JAVA PLATFORM FOR HIGH-PERFORMANCE AND REAL-TIME LARGE SCALE DATA

Project Number 318763

D3.6 – Final Operating System with Real-Time Support

Version 1.0
21 May 2015
Final

Public Distribution

Scuola Superiore Sant'Anna

Project Partners: **aicas**, **HMI**, **petaFuel**, **SOFTEAM**, **Scuola Superiore Sant'Anna**, **The Open Group**, **University of Stuttgart**, **University of York**, **Brno University of Technology**

Project Partner Contact Information

aicas Fridtjof Siebert Haid-und-Neue Strasse 18 76131 Karlsruhe Germany Tel: +49 721 66396823 E-mail: siebert@aicas.com	**HMI** Markus Schneider Im Breitspiel 11 C 69126 Heidelberg Germany Tel: +49 6221 7260 0 E-mail: schneider@hmi-tec.com
petaFuel Ludwig Adam Muenchnerstrasse 4 85354 Freising Germany Tel: +49 8161 40 60 202 E-mail: ludwig.adam@petafuel.de	**SOFTEAM** Andrey Sadovykh Avenue Victor Hugo 21 75016 Paris France Tel: +33 1 3012 1857 E-mail: andrey.sadovykh@softeam.fr
Scuola Superiore Sant'Anna Mauro Marinoni via Moruzzi 1 56124 Pisa Italy Tel: +39 050 882039 E-mail: m.marinoni@sssup.it	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Stuttgart Bastian Koller Nobelstrasse 19 70569 Stuttgart Germany Tel: +49 711 68565891 E-mail: koller@hlrs.de	**University of York** Neil Audsley Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325571 E-mail: neil.audsley@cs.york.ac.uk
Brno University of Technology Pavel Smrz Bozetechova 2 61266 Brno Czech Republic Tel: +420 54114 1282 E-mail: smrz@fit.vutbr.cz	

Contents

Document Control

Version	Status	Date
0.1	Document outline	27 March 2015
0.2	Complete First Draft	8 May 2015
1.0	QA for EC delivery	15 May 2015

Executive Summary

This document constitutes deliverable 3.6 - *Final Operating System with Real-Time Support* of work package 3 of the JUNIPER project. It consists of the final release of the enhanced Linux kernel with real-time bandwidth reservation scheduling support.

The purpose of this deliverable is to describe the content of the final implementation of the Operating System used in the JUNIPER project. It presents the main differences of the real-time improvements respect to deliverable 3.2 with a particular focus on the new included patch and its interface. Subsequently it presents some experiments useful to shows the functionality in terms of reservation for groups of tasks. A final appendix describes the procedure needed to create a working copy of the enhanced Operating System.

Deliverable 3.6 which is described in this document is the final version of the Operating System with Real-Time Support and extends the previous prototype, which has been described in Deliverable 3.2 - *Prototype Operating System with Real-Time Support*. Deliverable 3.1 - *Operating System Real-Time Support Definition* is a prerequisite for reading the present document.

1 Real-Time Enhancements Overview

This section provides an overview of the real-time enhancements contained in the final version of the operating system. The main real-time patches are listed below.

SCHED_DEADLINE[2] provides CPU reservation capabilities at the task level by implementing the Constant Bandwidth Server (CBS) plus the Earliest Deadline scheduling algorithms. It also provides an admission control strategy to guarantee that the timing constraints are not jeopardized.

Bounded-Delay Multipartition (BDM)[3] extends the SCHED_DEADLINE patch by providing CPU reservation for groups of real-time (SCHED_FIFO and SCHED_RR) threads. A more detailed description of this patch is provided in Section 2.

RT_PREEMPT [1] reduces the Linux kernel latencies by replacing most kernel spin-locks with mutexes that support Priority Inheritance (PI) and by moving interrupts and software interrupts to kernel threads. Moreover, it converts the old Linux timer API into separate infrastructures for high resolution kernel timers.

The Budget Fair Queueing (BFQ) [4] provides a fair allocation of disk I/O bandwidth; the amount of bandwidth actually reserved depends on the number of requests and their relative priorities. BFQ is highly tunable for both bandwidth and latency via the `sysfs` infrastructure and it is also compatible with the `cgroup` interface.

A kernel module provides the Java Virtual Machine with an API enabling the application processes to control FPGA accelerators, in terms of loading them, transmitting data and read back them.

2 Bounded-Delay Multipartition

In this section, we describe the implementation of the Bounded-Delay Multipartition model ([3]) in the Linux kernel, and we report its first evaluation.

2.1 Implementation

To better describe the BDM implementation, it is useful to recall the notion of *virtual platform* [3]: a virtual platform Y on a m processors system is modeled by m functions $Y_k \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ ($k = 1, \dots, m$) where, for each $t \in \mathbb{R}_{\geq 0}$, $Y_k(t)$ represents the "minimum amount of CPU time with parallelism at most k" guaranteed to the application in any (time) interval of length t.

The form of the functions Y_k depends on the particular algorithm that the operating system uses to implement the reservation algorithm. We call this algorithm the *root scheduling algorithm*, in order to distinguish it from the *local scheduling algorithm* used within the application to decide which threads (among those eligible for execution) are to be executed at a given time-instant.

Following [3], given arbitrary $\alpha \in \mathbb{Q} \cap [0, 1)$ and $\Delta \in \mathbb{Q}_{>0}$, our implementation provides the capability to create a virtual platform $Y := (Y_k)_{k=1}^{m}$ such that

$$Y_k(t) \geq k\,\alpha \cdot \max\{0,\, t - \Delta\}\,, \quad \text{for all } k = 1, \dots, m \text{ and } t \geq 0.$$

To achieve this, m new scheduling entities $\pi_1, \dots, \pi_m$, named *virtual processors*, are associated with each virtual platform. Each virtual processor represents a Hard Constant Bandwidth Server (H-CBS); the (Q, P)-parameters of these servers are given by the transformation

$$Q = \frac{\Delta}{2\,(1-\alpha)} \cdot \alpha,$$
$$P = \frac{\Delta}{2\,(1-\alpha)}.$$

Virtual processors are *statically* allocated to a processor and scheduled in Earliest Deadline First (EDF) order. On the other hand, the threads of the application executing within these virtual processors *can* "migrate" to different processors in conformity with the local scheduling algorithm. This work considers the case of a *Fixed-Priority* (FP) scheduling algorithm at the local level.

2.1.1 Root Scheduler

The main data structures involved in the root scheduler implementation are displayed in Listing 1.

A virtual platform is represented as a `task_group` object; this includes an array of pointers to virtual processors entities (`sched_dl_entity`) and an array of pointers to "real-time" run-queues (`rt_rq`): as already described, there is one virtual processor entity for each physical processor/CPU; moreover, conforming to Linux's current implementation of the FP scheduling policy (`rt_sched_class`), each virtual platform mantains a per-CPU (local) run-queue that is used to implement a distributed global scheduling algorithm (see 2.1.2 below). The "reservation parameters" of a virtual platform are encoded in a dl_bandwidth object (and “cached” in the corresponding sched_dl_entity's): following the previous notation, $Q =$ dl_runtime (ns) and $P =$ dl_period (ns).

```
struct dl_bandwidth {
        raw_spin_lock_t dl_runtime_lock;
        u64 dl_runtime;
        u64 dl_period;
};

/* struct for virtual platforms */
struct task_group {
      struct sched_dl_entity **dl_se;
      struct rt_rq **rt_rq;

      struct dl_bandwidth dl_bandwidth;

      ...
};

/* struct for virtual processors */
struct sched_dl_entity {
      struct rb_node rb_node;

      u64 dl_runtime;
      u64 dl_period;

      s64 runtime;
      u64 deadline;

      int dl_throttled, dl_new;
      struct hrtimer dl_timer;

      struct dl_rq *dl_rq;
      struct rt_rq *my_q;

      ...
};

#define dl_entity_is_task(dl_se) \
      (!(dl_se)->my_q)
```

Listing 1: Main data structures.

We remark that the structure sched_dl_entity is already present in mainline Linux, where it is used to store scheduling entities of SCHED_DEADLINE threads (H-CBSs): our implementation preserves the semantics of its members and augments them with a pointer of type dl_rq (the run-queue on which the virtual processor or the SCHED_DEADLINE thread is to be queued) and with a pointer of type rt_rq (the local run-queue “owned” by this virtual processor or NULL for a SCHED_DEADLINE thread). In particular, the members runtime and deadline represent the current budget and the absolute deadline of the H-CBS server, respectively; also, a timer (dl_timer) is started when the server exhausts its budget (we say that the server is being *throttled*) and set to fire at the next replenishment instant.

The adoption of the same C structure (`sched_dl_entity`) to represent both virtual processors and SCHED_DEADLINE threads allowed us to reuse code already available in Linux's current implementation of the SCHED_DEADLINE scheduling policy (`dl_sched_class`); for example, the functions `dl_runtime_exceeded`, `start_dl_timer`, `dl_timer`, `enqueue_dl_entity`, `dequeue_dl_entity` apply to virtual processor entities with minor modifications.

The `sched_dl_entity`'s of both virtual processors and SCHED_DEADLINE threads that are *Active* (i.e., non-throttled) are enqueued in the same per-CPU red-black trees (from which the name `rb_node`) in order of non-decreasing absolute deadline; the macro `dl_entity_is_task` (line 39) has been introduced to distinguish entities representing SCHED_DEADLINE threads from entities representing virtual processors.

The function `pick_next_task_dl` of the class `dl_sched_class` has been modified as displayed in Listing 2: given a (per-CPU) run-queue `rq`, we first identify the corresponding red-black tree (line 4); if there is no SCHED_DEADLINE or virtual processor entity in the tree, we return `NULL` (lines 9-10); if the tree is not empty, we select the leftmost `sched_dl_entity`, `dl_se`, in the tree (line 12). Then, if the selected entry represents a virtual processor, we return the highest priority real-time job in the corresponding local run-queue (lines 14-21); otherwise, the entry must represent a SCHED_DEADLINE that is returned (lines 24-26). Notice, in particular, that the function `pick_next_task_dl` can now return a thread with SCHED_FIFO or SCHED_RR policy.

```
   /*
 2  * Extract from:  struct task_struct *pick_next_task_dl(struct rq *rq)
    */
 4 struct dl_rq *dl_rq = &rq->dl;
   struct sched_dl_entity *dl_se;
 6 struct task_struct *p;

 8 /* dl_nr_total = # of SCHED_DEADLINE threads + # of virtual processors */
   if (unlikely(!dl_rq->dl_nr_total))
10       return NULL;

12 dl_se = pick_next_dl_entity(rq, dl_rq);
   if (!dl_entity_is_task(dl_se)) {
14       struct rt_rq *rt_rq = dl_se->my_q;
         struct sched_rt_entity *rt_se;
16
         rt_se = pick_next_rt_entity(rq, rt_rq);
18       /* rt_se != NULL */
         p = rt_task_of(rt_se);
20       ...
         return p;
22 }

24 p = dl_task_of(dl_se);
   ...
26 return p;
```

Listing 2: Scheduling a virtual processor.

2.1.2 Local Scheduler

The implementation of the local FP scheduling algorithm is based on Linux's `rt_sched_class`: the basic C structures, `sched_rt_entity` and `rt_rq`, are mantained to implement a `SCHED_FIFO` or `SCHED_RR` scheduling policy. The major effort consisted in the modification of the Linux's *pull-push mechanism*; this mechanism is used to implement a global or, more generally, an Arbitrary Processor Affinity (APA) scheduling algorithm. For simplicity we limit the discussion to the case of global scheduling, but similar considerations apply to the more general case of APA scheduling.

For a virtual platform Y, let $S_Y(t)$ denote the set of real-time threads assigned to Y which are *scheduled* on any of the m (physical) processors at time t, and let $m_Y(t)$ be the set of virtual processors of Y which have been selected by the root scheduler at time t. Then, in the case of global scheduling, our implementation guarantees the so called *Global FP-Invariant* (GFPI) property:

> *If j_r is a runnable thread in Y at time t and if $j_r \notin S_Y(t)$, then*
>
> *(i)* $|S_Y(t)| = |m_Y(t)|$,
>
> *(ii)* $p(j) \geq p(j_r)$ *for each* $j \in S_Y(t)$, *where* $p(j)$ *denote the priority of thread* j.

In order to preserve this invariant, our implementation introduces the functions `group_pull_rt_task` and `group_push_rt_tasks`. The first is called in `pick_next_dl_entity` (line 12 in Listing 2); this function tries to pull a thread on the local run-queue of a virtual processor by scanning all the run-queues in the corresponding virtual platform. The second is called in `post_schedule` on each processor when a scheduling decision is completed; if the "previous" or the "current" thread is a SCHED_FIFO or a SCHED_RR thread, this functions tries to push threads from the corresponding run-queue by searching for a "better" run-queue in the platform. As in mainline Linux, a successful push triggers a *rescheduling* event on the "remote" processor.

2.1.3 User Interface

Similarly to Linux's current real-time throttling infrastructure, the implementation of the BDM model provides an interface based on the CGROUP virtual filesystem.[1]

A virtual platform can be created by making a sub-directory under the `cpu` sub-system directory in this filesystem. Within each such directory, the files "cpu.rt_runtime_us" and "cpu.rt_period_us" are used to configure the reservation parameters (Q and P respectively) of the virtual platform; threads can be assigned to the virtual platform by appending their IDs to the file "tasks".

2.2 Evaluation

In this subsection, we describe two simple experiments illustrating the performance of the proposed solution. First, a runtime test is used to test the correctness of its implementation; then, its overheads are measured and compared to mainline Linux. We executed the experiments on an Intel®Core2™Q6600 quad-core machine with 4GB of RAM, running at 2.4GHz.

[1]For more information on Linux's CGROUP, we refer to the relative documentation in the kernel source tree.

2.2.1 Runtime Validation

For this experiment, we considered the virtual platforms Y_1 and Y_2 defined in Table 1, and the two real-time applications defined in Table 2; moreover, we considered the "disturbing" background workload defined in Table 3.

Virtual platform	# of virt. processors	α	Δ (ms)
Y_1	2	0.72	20
Y_2	2	0.22	20

TABLE 1: *Platforms for validation.*

Virtual platform	i	p_i	C_i (ms)	D_i (ms)	T_i (ms)
Y_1	1	13	10	60	60
	2	12	140	270	270
	3	11	90	520	520
Y_2	4	15	40	270	270
	5	14	40	520	520

TABLE 2: *Applications for validation.*

i	p_i	C_i (ms)	D_i (ms)	T_i (ms)
6	18	25	100	100
7	17	50	200	200
8	16	100	400	400

TABLE 3: *Background workload.*

`rt-app`[2] has been used to generate the workload and to count the number of deadline misses of its threads over a time-window of 120 seconds. We ran this experiment 20 times against both our implementation and mainline Linux. In agreement with the schedulability tests described in D3.1, no misses is detected when using virtual platforms (since the applications are both schedulable). Table 4 reports the corresponding results when using Linux's throttling mechanism.

[2]https://github.com/scheduler-tools/rt-app

i	Throttling (no background)	Throttling (with background)
1	6 ± 1	210 ± 9
2	0 ± 0	222 ± 6
3	0 ± 0	5 ± 1
4	0 ± 0	0 ± 0
5	0 ± 0	0 ± 0

TABLE 4: *Average number of deadline misses for the applications of Table 2 over 20 runs, when using Linux's throttling.*

According to these results, the Linux's throttling mechanism is not able to guarantee the real-time constraints of the applications; this holds even when no background workload is present.

2.2.2 Overhead Measurement

While running the same experiment described in 2.2.1, we measured the execution time of the main scheduling functions, using `ftrace`;[3] the measured kernel functions are:

(a) `pick_next_task_dl`,
(b) `post_schedule`,
(c) `enqueue_task_rt`,
(d) `pick_next_task_rt`,
(e) `task_tick_rt`.

Table 5 and Table 6 report the resulting statistics for the case of virtual platforms and of Linux's throttling, respectively.

Function	Hits ($\times 10^3$)	Duration (μs)
(a)	155	1.1 ± 148.4
(b)	155	0.8 ± 65.3
(c)	147	0.25 ± 9.1
(d)	18	1.6 ± 83.7
(e)	29	0.4 ± 3.5

TABLE 5: *Overhead measurements of kernel functions for the applications of Table 2, when using virtual platforms.*

[3] See `Documentation/trace/ftrace.txt` in the kernel source tree.

Function	Hits ($\times 10^3$)	Average (μs)
(a)	2251	0.1 ± 17.2
(b)	2251	0.7 ± 51.4
(c)	8	0.7 ± 1.8
(d)	2251	0.2 ± 22.1
(e)	29	0.4 ± 2.2

TABLE 6: *Overhead measurements of kernel functions for the applications of Table 2, when using Linux's throttling.*

According to these results, the overheads of virtual platforms are comparable with that of Linux's throttling mechanism; notice also that virtual platforms result in a lower number of scheduling events w.r.t. Linux's throttling.

A Installation

This Appendix presents the procedure required to obtain a working copy of the Operating System including the Linux kernel version 3.14 enhanced with the patches previously described (i.e. BDM, BFQ and PREEMPT_RT). In order to simplify the procedure, the real-time enhanced kernel has been made available as a package for the current stable version of the **Debian Linux** distribution (i.e., version 7 with codename *Wheezy*).

The Appendix updates the one included in Deliverable 3.2 - *Prototype Operating System with Real-Time Support*. In particular, Appendix A.1 describes the actions needed to create a Xen unprivileged domain running a stable version of the Debian distribution that is compatible with the real-time enhanced kernel. Instead, Appendix A.2 shows how to install the kernel and the test presented in Section 2.2.

A.1 Installation of a Debian Wheezy in a Xen unprivileged domain

The unprivileged domain (domU) is created using the *xen-tool* application while the installation of the Debian distribution is done with the *debootstrap* tool.

Note: The procedure is described considering Debian Wheezy also as a host Operating System (dom0) and the standard installation of Xen version 4.3. The use of other host OS could require small modifications of Xen configuration files or packages upgrade.

The command shall be executed in a shell with root privileges and creates a new domU with name *name*, IP address *addr*, number of virtual CPUs *n* which uses its own kernel instead of the one installed in the host machine.

```
xen-create-image --pygrub --dist=wheezy --hostname <name> --vcpus=<n>
                              --ip=<addr>
```

The default behaviour of the xen server is to start the domU after its creation. If this does not happens, the dom must be started executing the command:

```
xm create -f /etc/xen/<name>.cfg
```

The shell of the running unprivileged domain can be accessed both using the secure shell (SSH) at the network address *ip* or using the Xen tool:

```
xm console <name>
```

A.2 Installation of the enhanced Linux kernel

These subsections describe the steps needed to install the kernel enhanced with the real-time patches described in Section 1 on a Debian Wheezy, which shall be executed in a shell with root privileges.

The following commands retrieve the *juniper-private* repository using the git tool and execute the script to perform the installation of the kernel:

```
# git clone https://github.com/JGYork/juniper-private.git
# cd juniper-private/M30/d3_6
# ./d3_6-install.sh
```

If the git tool is missing, it can be installed using the command:

```
apt-get update && apt-get install git
```

The installation script performs the following actions:

- add the wheezy-backports packages repository to the list of locally available repositories;
- update the kernel to wheeze-backports version, and fulfill any required package dependencies;
- install the real-time kernel in the system;
- install the testing tools described in Section 2.2.

References

[1] RT_PREEMPT group. Rt_preempt patch set. `http://www.kernel.org/pub/linux/kernel/projects/rt/`.

[2] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global edf in linux. In *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2011.

[3] Giuseppe Lipari and Enrico Bini. A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation. In *Proceedings of the* 31st *IEEE Real-Time Systems Symposium*, pages 249–258, San Diego, CA, USA, December 2010.

[4] Paolo Valente and Fabio Checconi. High throughput disk scheduling with fair bandwidth distribution. *IEEE Trans. Computers*, 59(9):1172–1186, 2010.