.

The hardware and software used in the experiment are shown in table 4.1.

Hardware platform	
Processor	Intel(R) Core(TM) Duo E8500
Frequency	3.16GHz
RAM	
Software platform	
Linux distribution	Ubuntu 11.10
Compiler version	gcc 4.6.1
Kernel version	3.4.0-rc+

Table 4.1: Hardware-Software platform

The chapter is organized like this: the tracing tool and synthetic benchmark tool we use in the experiment are described; then it's the design and result analysis of each experiment; finally, what we learn from the experiment is concluded.

4.1 Ftrace in Linux kernel

Ftrace[8] is an internal tracer designed to help out developers of systems to find out what is going on inside the kernel. The name ftrace comes from "function tracer", which is its original purpose and the reason it is used here. Now there are various kinds of tracers incorporated in Ftrace. You can use it to trace context switces, hong long interrupts are disabled, and so on.

Ftrace uses *debugfs* file system to hold control files as well as file to display output. Typically, ftrace is mounted at `/sys/kernel/debug`.

```
#mount -t debugfs nodev /sys/kernel/debug
```

After this command, a directory `/sys/kernel/debug/tracing` will be created containing interfaces to configure ftrace and display results.

```
#cd /sys/kernel/debug/tracing
```

The following commands will be assumed to be called under `tracing` directory. There are several kinds of tracers available in ftrace, simply cat the `available_tracers` file in the `tracing` dorectory. The output could vary with enabling or disabling kernel configuration options concerned with ftrace functionality in compilation time.

```
#cat available_tracers
blk function_graph mmiotrace wakeup_rt wakeup function
   sched_switch nop
```

The `function` is the function tracer. It uses the `-pg` option of `gcc` to have every function in the kernel call a special function `mcount()` for tracing all kernel functions. The `function_graph` is similar to the function tracer except that the function tracer probes functions on their entry whereas the function graph tracer traces on both entry and exit of a function. It is called function graph tracer because it provides the ability to draw a graph of function calls similar to C code as tracing results. This `function_graph` is what we use in experiments. To enable the function graph tracer, just echo `function_graph` into the `current_tracer` file.

```
#echo function_graph > current_tracer
```

A trace can be started and stopped through configuring `tracing_on` file. Echo 0 into this file to disable the tracer or 1 to enable it. Cat the file will display whether the tracer is enabled or not.

The output of the trace in held in file `trace` in a human readable format. The ftrace will by default trace all functions in the kernel. In most cases, people only care about particular functions. To dynamically configure which function to trace, the `CONFIG_DYNAMIC_FTRACE` kernel option should be set in compilation time to enable dynamic ftrace. Actually, `CONFIG_DYNAMIC_FTRACE` is highly recommanded and defaultly set because of its performance enhancement. To filter which function to trace or not, two files are used: `set_ftrace_filter` for enabling the tracing of a specific function and `set_ftrace_notrace` to disable the tracing of some function. A list of available functions that you can add to these files is listed in `available_filter_functions`. In the later experiment, the oxc function `task_tick_oxc` is traced by setting up like this:

```
#echo task_tick_oxc > set_ftrace_filter
```

4.2 Tbench

The tbench [9] benchmark is a tool that measures disk throughput for simulated netbench runs. Tbench reads a load description file called client.txt that was derived from a network sniffer dump of a real netbench run and it produces only the TCP and process load and no filesystem calls. One exmple to run tbench test:

```
$tbench_srv
$tbench 2 -t 100
```

The `tbench_srv` should be invoked before running `tbench`. The second command starts two tbench connections with one client thread and one server thread in each connection. The two connections will run simultaneously and the runtime of the benchmark will be 100 seconds.

4.3 Experiment A

4.3.1 The experiment design

In this experiment, the execution time of oxc functions are measured. In a Linux system, even if the oxc patch is applied in the kernel, when there is no oxc tasks, the system performs as a plain Linux system. In such a case, the possible oxc overheads include the code execution time in function `is_oxc_task` and the oxc related initialization when a scheduling group is created; both are negligible.

During the experiment, the execution time of following oxc functions are measured using Ftrace:

- `check_preempt_curr_oxc`
- `pick_next_task_oxc`
- `put_prev_task_oxc`
- `task_tick_oxc`

They are most often called oxc functions as they enclose in most frequently invoked scheduling operations. The oxc functions like `enqueue_task_rq_oxc` and `dequeue_task_rq_oxc` only happen when a task enters or leaves an ox container. And are not often happens in the following experiment set up.

There will be six individual tests differing in the number of hyper ox containers in the system. In the first test, there is only one hyper ox container; then, in each test one more hyper ox container would be added. All hyper ox containers in the experiment are identical. Each hyper ox-container has two ox containers with the same CPU reservation parameter $0.1ms/1ms$. Each ox container has one dummy task within it. The dummy task simply runs a forever while loop and it is a rt task with policy `SCHED_FIFO`. This while loop task will exhuaust the reservation in its ox container. The scheduling policy is arbitrarily chosen without special thoughts. On the contrary, the situation for modular scheduling inside an ox container is intentionally arranged as simple as possible to clarify the effect of oxc framework.

The measured execution time of above oxc functions comprises the time consumed by codes involving with the oxc control and operations defined in modular scheduler which are encapsulated inside the these functions.

4.3.2 Experiment results

The results of six tests are listed in table 4.2. The index of a test indicates the number of hyper ox containers in that test. The two fields in the pair are average vaule and standard deviation of the measured function execution time in micro seconds.

	test1	test2	test3	test4	test5	test6
pick_next_task_oxc	(0.168, 0.083)	(0.155, 0.070)	(0.206, 0.178)	(0.230, 0.215)	(0.211, 0.216)	(0.246, 0.251)
put_prev_task_oxc	(0.827, 0.049)	(0.834, 0.096)	(0.820, 0.103)	(0.801, 0.111)	(0.829, 0.146)	(0.852, 0.251)
task_tick_oxc	(0.272, 0.192)	(0.275, 0.182)	(0.263, 0.155)	(0.261,0.15)	(0.245,0.158)	(0.249,0.146)
check_preempt_curr_oxc	-	-	-	-	-	-

Table 4.2: Measured execution time, in micro seconds, of oxc functions in the format of *(mean, standard deviation)*

The first surprise is from the row for `check_preempt_curr_oxc`. That is, no tracing result for `check_preempt_curr_oxc` is recorded. An analysis of overhead in this function is necessary. The details of this function is in list 3.4.4. There are three cases when to check if a task can preempt the currently running task. When only one of the two is an oxc task or both two are oxc tasks and not in the same ox containers, the comparison cost is just several `if-else` instructions. If they are two oxc tasks in the same container, this function follows the procedure in Linux scheduling; in addition, in our experiment setup, there is only one task inside an ox container. In short words, this function is not a significant soure for oxc framework overhead in tests. Although this could be a reason to explain that the ftrace fails to measure the execution time of `check_preempt_cur_oxc` during tests, it reminds us that, given the feature of hierarchical scheduling, it may be attractive to develop a new recording tool so as to evaluate the oxc framework more accurately.

The test results of other three oxc functions are illustrated in figure 4.1, 4.2 and 4.3. The variable parameter in each test is the number of ox containers. And experiment results show that at least in current oxc framework, the codes execution time is influenced by the number of ox containers in the system.

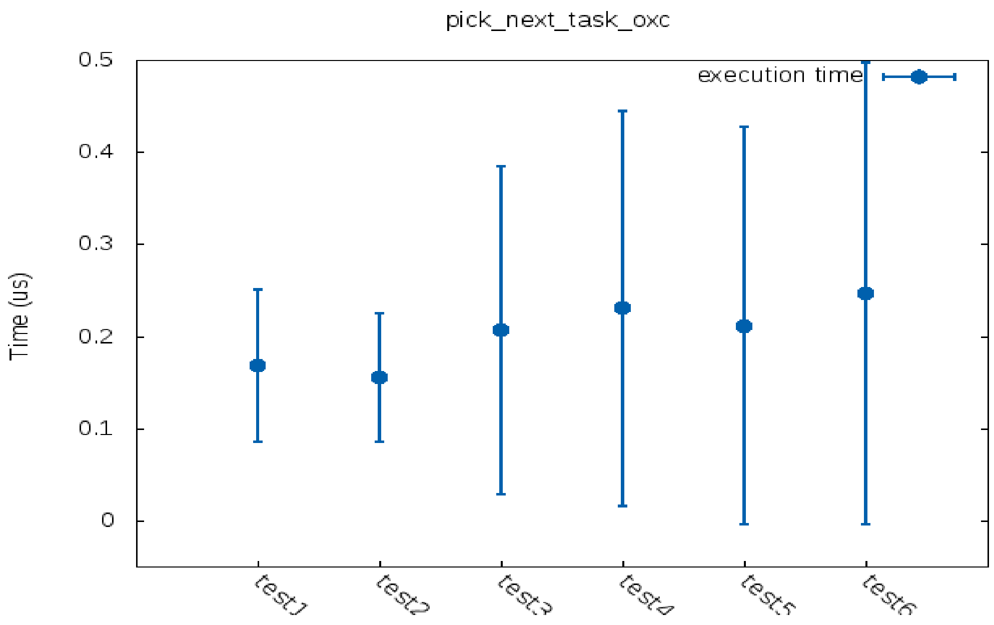


Figure 4.1: Measured execution time for `pick_next_task_oxc`

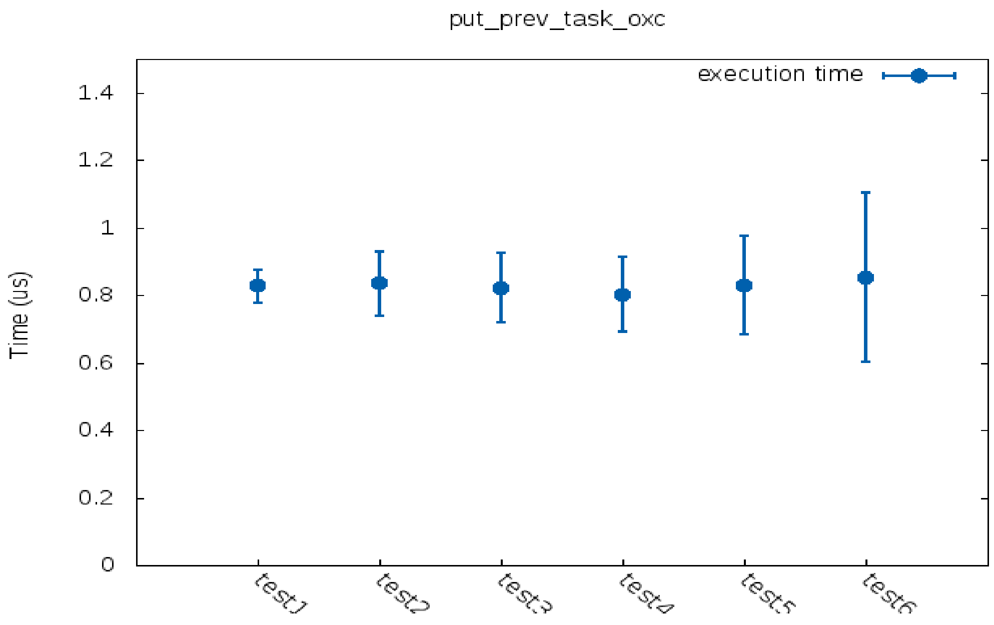


Figure 4.2: Measured execution time for `put_prev_task_oxc`

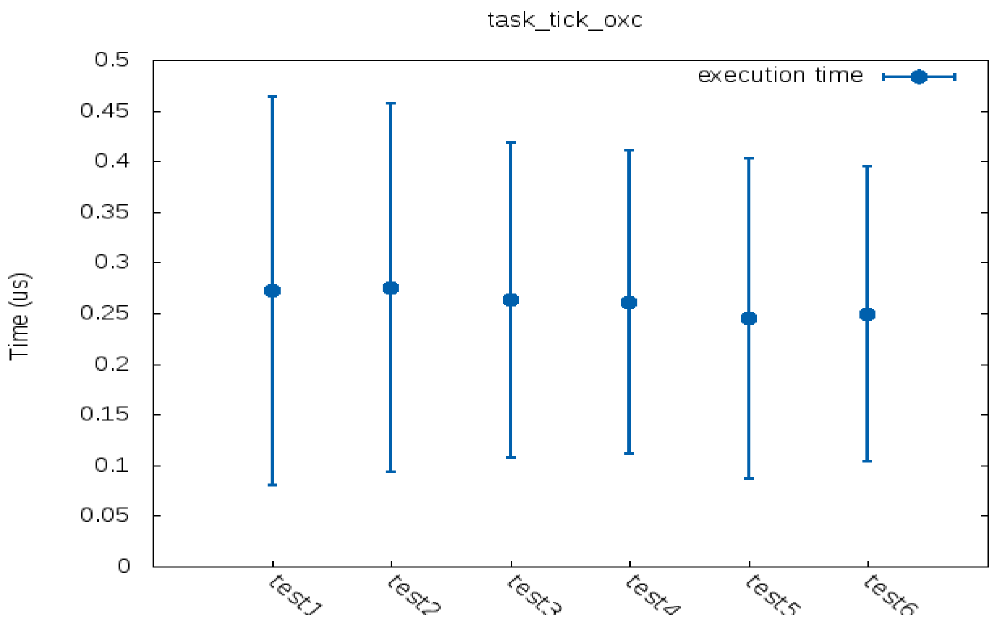


Figure 4.3: Measured execution time for `task_tick_oxc`

Figure 4.1 shows the statistical result of `pick_next_task_oxc` in each test. One observation from the figure is that with more ox containers joining the system, the time spent on executing the function codes fluctuates more. This trend also reflects in the results for `put_prev_task_oxc`, as in figure 4.2. However, figure 4.3 for `task_tick_oxc` does not show this pattern.

Now we are going to probe why the execution time of `task_tick_oxc` is more stable than the result of the other two. A look at the body of the three functions in list 3.4.5, 3.4.6 and 3.4.8, we can find that except for the enclosed scheduling operations, other codes in the three functions are actually the same: to update the per CPU runqueue, to update the per container runqueue and update the current ox container. So, the different variance behaviour in measured execution time for oxc functions may be affected by the performance of sheduling operations inside oxc container.

The three encapsulated scheduling operations `pick_next_task_rt`, `put_prev_task_rt` and `task_tick_rt` are defined in rt scheduling class. Specifically, the codes of `task_tick_rt`, which is called inside oxc function `task_tick_oxc`, is listed below. This is a very simple function. If people read the other two scheduling operations' codes in *linux/sched/rt.c*, this function is still less complex in dealing with runqueues. When scheduling operations are called inside an ox container, there will be extra cost because the raw handle of implementation details, and such an influence may be smaller when the scheduling operation itself is simple. This explains why the result in 4.3 is stable in both mean value and standard variance.

Listing 4.1: `task_tick_rt`

```c
static void task_tick_rt(struct rq *rq, struct task_struct *p,
    int queued)
{
        update_curr_rt(rq);

        watchdog(rq, p);

        /*
         * RR tasks need a special form of timeslice management.
         * FIFO tasks have no timeslices.
         */
        if (p->policy != SCHED_RR)
                return;
        /*
         * The left part has no effects in our tests.
         */
        ...

}
```

4.4 Experiment B

4.4.1 Experiment design

The aim of this experiment is to estimate the overall overhead in oxc framework through comparison with non real-time CPU bandwidth control mechanisms that are already in Linux kernel. During tests the synthetic load is generated by the tbench benchmark tool. The CPU bandwidth allocated to tbench connections are allocated by oxc control, rt throttling and cfs bandwidth control individually. The tbench throughput results of oxc framework are then compared with rt throttling and cfs bandwidth control. By such comparisons, the overhead introduced by oxc control is then evaluated in a relative way.

Two tbench connections will be set up in the system. Each connection will be dedicated to one CPU. Without constraints, they will consume all CPU time. In the experiment, the CPU bandwidth allocated to tbench traffic is restricted. The per CPU bandwidth parameter used in tests includes $0.05ms/1ms$, $0.1ms/1ms$, $0.2ms/1ms$, $0.4ms/1ms$, $0.6ms/1ms$ and $0.8ms/1ms$. Note that these are the per CPU bandwidth that is planned to assign to tbench tasks. Each cpu bandwidth control mechnism will restricts the tbench execution not exceed the configured value. And the throughput results will be proportional to the overhead in each bandwidth control mechanism.

When rt throttling is tested, tbench clients and servers will be scheduled as rt tasks with policy `SCHED_RR`. Otherwise the client and server in the same connection cannot run at all. Correspondingly, to compare with rt throttling, tbench threads inside the ox container will be set as rt tasks with `SCHED_RR` policy too. When comparing the oxc control with cfs bandwidth control, the tbench threads inside ox containers will run as normal tasks.

4.4.2 Experiment results

The thrughput results are shown in table 4.3 and 4.4.

per CPU bandwidth	rt throttling	oxc control + rt scheduling
0.05/1ms	21.9335	18.5313
0.1ms/1ms	43.5794	36.89324
0.2ms/1ms	92.7356	73.5099
0.4ms/1ms	172.582	147.806
0.6ms/1ms	233	225.72
0.8ms/1ms	319.297	297.0607

Table 4.3: Throughputs, in Mbps/sec, under rt throttling and oxc control

per CPU bandwidth	cfs bandwidth control	oxc control + cfs scheduling
0.05ms/1ms	24.8825	19.2151
0.1ms/1ms	52.2106	39.4268
0.2ms/1ms	106.226	77.4225
0.4ms/1ms	215.071	151.465
0.6ms/1ms	323.628	234.369
0.8ms/1ms	433.025	305.1

Table 4.4: Throughputs, in Mbps/sec, under cfs bandwidth and oxc control

After a glance on the two tables, it's apparent to note that the throughput results under cfs bandwidth control outperform the other two. There are two reasons for this. Firstly, the overhead brought by cfs bandwidth control is indeed lower than the other two mechnanisms. Secondly, both oxc tasks and rt tasks are more greedy than cfs tasks when they occupy a CPU. For oxc tasks, both rt and normal tasks are lower priority tasks, between which normal tasks are lower priority tasks. the oxc task or rt tasks will not give up a CPU to a lower priority task until the CPU reservation is totally consumed. So lower priority tasks in the system are stressed more under oxc control or rt throttling, especially when high reservation parameter is configured. However, in cfs scheduling, cfs tasks, with or without CPU reservation, can share the CPU evenly.

Having solved the above question, let's first study the peformance of tbench tasks in oxc framework when they are scheduled with `SCHED_RR` policy. The comparison between oxc control and rt throttling in table 4.3 is visulized in figure 4.4. At first, the throughput result under rt throttling is higher and growing faster than the result in oxc control. However, as the more CPU bandwdith is reserved to tbench tasks, the throughtput results under the two means are converging. In fact, the throughtput growing trend in oxc control are consistent. When it comes to rt throttling, it's not like this. When a relatively small fraction of CPU is assigned to rt throttling and oxc control, rt throttling shows better performance. However, with increasing the reserved CPU bandwdith, the stress of rt throttling on the whole system is rising too, which slows growth of the throughput. Under oxc control, the throughput result is almost linear with the given reserved bandwidth. The

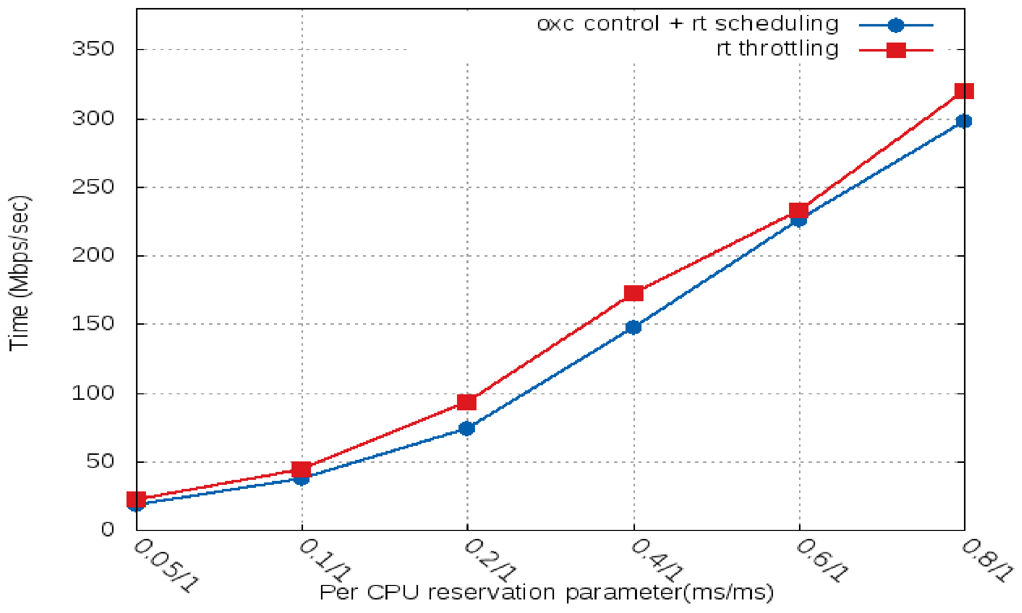


Figure 4.4: *oxc control* vs. *rt throttling*

overhead in oxc control behaves as a constant factor.

The comparason between oxc control and CPU bandwidth control is ahown in figure 4.5. As we just analyzed, cfs bandwidth control has much better throughput result. One observation is that althouth with less rasing speed, the throughput increasing trend in oxc control has the similar shape as in cfs bandwidth control. The result under oxc framework has another meaningful implication. When oxc control is used, allocations of bandwidth in the system should be cautioned so as to achieve an optimal system performance.

At last, figure 4.6 mixes the statistics in table 4.3 and 4.4 and draws the throughput results under oxc control with rt throttling and cfs scheduling in the same graph. The results are quite close, and the difference can be regarded as the scheduling cost between rt and cfs scheduling in an ox container. Still, cfs scheduling shows better results than rt scheduling even under oxc framework. This says that cfs scheduling introduces less overhead in the system than rt scheduler. Such a comparison causes us to think about one possible application of oxc framework. In some cases, when to compare two schedulers, we can set up the environment inside an ox container; or by

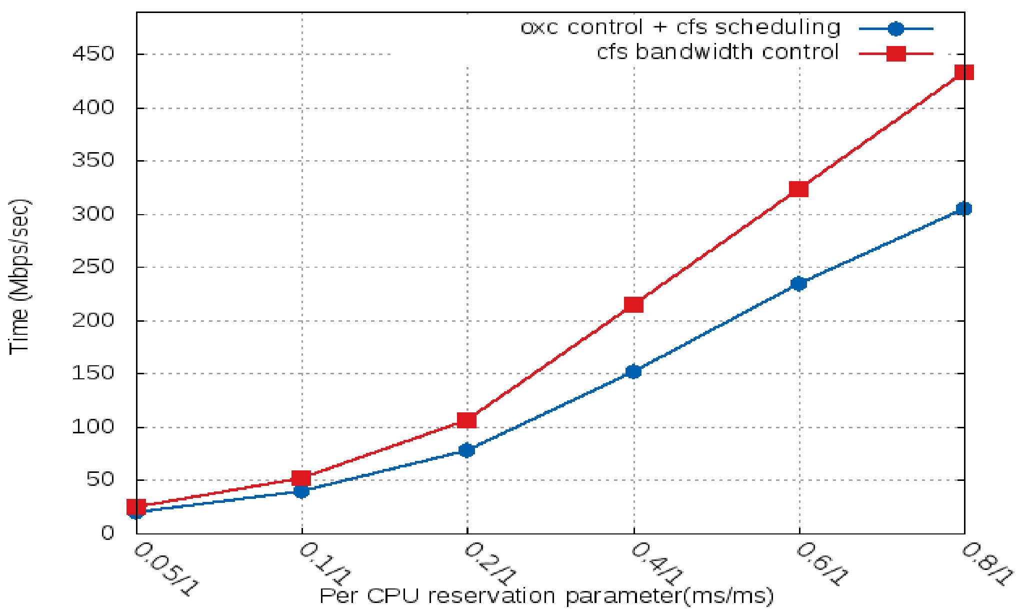


Figure 4.5: *oxc control* vs. *cfs bandwidth control*

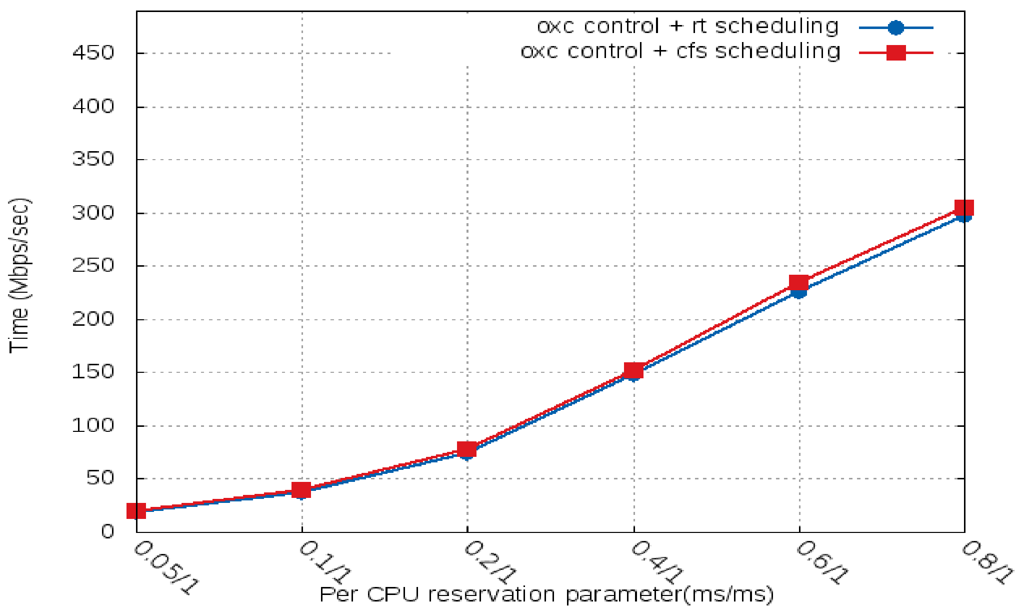


Figure 4.6: *oxc control + rt throttling* vs. *oxc control + cfs scheduling*

preparing a certain number of ox containers, a lightweight networked testbed.

4.5 Experiment feedbacks

The experiment does not show that the performance of oxc control is better than existing bandwidth control methods. This is also not the experiment objective(the comparison result with rt throttling is a small surprise). The experiment outcome has an unstated meaning for future development of oxc framework. Experiment A gives precise measurement of oxc function execution time and confirms the necessary to improve implementation quality of the oxc framework. In experiment B, the overall performance of oxc framework is compared with rt throttling and cfs bandwidth control. Its experiment analysis show us that how to distribute CPU bandwidth will affect both the work inside an ox container and the whole system behaviour; it also raises one example for oxc control usage. These feedbacks would be incorporated in the evolvement of the oxc framework.

Chapter 5

Conclusions and Future Work

Bibliography

[1] L. Abeni and G. C. Buttazzo, “Integrating multimedia applications in hard real-time systems,” in *IEEE Real-Time Systems Symposium*, pp. 4–13, 1998.

[2] “Rtai - real time application interface.” `https://www.rtai.org/`.

[3] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, “Aquosa - adaptive quality of service architecture,” *Softw., Pract. Exper.*, vol. 39, no. 1, pp. 1–31, 2009.

[4] D. Faggioli, M. Trimarchi, F. Checconi, and S. Claudio, “An edf scheduling class for the linux kernel,” in *Proceedings of the 11th Real-Time Workshop (RTLW)*, October 2009.

[5] W. Mauerer, *Professional Linux Kernel Architecture.* Wiley Publishing, Inc., 2008.

[6] R. Love, *Linux Kernel Development (3rd Edition).* Addison-Wesley, 2010.

[7] “Red-black trees (rbtree) in linux.” Documentation/scheduler/rbtree.txt.

[8] “ftrace - function tracer.” Documentation/trace/ftrace.txt.

[9] “tbench(1) - linux man page.” `http://linux.die.net/man/1/tbench`.