

# Hierarchical Multiprocessor CPU Reservations for the Linux Kernel

---

Fabio Checconi, Tommaso Cucinotta,  
Dario Faggioli and Giuseppe Lipari

June 30, 2009

# Goal

---

Support arbitrary CPU reservations in the Linux kernel, while preserving POSIX compliance and the current scheduler structure as much as possible.

## Context: the IRMOS Project

---

IRMOS is an EU project, aiming at supporting real-time requirements in Service Oriented Infrastructures.

The reference systems are complex deployments of multi-component services in a network of distributed resources.

# Scheduling Requirements

---

CPU is Yet Another Resource, (on the host side) we need a scheduler:

- ▶ that can handle multiprocessor virtual machines (KVM is used to deploy VMs hosting services);
- ▶ that supports hard limits (people buy service time);
- ▶ that provides predictable response times (real-time services must respect real-time constraints).

# Requirements Remapping

---

Almost everything is already there...

- ▶ each VM is put in its own cgroup;
- ▶ sched\_rt and throttling expose an interface to support predictable service and hard limits.

Our paper describes how we enhanced throttling basing it on EDF and on a new system model/analysis recently introduced by Bini et al.

# System Model

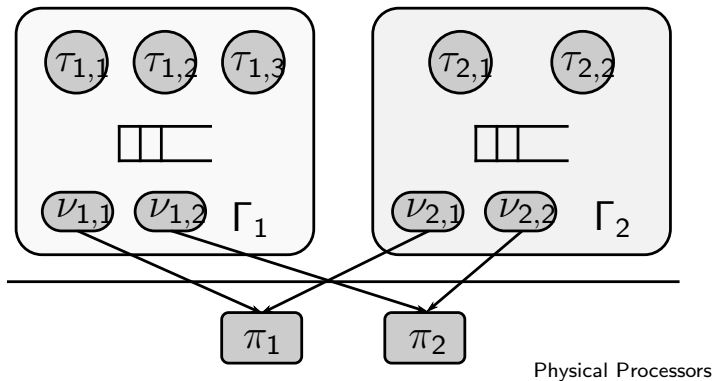
---

Most of what follows is borrowed from “The Multi Supply Function Abstraction for Multiprocessors,” by Bini et al., RTCSA '09.

- ▶ Tasks belonging to the same application are grouped in the same task group;
- ▶ each task group receives service from a set of independent *virtual processors*  $\nu_{i,1,\dots,m}$ ;
- ▶ whenever a virtual processor is selected for execution, a task belonging to its task group is scheduled.

# Block Diagram

---



# Scheduling Algorithm

---

The system model allows for a number of possible configurations; we opted for:

- ▶ Partitioned, hierarchical Hard-CBS to schedule virtual processors on physical CPUs;
- ▶ Global fixed priority scheduling among tasks on the same task group;
- ▶ Static, symmetric bandwidth assignment among virtual processors: If a task group is assigned  $Q_i/P_i$  all of its virtual processors will get  $Q_i/P_i$ .



# H-CBS

---

The Hard-CBS is a non workconserving scheduling algorithm based on EDF.

Each scheduled entity (virtual processors in our case) can be assigned a share of the physical processor time, in the form of  $Q$  time units every  $P$ . If an entity requires more than allocated it is throttled.

To characterize how each single virtual processors receive service from the physical processors they are scheduled on, we use the  $(\alpha, \Delta)$  model, which characterizes the service in terms of *bandwidth*  $\alpha$ , and *delay*  $\Delta$ .

For the H-CBS server we're using, we have:

$$\alpha = \frac{Q}{P} \quad \Delta = 2P - 2Q.$$

## $M(\alpha, \Delta)$

---

Bini et al. introduced a way of composing multiple single CPU reservations into a single multiserver one.

Using their and other known results allows us to derive a schedulability test for our algorithm.

# Interfering Workload

---

For each task  $\tau_k$  we need to consider the interfering *workload* from higher priority tasks:

$$\overline{W}_k^{\text{FP}} = \sum_{i=1}^{k-1} \overline{W}_{k,i},$$

where

$$\overline{W}_{k,i} = N_{k,i} C_i + \min\{C_i, D_k + D_i - C_i - N_{k,i} T_i\},$$

with  $N_{k,i} = \left\lfloor \frac{D_k + D_i - C_i}{T_i} \right\rfloor$ .

# Interference

---

Now we can consider how the interfering workload is distributed among the various virtual processors, and find an upper bound to the interference:

$$\bar{T}_k = L_0 + \sum_{\ell=1}^m \min \left( L_\ell, \frac{\max \left( 0, W_k - \sum_{p=1}^{\ell-1} pL_p \right)}{\ell} \right).$$

$L_\ell$  is the duration, in  $[0, D_k)$ , over which service is provided by  $\ell$  virtual processors in parallel.

# Schedulability

---

A task set  $\Gamma = \{\tau_i\}_{i=1,\dots,n}$  is schedulable by a fixed priority algorithm on a set of virtual processors  $\mathcal{V} = \{\nu_j\}_{j=1,\dots,m}$  modeled by  $\{Z_j\}_{j=1,\dots,m}$ , if

$$\forall k \in \mathbb{N} : 1 \leq k \leq n \quad C_k + \bar{T}_k^{\text{FP}} \leq D_k,$$

with  $\{L_\ell\}_{\ell=0,\dots,m}$  calculated as follows:

$$L_0 = D_k - Z_1(D_k)$$

$$L_\ell = Z_\ell(D_k) - Z_{\ell+1}(D_k)$$

$$L_m = Z_m(D_k).$$

# Implementation

---

- ▶ Use an RB tree to store groups and tasks, ordered by priorities *or* deadlines (boosting can promote a group to a fixed priority);
- ▶ changed the `rt_bandwidth` timer to be per-runqueue;
- ▶ added a *task runqueue* per each task group, to store its child tasks, which cannot be stored together with child runqueues (they have no deadline).

## Tree Sorting

---

```
int rt_entity_before(struct sched_rt_entity *a,
                    struct sched_rt_entity *b)
{
    struct rt_rq *rqa = group_rt_rq(a),
        *rqb = group_rt_rq(b);
    if ((!rqa && !rqb) || (rqa->rt_nr_boosted &&
        rqb->rt_nr_boosted))
        return rt_se_prio(a) < rt_se_prio(b);
    if (rqa->rt_nr_boosted)
        return 1;
    if (rqb->rt_nr_boosted)
        return 0;
    return rqa->rt_deadline - rqb->rt_deadline < 0;
}
```



## Task Runqueues

---

The only user-visible change is the introduction of task runqueues, needed to keep tasks separated from groups (groups have priorities only when boosted).

In addition to specify a  $Q/P$  assignment for each cgroup, the user has to specify an assignment for its task runqueues.

The bandwidth used for task runqueues cannot be used for groups.

## Interface Implications

---

To create a task group, as usual:

```
# mount -t cgroup cgroup /dev/cgroup ; cd /dev/cgroup
# mkdir tg0
```

To assign  $Q = 20\text{ms}$  over  $P = 100\text{ms}$  to its tasks:

```
# echo 100000 > tg0/cpu.rt_period_us
# echo 20000 > tg0/cpu.rt_runtime_us
# echo 100000 > tg0/cpu.rt_task_period_us
# echo 20000 > tg0/cpu.rt_task_runtime_us
```

# Data Structures

---

```
struct rt_edf_tree {
    struct rb_root rb_root;
    struct rb_node rb_leftmost;
};

struct rt_rq {
    struct rt_edf_tree active;
    u64 rt_deadline;
    struct hrtimer rt_period_timer;
    /* ... */
};
```

## Data Structures (2)

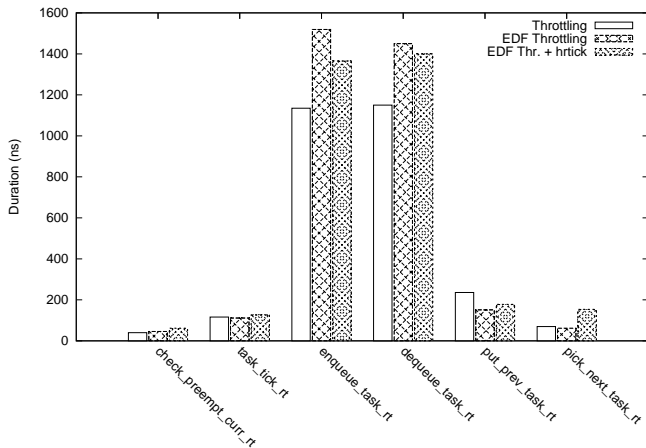
---

```
struct task_rt_group {
    struct sched_rt_entity **rt_se;
    struct rt_rq **rt_rq;

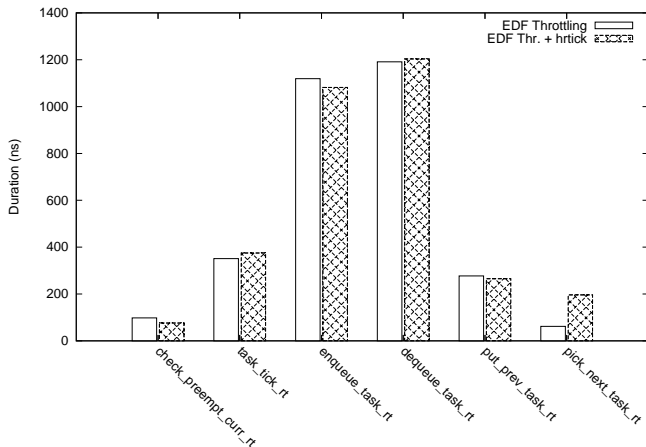
    struct rt_bandwidth rt_bandwidth;
    struct task_group *tg;
};

struct task_group {
    struct task_rt_group rt_rq_group;
    struct task_rt_group rt_task_group;
    /* ... */
};
```

# Overheads



# HRTick



# Future Work

---

From an academic POV:

- ▶ Give a formal treatment to shared resources access;
- ▶ evaluate bandwidth partitioning alternatives.

About the code:

- ▶ Support non-cgroup configs;
- ▶ evaluate overheads more extensively;
- ▶ reintroduce priority arrays for task runqueues (?);
- ▶ one cpupri per task group (?);
- ▶ auto-determined bandwidth for task runqueues (?);
- ▶ and many, many others...