Project Number 318763

D3.2 – Prototype Operating System with Real-Time Support

Version 1.0
4 August 2014
Final

Public Distribution

Scuola Superiore Sant'Anna, University of York

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

Every effort has been made to ensure that all statements and information contained herein are accurate, however the JUNIPER Project Partners accept no liability for any error or omission in the same.

© 2014 Copyright in this document remains vested in the JUNIPER Project Partners.

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

Contents

Document Control

Version	Status	Date
0.1	Document outline	16 June 2014
0.2	Complete First Draft	28 July 2014
1.0	QA for EC delivery	4 August 2014

Executive Summary

This document constitutes deliverable 3.2 - *Prototype Operating System with Real-Time Support* of work package 3 of the JUNIPER project. It consists of a prototype implementation release of the Linux with real-time bandwidth reservation scheduling support.

The purpose of this deliverable is to describe the content of the first 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 their interfaces. Subsequently it presents some tools useful to test the real-time system functionality in terms of reservation for computational and storage resources. A final appendix describes the procedure needed to create a working copy of the Operating System prototype.

Deliverable 3.2 which is described in this document is the first step to the final version of the Operating System with Real-Time Support, which will be submitted later in deliverable 3.6 - *Final Operating System with Real-Time Support* in the third year of the project. Deliverable 3.1 - *Operating System Real-Time Support Definition* is a prerequisite for reading the present document.

1 Introduction

In order to enforce the timing constraint that characterize JUNIPER applications the real-time support must be considered in all components of the framework, starting from the Operating System at the lower layer. The JUNIPER project wants to use international established standards (POSIX) and needs to improve the support for time-sensitive applications in order to manage open real-time systems where tasks can dynamically enter or leave the system at any time.

For these reasons the Linux kernel has been selected and a set of extensions will provide it with real-time capabilities. These extensions are provided as patches for the mainline kernel. This approach as been selected in order to perform dissemination activities in the Linux community addressed toward the integration of the JUNIPER extensions inside the mainline kernel, as has successfully been done with the original SCHED_DEADLINE patch. The list of real-time extensions that are under development and integration in the context of the JUNIPER project are extensively described in deliverable 3.1 - *Operating System Real-Time Support Definition*.

Virtualisation is a popular technology widely used for implementing clusters of servers for cloud computing. Therefore, it is essential for the success of our technology to be able to fruitfully work in a virtualised environment. In line with that target, a part of the Appendix is dedicated to describe how to execute the prototype presented in this document in an unprivileged domain (domU) in a Xen server.

In this deliverable is described the first prototype of the Operative System support for the JUNIPER project. The contributions of the document are divided as follows:

- Section 2 describes the real-time extensions included in the first prototype of the Operating System;
- Later, Section 3 presents the application programming interfaces (API) that are needed to manage the included real-time extensions;
- Finally, Section 4 shows the usage of some tools that can show the effectiveness of the real-time support introduced.
- The Appendix provides a detailed set of instructions useful to build a working installation of the presented prototype. It also describe how to prepare an unprivileged domain to execute the prototype in a virtualised environment.

2 Real-Time Patches

2.1 PREEMPT_RT

The standard Linux kernel only meets soft real-time requirements: it provides basic POSIX operations for userspace time handling but has no guarantees for hard timing deadlines. With Ingo Molnar's Realtime Preemption patch (referenced to as PREEMPT_RT in this document) and Thomas Gleixner's generic clock event layer with high resolution support, the kernel gains hard realtime capabilities.

The PREEMPT_RT patch has raised quite some interest throughout the industry. Its clean design and consequent aim towards mainline integration makes it an interesting option for hard and firm realtime applications, reaching from professional audio to industrial control. All the real-time patches described in this document are based (i.e., developed on it).

The RT-Preempt patch converts Linux into a fully preemptible kernel. This is achieved by:

1. Making in-kernel locking-primitives (using spinlocks) preemptible though reimplementation with rtmutexes.
2. Critical sections protected by spinlocks and rwlocks are now preemptible. The creation of non-preemptible sections (in kernel) is still possible with raw_spinlock.
3. Implementing priority inheritance for in-kernel spinlocks and semaphores.
4. Converting soft interrupt handlers into preemptible kernel threads: The RT-Preempt patch treats soft interrupt handlers in kernel thread context, which is represented by a task_struct like a common user space process. However it is also possible to register an IRQ in kernel context.
5. Converting the old Linux timer API into separate infrastructures for high resolution kernel timers plus one for timeouts, leading to user space POSIX timers with high resolution.

2.2 SCHED_DEADLINE

The SCHED_DEADLINE policy is basically an implementation of the Earliest Deadline First (EDF) scheduling algorithm, augmented with a mechanism called Constant Bandwidth Server (CBS) that makes it possible to isolate the behavior of tasks between each other.

SCHED_DEADLINE uses three parameters, named "runtime", "period", and "deadline" to schedule tasks. A SCHED_DEADLINE task is guaranteed to receive "runtime" microseconds of execution time every "period" microseconds, and these "runtime" microseconds are available within "deadline" microseconds from the beginning of the period. In order to implement this behaviour, every time the task wakes up, the scheduler computes a "scheduling deadline" according to the CBS[2, 1] algorithm. Tasks are then scheduled using EDF[4] on these scheduling deadlines (the task with the smallest scheduling deadline is selected for execution). Notice that this guaranteed is respected if a proper "admission control" strategy (see Section 3) is used.

Summing up, the CBS algorithm assigns scheduling deadlines to tasks so that each task runs for at most its runtime every period, avoiding any interference between different tasks (bandwidth isolation), while the EDF algorithm selects the task with the smallest scheduling deadline as the one to be executed first. Thanks to this feature, also tasks that do not strictly comply with the "traditional" real-time task model can effectively use the new policy.

2.3 BFQ I/O Scheduler

Budget Fair Queueing (BFQ) is a Proportional Share, or equivalently Fair Queueing, disk scheduler that allows each thread to be assigned a fraction of the disk bandwidth.

In BFQ[6], as in the Completely Fair Queueing (CFQ) available in mainline Linux, synchronous requests are collected in per-task queues, and asynchronous requests are collected in per-device or, in the case of hierarchical scheduling, per-group queues. On the other hand, differently from CFQ, when the underlying device driver asks for the next request to serve and there is no queue being served, BFQ uses B-WF2Q+[5], a modified version of WF2Q+[3], to choose a queue, and then selects the first request from that queue in C-LOOK order.

When a new queue is selected it is assigned a budget decremented each time a request from the same queue is served. The budget is allocated at the level of disk sector, not considering the overhead introduced by the filesystem. When the device driver asks for new requests and there is a queue under service, they are chosen from that queue until one of the following conditions is met:

- the queue exhausts its budget, or
- the queue is spending too much time to consume its budget, or
- the queue has no more requests to serve.

As in CFQ, if a synchronous queue has no more requests to serve, but it has some budget left, the scheduler idles (i.e., it tells to the device driver that it has no requests to serve even if there are other active queues) for a short period, waiting for a new request from the task owning the queue.

BFQ supports hierarchical scheduling: queues are collected in a tree of groups, and there is a distinct B-WFQ2+ scheduler on each non-leaf node; leaf nodes are "ordinary" (i.e., non-hierarchical) request queues.

3 User Interface

3.1 SCHED_DEADLINE

In order for deadline scheduling to be effective and useful, it is important to have some method to keep the allocation of the available CPU bandwidth to the tasks under control. This is usually called "admission control" and if it is not performed at all, no guarantee can be given on the actual scheduling of the -deadline tasks.

Since when RT-throttling has been introduced in Linux, each task group has a bandwidth associated, calculated as a certain amount of runtime over a period. Moreover, to make it possible to manipulate such bandwidth, readable/writable controls have been added to both procfs (for system wide settings) and cgroupfs (for per-group settings). Therefore, the same interface is being used for controlling the bandwidth distribution to SCHED_DEADLINE tasks. However, more discussion is needed in order to figure out how to manage SCHED_DEADLINE bandwidth at the task group level.

A major difference between deadline bandwidth management and RT-throttling is that SCHED_DEADLINE tasks have bandwidth on their own (while RT ones don't!), and thus we don't need an higher level throttling mechanism to enforce the desired bandwidth.

The system wide settings are configured under the /proc virtual file system.

For now the RT knobs are used for SCHED_DEADLINE admission control and the deadline runtime is accounted against the RT runtime. We realise that this isn't entirely desirable; however, it is better to have a small interface for now, and be able to change it easily later. The ideal situation is to run RT tasks from a deadline server; in which case the RT bandwidth is a direct subset of deadline one.

This means that, for a (root) domain comprising M CPUs, SCHED_DEADLINE tasks can be created while the sum of their bandwidths stays below:

$$M \cdot \frac{\text{sched\_rt\_runtime\_us}}{\text{sched\_rt\_period\_us}}$$

It is possible to disable the bandwidth management logic, and be thus free of oversubscribing the system up to any arbitrary level. This is done by using the command

```
# echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

The default value for SCHED_DEADLINE bandwidth is to have *rt_runtime* equal to 950000. With *rt_period* equal to 1000000, by default, it means that SCHED_DEADLINE tasks can use at most 95% of the available total bandwidth in each domain.

Concerning the task-level interface, the definition of a SCHED_DEADLINE task that executes for a given amount of runtime at each instance, is achieved by declaring:

- a (maximum or typical) instance execution time,
- a minimum interval between consecutive instances,
- a time constraint by which each instance must be completed.

Therefore:

- a new *struct sched_attr*, containing all the necessary fields is defined inside the kernel;
- the new scheduling related syscalls that manipulate it, i.e., *sched_setattr()* and *sched_getattr()* are implemented.

The following sample code illustrates the use of these new structure and syscalls

```
#include <sched.h>

...
struct sched_attr attr;

attr.size = sizeof(struct attr);
attr.sched_policy = SCHED_DEADLINE;
attr.sched_runtime = 300000000;
attr.sched_period = 1000000000;
attr.sched_deadline = attr.sched_period;
...

if (sched_setattr(gettid(), &attr, 0))
        perror("sched_setattr()");

...
```

We remark a SCHED_DEADLINE task cannot fork. Also, deadline tasks can not have an affinity mask smaller that the entire root domain they are created on. However, affinities can be specified through the cpuset facility; see *Documentation/cgroups/cpusets.txt*. An example of a simple configuration (to pin a deadline task to CPU 0) follows, where *rt-app* (see Section 4) is used to create a SCHED_DEADLINE task.

```
# mkdir /dev/cpuset
# mount -t cgroup -o cpuset cpuset /dev/cpuset
# cd /dev/cpuset
# mkdir cpu0
# echo 0 > cpu0/cpuset.cpus
# echo 0 > cpu0/cpuset.mems
# echo 1 > cpuset.cpu_exclusive
# echo 0 > cpuset.sched_load_balance
# echo 1 > cpu0/cpuset.cpu_exclusive
# echo 1 > cpu0/cpuset.mem_exclusive
# echo $$ > cpu0/tasks
# rt-app -t 100000:10000:d:0 -D5
```

3.2 BFQ

Currently, I/O schedulers in Linux are assigned globally at boot time only. To choose I/O schedulers at boot time, use the argument *elevator=bfq*; *cfq*, *deadline*, *noop* are also available.

Each I/O queue has a set of I/O scheduler tunables associated with it. These tunables control how the I/O scheduler works. You can find these entries in */sys/block/<device>/queue/iosched/* assuming that you have sysfs mounted on /sys.

Starting with Linux 2.6.10, it is now possible to change the I/O scheduler for a given block device at runtime (thus making it possible, for instance, to set the CFQ scheduler for the system default, but set a specific device to use the BFQ or deadline schedulers - which can improve that device's throughput).

To set a specific scheduler, simply do this:

```
# echo SCHEDNAME > /sys/block/DEV/queue/scheduler
```

where SCHEDNAME is the name of a defined I/O scheduler, and DEV is the device name (e.g., hda, hdb, sga). The list of defined schedulers can be found by looking at */sys/block/DEV/queue/scheduler*: the list of valid names will be displayed, with the currently selected scheduler in brackets:

```
# cat /sys/block/hda/queue/scheduler
noop deadline cfq [bfq]
# echo deadline > /sys/block/sda/queue/scheduler
# cat /sys/block/hda/queue/scheduler
noop [deadline] cfq bfq
```

Some of the parameters available to the user for configuring BFQ, exported through the canonical I/O scheduler sysfs interface are:

- *timeout_sync, timeout_async*: the maximum amount of unpreempted service time that can be given to a task, respectively for synchronous and asynchronous queues. It allows the user to specify a maximum slice length to put an upper bound to the latencies imposed by the scheduler.
- *max_budget*: the maximum amount of service, measured in disk sectors, that can be provided to a queue once it is selected (of course within the limits of the above timeouts). According to what we said in the description of the algorithm, larger values increase the throughput for the single tasks and for the system, in proportion to the percentage of sequential requests issued. The price is increasing the maximum latency a request may incur in. The default value is 0, which enables auto-tuning: BFQ tries to estimate it as the maximum number of sectors that can be served during timeout_sync.
- *max_budget_async_rq*: in addition to the max_budget, limit, async queues are served for a maximum number of requests, after that a new queue is selected.
- *low_latency*: if equal to 1 (default value), interactive and soft real-time applications are privileged and experience a lower latency.
- A few parameters related to the low-latency heuristics are also exported.

Other parameters have the same meaning as in CFQ (see *Documentation/block/cfq-iosched.txt*).

4 Testing Applications

4.1 rt-app

rt-app is a test application that starts multiple periodic threads in order to simulate a real-time periodic load. It supports SCHED_FIFO, SCHED_RR, SCHED_DEADLINE and SCHED_OTHER.

Code is currently maintained on GitHub at:

https://github.com/scheduler-tools/rt-app

For details on software requirements and on the compilation process, we refer to the README in the source tree. We describe the basic usage here.
In order to define the taskset, it is possible to use commandline by running the command

```
# rt-app [options] -t <period>:<exec>[:policy[:affinity[:dl[:prio]]]] -t ..
```

The main options are:

- *-h*, to show the help menu;
- *-f*, to set the default policy for threads to SCHED_FIFO;
- *-r*, to set the default policy for threads to SCHED_RR;
- *-D*, time (in seconds) before stopping the threads;
- *-l*, to save logs to a different directory;
- *-g*, generate gnuplot script (require -l).

Each thread is specified by the following parameters:

- *period*, thread period in microseconds;
- *exec*, thread WCET in microseconds;
- *policy*, “f” for SCHED_FIFO, “r” for SCHED_RR, “d” for SCHED_DEADLINE and “o” for SCHED_OTHER;
- *affinity*, a comma-separated CPU index (starting from 0) e.g. 0,2,3 for first, third and fourth CPU;
- *prio*, thread priority in SCHED_FIFO/SCHED_RR/SCHED_OTHER;
- *dl*, thread deadline in microseconds (used only for plotting).

Keep in mind that on commandline it is not possible to define resources and how tasks access them. Alternatively, it is also possible to use a json to define the taskset by running

```
# rt-app <config_file>
```

See under doc/ for an example.

rt-app outputs per-thread logs (by default in the /tmp directory) listing, among other information:

- *start*, the arrival time of the instance;
- *end*, the finishing time of the instance;
- *deadline*, the absolute deadline of the instance;
- *dur.*, the execution time of the instance;
- *slack*, absolute deadline minus finishing time.

4.2 fio

fio is a tool that will spawn a number of threads or processes doing a particular type of I/O action as specified by the user. fio takes a number of global parameters, each inherited by the thread unless otherwise parameters given to them overriding that setting is given. The typical use of fio is to write a job file matching the io load one wants to simulate.

fio resides in a git repo, the canonical place is:

git://git.kernel.dk/fio.git

For details on the configuration and building process, we refer to the README in the source tree. We describe the basic usage here.

The I/O action is usually specified by means of a job file; fio parses this file and sets the action up as described, when run by the command

```
# fio job_file
```

The job file is divided into a "global section" and a "job section". The global section contains default parameters for jobs specified in the job section. Specific job definitions may override any parameter set in the global section.

Most useful parameters include (see man pages for a complete list):

- *readwrite*, the type of I/O pattern (accepted values are "read" for sequential reads, "write" for sequential writes, "randread" for random reads, "randwrite" for random writes "rw" for mixed sequential reads and writes, "randrw" for mixed random reads and writes;
- *size*, the total size of I/O for this job (fio will run until this many bytes have been transferred, unless limited by other options);
- *blocksize*, the block size for I/O units (default is 4k);
- ioengine, how the job issues I/O (some accepted values are "sync" for basic read(2) or write(2) I/O, "libaio" for Linux native asynchronous I/O, "mmap" for file memory mapping with mmap(2) and data copying with memcpy(3), "splice" for using splice(2) to transfer the data);
- *direct*, if true use non-buffered I/O (usually O_DIRECT);
- fsync, the number of I/Os to perform before issuing an fsync(2) of dirty data;
- *prioclass*, to set the priority class (see ionice(1));
- *ioscheduler*, to switch the device hosting the file to the specified I/O scheduler.

fio does not support setting the ioscheduler to BFQ yet. This can be achieved, as also described in Section 3, by writing into the scheduler file in the virtual filesystem, e.g.:

```
# echo bfq > /sys/block/<device>/queue/scheduler
```

When fio completes, it outputs show data for each thread and for each group of threads/job file and for each disk in this order, including:

- *io*, number of megabytes of I/O performed for each thread and for each group of threads;
- *clat*, completion latency minimum, maximum, average and standard deviation;
- *bw*, bandwidth minimum, maximum, percentage of aggregate;
- *aggrb*, aggregate bandwidth of threads in the group;
- *ticks*, number of ticks we kept the disk busy.

A Installation

This Appendix presents the procedure required to obtain a working copy of the Operating System prototype including the Linux kernel version 3.14 enhanced with the patches previously described (i.e. 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*).

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 prototype kernel and the tools described in Section 4.

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 2 on a Debian Wheezy, which shall be executed in a shell with root privileges.

The following commands retrieve the installation script from the repository and to execute it:

```
# wget -O http://retis.sssup.it/people/nino/Juniper/d3_2/d3_2-install.sh
# chmod +x d3_2-install.sh
# ./d3_2-install.sh
```

The installation script performs the following actions:

- define the remote folder location, if not set by the user with the JUNIPER_REPO variable;
- 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;
- download the real-time kernel, and install it in the system;
- download and install the testing tools described in Section 4.

References

[1] Luca Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[2] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, December 1998.

[3] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[4] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[5] P. Valente. Extending WF^{2}Q+ to support a dynamic traffic mix. *Advanced Architectures and Algorithms for Internet Delivery and Applications, 2005. AAA-IDEA 2005. First International Workshop on*, pages 26–33, 15-15 June 2005.

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