Project Number 318763

D3.1 – Operating System Real-Time Support Definition

Version 1.0
4 December 2013
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**

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

List of Figures

Document Control

Version	Status	Date
0.1	Document outline	4 November 2013
0.2	Complete First Draft	22 November 2013
1.0	QA for EC delivery	4 December 2013

Executive Summary

This document constitutes deliverable *D3.1 – Operating System Real-Time Support Definition* of Work Package 3 of the JUNIPER project, containing design description and algorithms for Linux real-time bandwidth reservation scheduling for big data systems.

The JUNIPER project adopts the use of a standard Linux operating system with real-time scheduling extensions applied to both processors and disk I/O. Following D1.2-SS.1, real-time guarantees will be based on resource reservations techniques. In this deliverable, we describe these techniques (see also D1.2); deliverables D3.2 and D3.6 will focus on their implementations and on the corresponding user interfaces (see D1.2-SS.2,3,4,5,7). A key part of the project will be to align OS level scheduling and resource access with that required by the real-time JVM (D3.3), by the FPGA (D3.4 and D3.5) and by user-level software libraries for big data (D3.7).

1 Introduction

Big Data Systems are systems whose goal is the processing, analysis and management of very large amount of data. Big Data systems are usually optimised for throughput, i.e. the performance of the system is measured in terms of total amount of data processed per second.

Real-Time Big Data systems are a subclass of the former in which (part of) the processing must be performed within specified time constraints. Typically, such constraints are not critical, in the sense that nothing catastrophic happens if some output data is produced late with respect to the specified constraints. However, the *Quality of Service* provided by the application depends on its ability to respect the timing constraints. For example, if the processing of an incoming stream of data is not completed within a certain time instant, data may be lost; also, end-users of the application may be disadvantaged by the late response to a request for information. In the real-time literature, such systems are often referred as *soft real-time*, as opposed to the safety critical systems which are referred as *hard real-time*. Therefore, in this deliverable, when mentioning *real-time* we will implicitly refer to soft-real-time.

Big Data applications are expected to be run on a distributed platform consisting of commodity hardware platforms, in some case augmented with special accelerating hardware. For example, most data centers are built of normal off-the-shelves PCs connected in racks. Also, typical Big Data applications run on heterogeneous hardware distributed around the world, and some of the available hardware may consist of special High Performance Computers. Therefore, it is impossible to apply here classical real-time theory which has been developed mainly for safety critical application running on dedicated hardware.

However, it is well known that real-time performance can only be guaranteed if the underlying system software provides means to control the scheduling and the allocation of resources to the executing processes. In fact, in order to guarantee that a certain processing application completes within its time bounds, it is necessary to reserve it a certain minimum bandwidth and prioritise its access to the resources (processors, disks and networks) with respect to other non real time activities. Therefore, it is important that each node is equipped with an OS that can reserve bandwidth and prioritise processes.

1.1 Open Systems

Given the heterogeneity of the hardware platforms, and the fact that we address *soft real-time* applications, it follows that we have to choose an Operating System which is widely available for different hardware platforms. The most natural choice is to choose Linux as OS, because: 1) it is open source, so it is possible to modify the OS for our goals; 2) it is available on almost every hardware platform and on most processors in the market; 3) it support a very large number of libraries and tools for the development of distributed applications, and in particular it already supports many of the tools and libraries used for developing Big Data Systems.

Linux is however a complex OS. It is multi-task and multi-user, and many different processes can run concurrently with our application. Some of these are system processes (such as daemons, kernel threads, etc.), some are servers needed to provide and maintain important services. In many

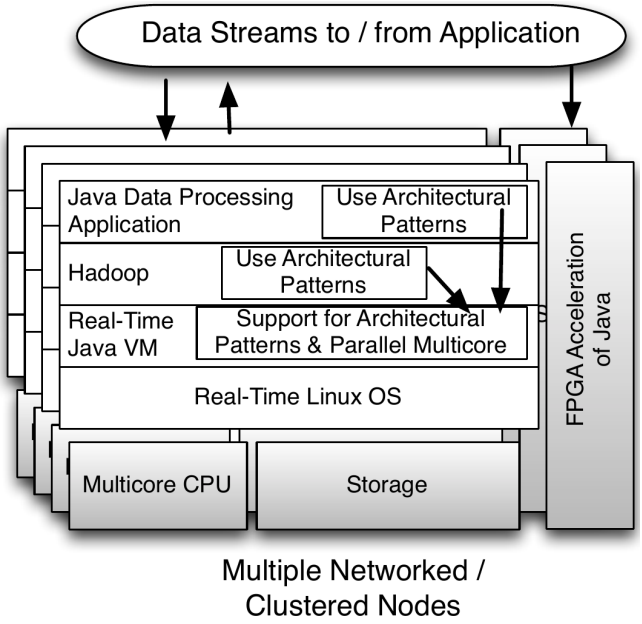

Figure 1: The architecture of a JUNIPER distributed system

cases, most of these processes cannot be easily removed or suspended while we execute our application. Once again, we cannot make the assumption that we have a dedicated system for running our applications.

For this reason, in the JUNIPER project we consider *open real-time systems* where tasks can dynamically enter or leave the system at any time. Therefore, a run-time admission control scheme is needed to make sure that the new tasks do not jeopardise the guarantees that we made to the already existing tasks. The architecture of a JUNIPER distributed system is shown in Figure 1: the OS must support real-time execution for all components of the JUNIPER system.

1.2 Resource Reservations

Since our system environment is heterogeneous, and every node may need to run many different concurrent tasks, not all of them part of our application, it is necessary to *reserve* resource bandwidth and isolate task execution. In particular, for robustness, security and safety issues, it is necessary to isolate and protect the temporal behaviour of one task from the others. Resource Reservations [21] were proved as effective techniques to achieve the goals of temporal isolation and real-time execution in open systems.

Resource reservation techniques have initially been designed for the execution of independent tasks on single processor systems. Recently, they were extended to cope with hierarchical scheduling systems [7, 23, 13], for multi-core systems [5, 24], and with tasks that interact with each other using locks [8, 18, 6].

According to the Resource Reservation Framework, when a task or a group of tasks is activated in the system, it goes through an admission control procedure. The task asks the system to be reserved

a fraction of the resources in the system. For example, it may ask for one full core plus 50% of the bandwidth of a second processing core. In addition, it may require a fraction of the bandwidth of the disk for reading data from the files. The system accepts the request if there are enough free resources for accommodating it. In such a case, the group of tasks is accepted in the system and a *contract* is established: from now on the tasks are guaranteed to receive the reserved amount of resources whenever necessary. If instead the request cannot be accommodated, the task cannot be activated.

At run-time it is then necessary to schedule the task requests so that the *contract* is respected. This means that a proper scheduling strategy must dispatch requests to the processing resources in the proper order so that the contract is respected.

In this project, we will use the framework presented by Lipari and Bini [14] for providing processor reservations to group of tasks in multi-core systems. An overview of such technique is presented in Section 4, where we also present SCHED_DEADLINE, a resource reservation scheduler for Linux, and we describe how we will modify it to implement the framework of [14].

Then in Section 5 we present the M-BWI protocol [6] for dealing with tasks that access shared resources using mutually exclusive semaphores, and we present an early prototype implementation in Linux [19].

Since we are dealing with Big Data systems, it is very important to provide means to read large amount of data from mass storage, in particular from magnetic disks and Solid State Drives (SSDs). Providing Resource Reservations on magnetic disks is particularly challenging for characteristics of the specific technology. Also, for large amount of data magnetic disks are still the dominant technology in terms of cost and availability. In Section 6 we present BFQ [29], a scheduling algorithm for disks that we will utilise in the JUNIPER Project for guaranteeing real-time performances in disk access.

2 Contributions in JUNIPER

We divide our contributions to Linux real-time bandwidth reservation scheduling for big data systems in three parts.

First, we will implement the Bounded-Delay Multi-partition (BDM) model ([14]) described in Section 4, in order to provide hierarchical CPU reservation. SCHED_DEADLINE currently supports CPU reservation at a task level only; we will extend SCHED_DEADLINE to allow CPU reservation for group(s) of real-time tasks (i.e., SCHED_FIFO and SCHED_RR tasks).

Second, we will implement the Multiprocessor BandWidth Inheritance (M-BWI) protocol ([6]) described in Section (5), to extend the CPU reservation framework in the presence of shared resources. SCHED_DEADLINE currently adopts deadline inheritance, without control on the bandwidths; we report on an early implementation of the M-BWI protocol in Section 5.

Third, we will use the Budget Fair Queueing (BFQ) disk scheduler ([29]) described in Section 6, in order to provide disk I/O reservations. BFQ currently provides fair allocation of I/O bandwidth, the amount of bandwidth actually reserved depending on the number of requests and their relative priorities; we will compute the set of priorities to guarantee the reservations.

These works will be developed on top of Linux with the RT_PREEMPT patch ([9]). This patch add preemption points to the Linux kernel, by replacing most kernel spin-locks with mutexes that support Priority Inheritance (PI) and by moving interrupts and software interrupts to kernel threads. The RT_PREEMPT patch focus is, in short, make the Linux kernel more deterministic.

3 Related work

As described in Section 2, our contributions within the JUNIPER project will be focused on CPU and on disk I/O reservations. A related problem concerns with the possibility to extend the techniques described in the following sections to other resources, such as memory and network I/O and, more generally, with the connections existing between different types of reservations.

On this regard, we mention that the BFQ algorithm described in section 6 has been modified and adopted in the context of network (packets) scheduling. Notably, an approximation of the algorithm, called QFQ+ ([28, 20]), has been developed and is in mainline Linux since 3.8. The QFQ+ scheduler is considered to be the state-of-the-art in the field of network I/O reservation.

On the contrary, work related to the implementation of efficient memory (bandwidth) reservation scheduling algorithms is still at an early stage. This could be ascribed to the complexity of the problem: locality of memory references of each application ([4, 30, 11]), caches contention ([10, 17, 31]), memory controller congestion ([2, 16]) and efficient monitoring and storing of these parameters are only some of the aspects that need to be considered when designing such an algorithm.

We conclude by observing that, even if the *value* of the reservation (e.g., the fraction of CPU time or the share of disk I/O) is not affected by the memory bandwidth, the "quality" of the reservation certainly is (e.g., remote memory accesses can double the cost, in CPU cycles, of local ones and can degrade both disk and network I/O performance due to caching/buffering). The choice of a particular (disk) blocks allocation policy (e.g., the choice of the filesystem) can also affect I/O performance (however, there is no a-priori incompatibility between a disk I/O scheduler and a filesystem, given to the modularity of the Linux kernel). No attempt will be made to formalise these concepts in the following sections and we will generally ignore related issues in this document.

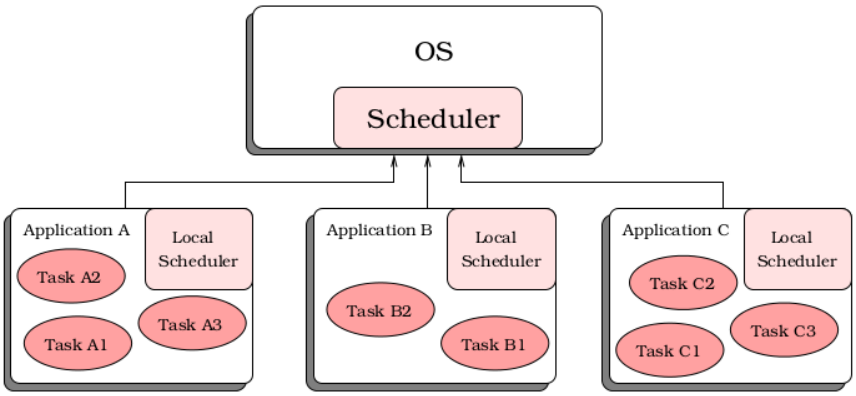

Figure 2: Hierarchical scheduler structure.

4 Processor Reservation

Thanks to the recent advances in the field of computer architectures, it is now common practise to concurrently execute different real-time applications in the same system. The motivation for executing different applications in the same system is in cost reduction and in the re-use of legacy applications on new, faster systems.

When executing many real-time applications in the same system, a problem is how to schedule these applications and guarantee at the same time that their timing requirements are not violated. One simple way to solve this problem is to use a unique scheduling paradigm for the whole system and design all applications according to the chosen paradigm.

However, it is often necessary to use an already implemented application "as it is", without going back to the design phase. Moreover, if an application is already working well within an old platform with its own scheduler, it may be expensive to change or to redesign from scratch its code to *compose* it with new applications.

One way of composing existing applications with different timing requirements is to use a two-level scheduling paradigm (see Figure 2): at the *global level*, a scheduler selects which application will be executed next and for how long. Each application then possesses a *local scheduler* that selects which task will be scheduled next.

The global scheduler assigns each application a fraction of the total processor time according to certain policy. Moreover, a real-time global scheduler must "protect" one application from all others, by ensuring that if an application is requiring more than expected, it does not compromise the performance guaranteed to the other applications.

Inside every application, a dedicated *local scheduler* decides which among the application's tasks should execute. The local scheduler has visibility only of the application's tasks, and it is invoked only when the global scheduler allocates the resource to the application. In this way, each application can specify its own scheduling policy, which can be in general different from the scheduling policy of the other applications. Therefore, using this two-level scheduling approach, it is possible to completely separate the concerns:

- The global scheduler is in charge of implementing the resource reservations scheme, by assigning the resources to the applications, so that temporal isolation and performance guarantees are respected;
- The local scheduler is invoked when the resource is effectively allocated to the application, and decides which of the tasks will execute on each resource.

Notice that we are not assuming any synchronization between the global scheduler and the local one. Doing so, the design of the two adjacent layers is completely decoupled, allowing a great degree of freedom in the selection of the scheduling algorithms and the construction of a hierarchical scheduler structure as deep as needed. On the other hand, the price of this flexibility is that we may lose some optimality in the scheduler.

In Section 4.1 we will first discuss the concept of global scheduler and *time partition*, and we will also present the notation used in this deliverable.

4.1 Time multi-partitions

In the course of this section, the overall system is composed of a set of real-time applications that run concurrently onto a multiprocessor M constituted by m processors. An *application* A is a set of n independent sporadic tasks $\{\tau_1, \ldots, \tau_n\}$. We denote the set of applications by $\mathcal{A}$. A sporadic task is a cyclically activated thread that is identified by three parameters, $\tau_i = (C_i, D_i, T_i)$. Every time a task is activated, a *job* must be executed. The *minimum inter-arrival time* T_i is the minimum separation between two consecutive jobs of τ_i. Each job of τ_i has a *computation time* C_i and must be completed within a *deadline* D_i from its activation.

Due to the presence of the global scheduler, the processing resource may or may not be assigned to the application. It is then useful to define a *time multi-partition* of an application, as the time the global scheduler assigns to it in each processor.

Definition 1. *A* time multi-partition P *is a countable multiset of intervals,*

$$P = \{[a_i, b_i)\}_{i \in \mathbb{N}}.$$

A *static global scheduler* pre-computes the multi-partitions off-line and, at run-time, a dispatch mechanism will make use of a simple table to allocate the resource. Conversely, an *on-line global scheduler* algorithm uses its rules for dynamically allocating the resource. Since we are dealing with *open real-time systems*, and hence we cannot know a-priori how many applications will run in the system and their characteristics, we will only consider on-line scheduling policies.

For a given multi-partition, we define the minimum amount of time that is available to the application in every interval of length t with a given parallelism.

Definition 2. *Given a multi-partition* P*, we define the* level-j supply function of P*,* $Y_{j,P}$*, as*

$$Y_{j,P}(t) = \min_{t_0 \geq 0} \int_{t_0}^{t_0+t} \min\{j, \gamma_P(x)\} d\,x, \tag{1}$$

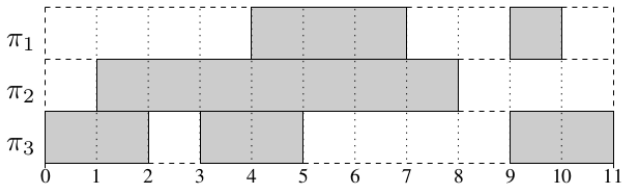

Figure 3: From resource schedule to supply functions.

where γ_P *is the* characteristic function of P*,*

$$\gamma_P(t) = \sum_{[a_i,b_i)\in P} \gamma_{[a_i,b_i)}(t). \tag{2}$$

and $\gamma_{[a_i,b_i)}$ *is the characteristic function of the interval* $[a_i, b_i)$*:*

$$\gamma_{[a_i,b_i)}(t) = \begin{cases} 1 & \text{if } t \in [a_i, b_i), \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

The supply functions provide important information about the multi-partitions. For example, if a task requires C units of computation every period T then it is schedulable on the multi-partition P only if $Y_{1,P}(T) \geq C$. This condition can in fact be read as *the multi-partition* P *provides at least* C *units of computation in any possible interval* T *long with parallelism 1.*

The set of rules (the algorithm) which, together with the events that occur on-line, determines the multi-partition is called a *server*. In our model stack the server is responsible for dividing the total processing resource that is available to the application among the different applications with their local scheduling algorithm. To generalise the supply function to the server mechanism, we introduce the following definitions.

Definition 3. *Given a server* S*, we define* $\ell(S)$ *as the set of multi-partitions* P *that can be* generated by *the server mechanism* S*.*

Definition 4. *Given a server* S*, the* level-j supply function of S*,* $Y_{j,S}$*, is defined by*

$$Y_{j,S}(t) = \min_{P\in\ell(S)} Y_{j,P}(t). \tag{4}$$

Now the schedulability of a task on a server S can be indeed checked off line. In fact, if we have a one-task system and $Y_{1,S}(T)$, which is the minimum amount of time provided in every time interval T-long with parallelism 1, is greater than or equal to the maximum possible time requested by the task, then the task is schedulable on the server S.

To clarify these definitions we propose an example in Figure 3. Suppose that $m = 3$ and that in the interval $[0, 11]$ a given server provides resource in accordance to the schedule drawn in gray. In this case $Y_1(11) = 10$ because there is always at least one processor available in $[0, 11]$ except in $[8, 9]$. Then $Y_2(11) = 16$, that is found by summing up all the resource except one with parallelism

3 provided in $[4, 5]$). Finally, $Y_3(11) = 17$ that is achieved by summing all the resources provided in $[0, 11]$.

In the following sections we sometimes identify a server with the sequence of its supply functions that we denote by $\mathcal{Y}$.

Definition 5. *An* interface I *is a predicate on the set* $\mathcal{Y}$ *of servers. We will denote the subset of all possible server that match the predicate with the same symbol* I.

4.2 Periodic server model

We now consider the class of algorithms that can be described by the *periodic server* abstraction. Many algorithms have been proposed that belong to the class of periodic servers [27, 25, 26, 1].

Intuitively, according to a periodic server algorithm, each application is assigned a pair (Q, P), with the meaning that the application will receive Q units of execution every P units of time. The global scheduling mechanism decides when to schedule the applications; the selected application, by using the local scheduling mechanism, decides which task will be executed next.

More precisely, a periodic server is characterized by two absolute parameters (Q, P), where Q is the *maximum budget* and P is the server *period*, two relative parameters (q, d), where q is the *current budget* and d is the server *deadline*, plus a local scheduling algorithm.

In this deliverable, and in the course of the JUNIPER project, we will use the server algorithm described in [13], and later used also in [14]. We report here the server rules for completeness, and remand to the cited papers for the proofs of correctness.

1. Initially, $q = 0$, $d = 0$ and the server is *inactive*.
2. When a task is activated at time t, if the server is inactive, then $q = Q$ and $d = t + P$, and the server become *active*. If the server is already active, then q and d remain unchanged.
3. At any time t, the global scheduling algorithm selects one active server. When the server is selected, it executes the first task in its ready queue (which is ordered by the local scheduling policy).
4. While some application task is executing, the current budget q is decremented accordingly.
5. The global scheduler can *preempt* the server for executing another server: in this case, the current budged q is no longer decremented.
6. If $q = 0$ and some task has not yes finished, then the server is *suspended* until time d; at time d, q is recharged to Q, d is set to $d + P$ and the server can execute again.
7. When, at time t, the last task has finished executing and there is no other pending task in the server, the server yields to another server. Moreover, if $t \geq d - q\frac{P}{Q}$, the server become inactive; otherwise it remains active, and it will become inactive at time $d - q\frac{P}{Q}$, unless another task is activated before.

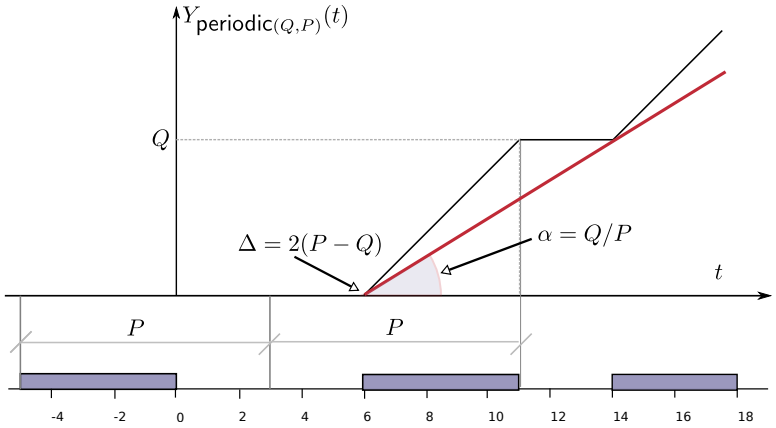

Figure 4: Periodic server.

In Figure 4, we plot the supply function $Y_{\text{periodic}(Q,P)}(t)$ of a server with parameters $Q = 5$ and $P = 8$. Since we do not know the global system load, we consider the worst-case situation, when the application tasks are activated just after the budget is exhausted. In this case, the first instant of time at which they will receive execution is at $2(P - Q)$.

Theorem 1 ([14]). *Let $m = 1$ and suppose that the system consists of a set of periodic servers. Then, given such a server S with parameters (Q, P), and defined $k = \left\lceil \frac{t-(P-Q)}{P} \right\rceil$, the (level-1) supply function $Y_{periodic(Q,P)}$ of S is given by:*

$$Y_{periodic(Q,P)}(t) = \begin{cases} 0 & \text{if } t \in [0, P-Q], \\ (k-1)Q & \text{if } t \in (kP-Q, (k+1)P-2Q], \\ t-(k+1)(P-Q) & \text{otherwise.} \end{cases} \tag{5}$$

In JUNIPER we will use the SCHED_DEADLINE [12] patch that implements the server rules above and uses EDF as the global scheduling policy (i.e., it picks the server with earliest absolute deadline d). Currently, SCHED_DEADLINE instantiates such a periodic server for each SCHED_DEADLINE task. In the course of the project, we will extend the periodic server mechanism to allow *groups* of SCHED_FIFO or SCHED_RR tasks (for more details, see 4.5).

4.3 Multi-core model

Notice that the server algorithm described in the previous section provides a time multi-partition with level of parallelism $m = 1$. If we need a multi-partition with a level of parallelism $m > 1$, we can use several such servers with different Q and P.

How are such servers scheduled by the global scheduling algorithm? There are mainly two possibilities: servers can be scheduled by a global scheduling algorithm with migration; or servers can be statically assigned to physical processors, and then scheduled according to a partitioned single-processor scheduler algorithm. In order to reduce overhead at the global level (see description below), we chose the second approach. This model has already been presented in previous papers, and in particular in [14], and is depicted in Figure 5.

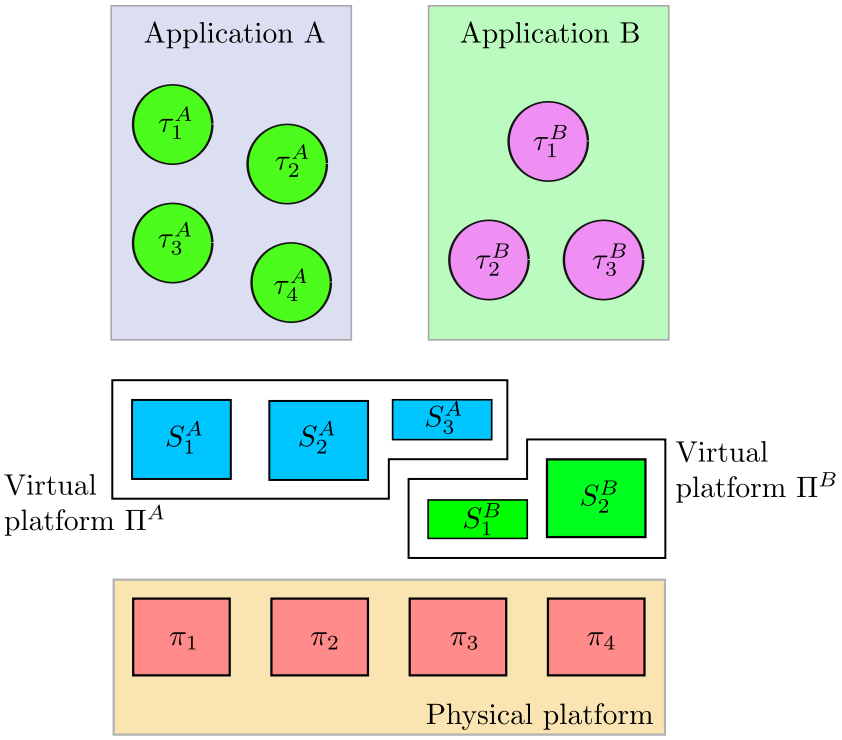

Figure 5: Pictorial representation of two virtual platforms on a 4-core processor system

The idea is that one application, consisting of a group of tasks, is assigned a *virtual platform* consisting of a number of *virtual processors*. Each virtual processor is implemented through the periodic server described in the previous Section with parameters (Q, P).

A server is statically assigned to a *physical* processor (core), and provides a percentage of the bandwidth on that processor equal to $\frac{Q}{P}$. The global scheduler, hence, is completely partitioned; on each physical node, a single-processor global scheduler selects which server will execute at each instant. The application local scheduler, instead, can be migrative or partitioned: generally, an application task can execute on any of the server its virtual platform consists of. Therefore, sharing of data structures for scheduling is restricted to local scheduling, and therefore we expect less overhead at the global level.

4.4 Schedulability in a virtual platform

In order to select the server parameters for an application, we first need a way to analyse the schedulability of an application on a given virtual platform. To simplify the analysis, we will adopt the (α, Δ) model proposed by Feng and Mok [7], for its simplicity and effectiveness. In this model, parameter α represents the server bandwidth, whereas parameter Δ represent the maximum initial delay of the multi-partition.

α_j can be easily obtained as the average slope of $Y_{j,S}$:

$$\alpha_j = \lim_{t \to +\infty} \frac{Y_{j,S}(t)}{t} = \sup_t \frac{Y_{j,S}(t)}{t}. \tag{6}$$

The limit above exists, since each $Y_{j,S}$ is super-additive. The quest for the value of Δ_j requires some more efforts. Informally speaking, once we have computed α_j, the delay Δ_j is the minimum amount we must right-shift the line $\alpha_j t$ to be completely below $Y_{j,S}(t)$. Formally:

$$\Delta_j = \sup\{d \leq 0 : \exists t \geq 0 \text{ s.t. } Y_{j,S}(t) \leq \alpha_j(t-d)\}. \tag{7}$$

For example, in case $m = 1$ and for a periodic server with parameters (Q, P), its follows from Theorem 1 and from the definitions above that:

$$\alpha = \frac{Q}{P}, \quad \Delta = 2(P-Q), \tag{8}$$

and inversely,

$$P = \frac{\Delta}{2(1-\alpha)}, \quad Q = \alpha P. \tag{9}$$

The benefit of the generalisation to the (α, Δ) model is the possibility to reduce the number of designs and to map the abstract (α, Δ) server to a specific model only at a later stage. These benefits come at the cost of worsening a bit the design since, as it follows from the previous definitions, for any given server S it holds

$$\widetilde{Y}_{\text{periodic}(Q,P)}(t) := \max\{0,\, \alpha(t-\Delta)\} \tag{10}$$

Given these equations, the whole virtual platform can be described by the a set of supply bound functions: Given a virtual platform with m servers $\Pi = \{S_1, \ldots, S_m\}$, with parameters $(Q_j, P_j), 1 \leq j \leq m$, the *$j$-level supply bound function* is defined as:

$$Y_j(t) = \min_{\Gamma \in \Pi_j} \sum_{S_k \in \Gamma} \widetilde{Y}_{\text{periodic}(Q_k,P_k)}(t) \tag{11}$$

where Π_j is the set of all possible subset of S with j elements.

We now report a schedulability test for a general application which uses Fixed-Priority (FP) as its local scheduling policy. The test is based on the concept of *interfering workload* W_i, that informally represents an upper bound to the amount of processing resource that can be requested by tasks with priority higher than task i's priority. While choosing other schedulability tests is possible, the proposed result has the advantage of highlighting the constraint on the server model.

We start from the formal definition of the notion of schedulability test.

Definition 6. *Given a scheduling policy P, a P-*schedulability test *S is a boolean function on $\mathcal{A} \times \mathcal{Y}$ that satisfies the following property:*

$$S(A, Y) = \text{TRUE} \implies A \text{ is } P\text{-schedulable on } Y.$$

Given an interface I, we say that Y_{wc} is a *worst-case server for I w.r.t.* the P-schedulability test S, when:

$$Y_{\text{wc}} \in I, \tag{12}$$

$$S(A, Y_{\text{wc}}) \implies S(A, Y), \quad \forall Y \in I, \forall A \in \mathcal{A}. \tag{13}$$

The main result of this section follows.

Theorem 2 ([3]). *Let* $P =$ FP *be any fixed-priority scheduling policy, and define the maximum* interfering workload *that can be experience by task* τ_i *in the interval* $[0, D_i]$ *as:*

$$W_i = \sum_{j \in \text{hp}(i)} W_{ji}, \tag{14}$$

where $\text{hp}(i)$ *denotes the set of indices of tasks with higher priority than* τ_i*, and* W_{ji} *is the amount of interfering workload caused by* τ_j *on* τ_i*, that is*

$$W_{ji} = N_{ji}C_j + \min\{C_j, D_i + D_j - C_j - N_{ji}T_j\} \tag{15}$$

with $N_{ji} = \left\lfloor \frac{D_i + D_j - C_j}{T_j} \right\rfloor$.

Then the function $S_{\text{FP}} \colon \mathcal{A} \times \mathcal{Y} \to \{\text{FALSE}, \text{TRUE}\}$*, defined by:*

$$S_{\text{FP}}(A, Y) = \bigwedge_{i=1}^{n} \bigvee_{k=1}^{m} kC_i + W_i \leq Y_k(D_i), \tag{16}$$

is a FP-schedulability test.

The meaning of Theorem 2 is that "the amount of computation time needed must be smaller than or equal to the time supplied by the server". We address the reader to the cited work for a detailed proof of the previous theorem.

Once defined the server model and identified the domain of the servers which can guarantee the schedulability of the given applications, we may want to impose other constraints on the server parameters. The next section elaborates on this.

4.5 Server Interface

In this section we propose a method to design a server. Our approach tries to combine simplicity (by limiting the number of parameters in the design space) and schedulability; however, it certainly introduces some resource waste. The main definition of this section is given below.

Definition 7. *An interface* I *is a* bounded-delay multi-partition *(BDM) interface when there exist a non-negative real number* Δ *and a* m*-tupla* $[\beta_1, \ldots, \beta_m]$ *such that*

$$0 \leq \beta_k - \beta_{k-1} \leq 1, \quad \forall k = 1, \ldots, m, \tag{17}$$

$$\beta_k - \beta_{k-1} \geq \beta_{k+1} - \beta_k, \forall k = 1, \ldots, m, \tag{18}$$

and the following two statements are equivalent:

$$Y \in I, \tag{19}$$

$$Y_k(t) \geq \beta_k (t - \Delta)_0, \quad \forall k = 1, \ldots, m, \forall t \geq 0. \tag{20}$$

For notational convenience, we define $\beta_0 = 0$ *and* $\beta_k = \beta_m$ *for all* $k > m$*.*

The main property of the BDM interface is that it enables a simple characterisation of the subset of servers I.

Theorem 3 ([15]). *Consider the sequence of m bounded-delay uni-processors functions:*

$$Y = \left(Y_{periodic(Q_k,P_k)}(t)\right)_{k=1}^{m}, \tag{21}$$

with

$$\Delta = 2(P_k - Q_k), \quad \alpha_k = \frac{Q_k}{P_k}$$

and with $(\alpha_k)_{k=1}^{m}$ a non-increasing sequence.
Further, let $I = \{\Delta, (\beta_1, \ldots, \beta_m)\}$ be a BDM interface. Then $Y \in I$ if:

$$\sum_{j=1}^{k} \alpha_j \geq \beta_k, \quad \forall k = 1, \ldots, m. \tag{22}$$

Moreover, the server $Y = \left(Y_{(\alpha_k,\Delta)}\right)_{k=1}^{m}$, where

$$\alpha_k = \beta_k - \beta_{k-1}, \quad \forall k = 1, \ldots, m \tag{23}$$

is a worst-case server for I w.r.t. the FP-schedulability test defined by the equation (16).

To better clarify the BDM interface model, we present a simple example. Let us suppose that $m = 3$ and that our application presents a BDM interface $I = \{6, (0.7, 1.2, 1.4)\}$. From Theorem 3 it follows that the worst-case server is $S^{\text{WC}} = \{(0.7, 6), (0.5, 6), (0.2, 6)\}$. Moreover, thanks to formula 22 it is possible to move some bandwidth from processor 3 to processor 2 achieving, for example, a server $S' = \{(0.7, 6), (0.7, 6)\}$, compatible with I.

Theorem 2 and Theorem 3 all together allow to define a strategy to solve the so called *allocation problem* for a set of real-time applications subject to FP:

> Given a set of real-time applications subject to FP, how to instantiate the corresponding servers and allocate them onto the multiprocessor M?

Any solution of the allocation problem depends on the specific requirements of the application, the hardware platform, etc. Let $(A^\sigma)_{\sigma=1}^{\ell}$ be a sequence of real-time applications. The *feasibility requirement* and the *schedulability requirement* are, respectively:

$$U_k \equiv \sum_{\sigma=1}^{\ell} \alpha_k^\sigma \leq 1, \tag{24}$$

$$\bigwedge_{i=1}^{n^\sigma} \bigvee_{k=1}^{m} kC_i^\sigma + W_i^\sigma \leq \sum_{j=1}^{k} \alpha_j^\sigma (D_i^\sigma - \Delta^\sigma)_0, \tag{25}$$

for each free indexes, where we extended the notation to each application.

i	C_i	T_i	D_i	W_i
1	1	6	6	0
2	15	27	27	6
3	9	52	52	50

Table 1: A simple real-time application.

Other requirements can add *flexibility* to the allocation problem by imposing limits on the resources used by the single applications,

$$\alpha_k^\sigma \leq B_k^\sigma, \tag{26}$$

$$U^\sigma \equiv \sum_{k=1}^{m} \alpha_k^\sigma \leq B^\sigma, \tag{27}$$

or on the resources used by the set of all applications,

$$U_k \leq B_k, \tag{28}$$

$$U \equiv \sum_{\sigma=1}^{\ell} U^\sigma \leq B. \tag{29}$$

Note that the schedulability requirement (25) can be written as a mixed-integer programming (MIP) problem in $n^\sigma \times m$ binary variables and m real variables; hence it is possible to reduce the allocation problem to a particular MIP problem.

As an example, consider the application whose parameters are reported in Table 1 and whose interfering workloads were computed according to equation (14). Setting $m = 2$ and $\Delta = 2$, we can write the feasibility requirement and the schedulability requirement as follows:

$$\begin{aligned}
\alpha_1 &\geq 0.25\, x_{1,1} \\
\alpha_1 + \alpha_2 &\geq 0.5\, x_{1,2} \\
\alpha_1 &\geq 0.84\, x_{2,1} \\
\alpha_1 + \alpha_2 &\geq 1.44\, x_{2,2} \\
\alpha_1 &\geq 1.18\, x_{3,1} \\
\alpha_1 + \alpha_2 &\geq 1.36\, x_{3,2} \\
\alpha_1 &\geq \alpha_2 \\
x_{1,1} + x_{1,2} &\geq 1 \\
x_{2,1} + x_{2,2} &\geq 1 \\
x_{3,1} + x_{3,2} &\geq 1,
\end{aligned}$$

where $\alpha_k \in \mathbb{R} \cap [0,\, 1]$ for $k = 1, 2$, $x_{i,k} \in \{0,\, 1\}$ for $i = 1, 2, 3$ and $k = 1, 2$.

The last three equations in the previous system, one for each task, are used to express the disjunctions in the schedulability requirement, by ensuring that at least one of the corresponding two inequalities

is satisfied. In our case, taking the total utilization $U := \alpha_1 + \alpha_2$ as the objective function (to be minimised), we obtain an optimal solution $(\alpha_1, \alpha_2) = (0.84,\ 0.52)$ with optimal value $U = 1.36$.

In the course of the JUNIPER project, we will implement the BDM model. For any given group of SCHED_FIFO or SCHED_RR tasks, the user will be able to define a BDM server by specifing the parameters $\{(Q, P)\}_{k=1}^{m}$ of the uni-processor servers. These uni-processor servers will be implemented following the periodic server rules. As in SCHED_DEADLINE, EDF will be used as the global scheduling policy, while FIFO or RR will be used as the local ones.

5 Reservations and Shared Resources

In the previous section we assumed tasks to be independent among them; we now relax that assumption and start dealing with tasks that interact through shared resources (on top of a shared memory system).

Tasks can access shared memory using mutually exclusive semaphores, often referred to as *mutex*es. The portion of code in a task between a *lock* and an *unlock* operation on a mutex is called *critical section*. If a task needs to enter a critical section, it may be blocked by the fact that another task has already locked the corresponding mutex: this latter task is called *lock owner*, or *lock holder*. In real-time systems it is important to compute for how long, in the worst case, a task may remain blocked on a lock.

A *priority inversion* happens when a task is blocked by a low priority task. Without a proper protocol to control access to critical sections, the duration of the priority inversion may become too long, or even unbounded; for this reason, the *priority inheritance protocol* (PIP) [22] has been proposed as a simple and effective way to reduce priority inversion. According to the PIP, when a task is blocked on a lock, the lock owner *inherits* the highest between its priority and the priorities of the blocked tasks. In this way, it cannot be preempted by medium priority tasks, thus reducing the blocking time.

Adding mutexes and critical sections to the system model makes the schedulability analysis of Section 4.2) more complex, as it is now necessary to compute the maximum blocking time for each task. To compute the maximum blocking time, it is in fact necessary to know the duration of the critical sections of all the tasks in the system. In an open system, however, it is not possible to ask the user to specify too many detailed information on *every task*, otherwise the system becomes too difficult to use and manage.

When trying to combine resource reservations with a resource access protocol, we would like to avoid the necessity to compute the blocking time of non critical tasks, and maintain the useful property of temporal isolation. So, the goals for our resource access protocol are the following:

1. We shall not require the user to specify any parameter to run the task, other that the desired budget and period (Q, P);

2. *Temporal protection*: the performance of a task (i.e., its ability to meet its deadline) shall depend only on its parameters (Q, P), on its worst-case execution time and period, and on the duration of the critical sections of the tasks with which it interacts;

3. If we do know the worst-case execution times (and duration of critical sections) of the task and of all *interacting tasks*, it must be possible to compute (Q, P) such that the task will meet its deadline;

4. We want to do this on multi-core systems as well.

5.1 Interacting Tasks

When a task τ_j successfully locks a resource R_k, it is said to become the *lock owner* of R_k, and we denote this situation with $R_k \rightarrow \tau_j$. If another task τ_i tries to lock R_k while it is owned by τ_j, we say that τ_i *blocks* on R_k, and we denote this situation with $\tau_i \rightarrow R_k$.

The section of code between a lock operation and the corresponding unlock operation on the same resource is called *critical section*. Critical sections can be *nested*. We assume that critical are always properly nested to avoid deadlock. The worst-case execution time of the longest critical section of τ_i on R_k is denoted by $\ell(R_k)$.

In the case of nested critical sections, chained blocking is possible. A *blocking chain* from task τ_i to task τ_j is a sequence of alternating tasks and resources:

$$H_{i,j} = \{\tau_i \rightarrow R_{i,1} \rightarrow \tau_{i,1} \rightarrow R_{i,2} \rightarrow \cdots \rightarrow R_{i,\nu-1} \rightarrow \tau_j\}$$

where each task in the chain is the lock owner of the preceding resource and is blocked on the following resource. For example, the following blocking chain $H_{1,3} = \{\tau_1 \rightarrow R_1 \rightarrow \tau_2 \rightarrow R_2 \rightarrow \tau_3\}$ consists of 3 tasks: τ_3 that accesses R_2, τ_2 that accesses R_2 within a critical section nested inside a critical section on R_1, and τ_1 accessing R_1. This means that at run-time τ_1 can be blocked by τ_2, and indirectly by τ_3. In this case τ_1 is said to be *interacting* with τ_2 and τ_3.

We define the subset of tasks interacting with τ_i as follows:

$$\Psi_i = \{\tau_j : \exists H_{i,j}\}. \tag{30}$$

Moreover, we define the set of tasks that directly or indirectly interact with a resource R_k as follows:

$$\Gamma_k = \{\tau_j : \exists H_{j,h} = \{\tau_j \rightarrow \cdots \rightarrow R_k \rightarrow \tau_h\}\}. \tag{31}$$

If tasks share resources using the resource reservation paradigm, they might start interfering with each other. In fact, a special type of priority inversion is possible in such a case, due to the fact that a server may exhaust its budget while serving a task inside a critical section: the blocked tasks then need to wait for the server to recharge its budget. such situation is depicted in Figure 6, where task τ_1 suffers a long blocking time from τ_3 whose budget is exhausted at time $t = 4$, while in a critical section on mutex M.

In the next section we describe the Multiprocessor Bandwidth Inheritance (M-BWI) protocol. The ultimate goal of this protocol is to provide bandwidth isolation between groups of non-interacting tasks in *open systems*: if $\tau_j \notin \Psi_i$, then τ_j cannot block τ_i and it cannot interfere with its execution.

5.2 Multiprocessor Bandwidth Inheritance

We informally describe here the rules of the M-BWI algorithm. A more complete and detailed description can be found in [6].

- When a task is blocked on a mutex we have several cases:

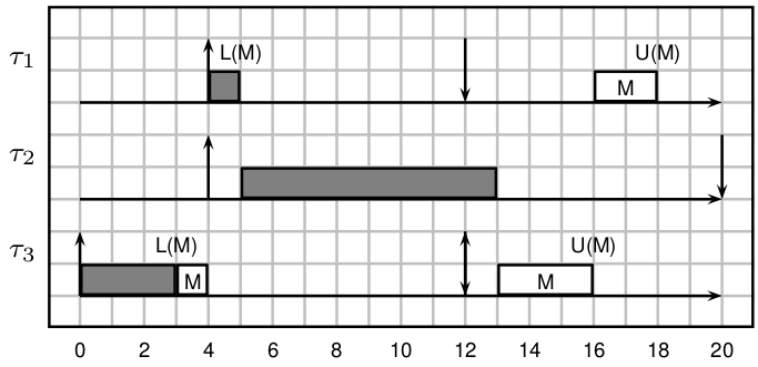

Figure 6: A task exhausts its budget while in a critical section, thus increasing the blocking time of another task.

  - if the lock-owner is itself blocked, the blocking chain is followed until a non-blocked lock-owner is found;
  - if the lock-owner is executing on another processor, the blocking task actively waits for the lock owner to release the mutex;
  - if the lock-owner is not executing, then it *inherits* the budget and scheduling deadline of the blocked task.

- Therefore, when holding the lock on a mutex, a task can have a list of pairs (budget,deadline) that it can use; it will always execute consuming the budget of the earliest deadline pair.
- When a task releases the mutex, it will discard the pairs of (budget,deadline) of the blocked tasks from its list.

Rather than going through a formal analysis of the protocol, we will present here one example that demonstrates how the protocol works. In Figure 7 we show the schedule produced by three tasks scheduled on 2 processors with migration. All of them access the same mutex M_1. At time $t = 5$ tasks τ_A and τ_B are executing on the two processors, and task τ_C is the lock-owner but it is not executing. At time $t = 6$ τ_B attempts to lock the mutex, so τ_C is woken up and inherits the budget and the deadline of τ_B, while this is blocked. At time $t = 9$ also τ_A attempts to lock the mutex, and since the lock owner is already executing on another processor, it starts a spin loop actively waiting for the mutex to be unlocked. At time $t = 14$ τ_C releases the lock, and the protocol uses a FIFO policy to wake up blocked tasks, so it wakes up τ_B. Finally, when at time $t = 18$ τ_B also releases the lock, τ_A stops its active waiting and starts to execute its critical section.

Given a task τ_i, the total amount of time that other tasks execute consuming the budget Q_i and that τ_i must actively wait for a lock release, is called interference time I_i. It is possible to compute an upper bound to the interference I_i by analyzing all blocking chains starting from τ_i.

In the general case of nested critical sections, the algorithm is rather complex; in Figure 8 we provide an example with 5 tasks on 2 processors, two resources, and nested critical sections: the request for R_1 is issued by τ_C at time 7 when it already owns R_2.

Notice that, despite the fact that both τ_D and τ_E only use R_2, they are blocked by τ_A, which uses only R_1. This is because the behaviour of τ_C establishes the blocking chains $H_{D,A} = \{\tau_D \rightarrow R_2 \rightarrow \tau_C \rightarrow$

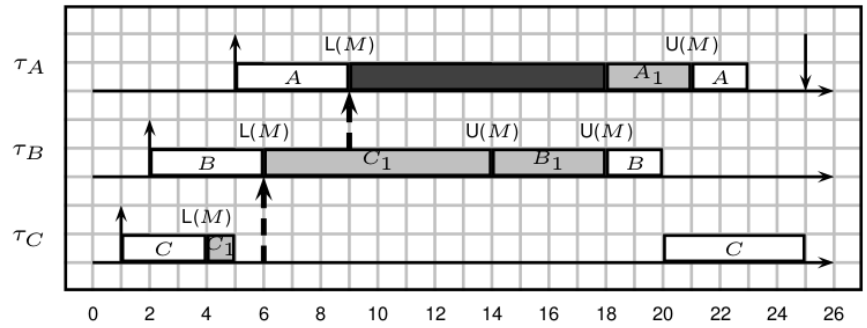

Figure 7: Example of M-BWI: τ_A, τ_B, τ_C, executed on 2 processors, that access only mutex M_1.

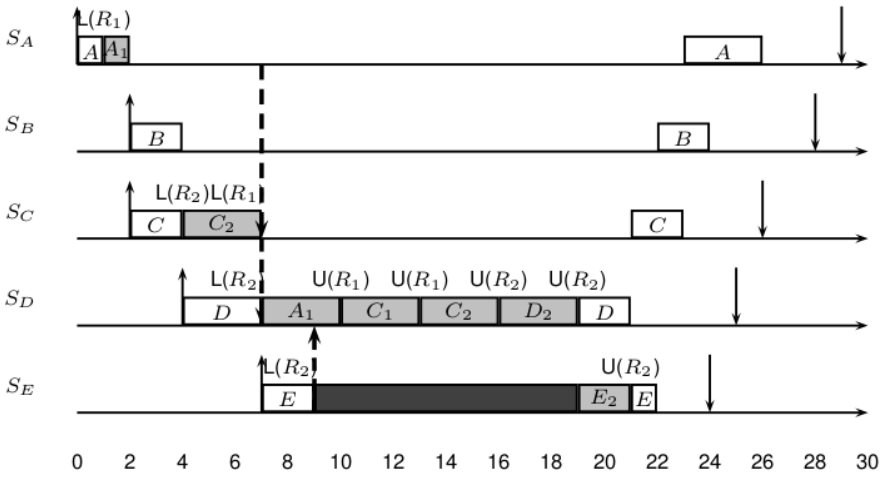

Figure 8: Second example, 5 tasks on 2 CPUs with 2 resources — task τ_C accesses R_1 inside R_2.

$R_1 \rightarrow \tau_A\}$ and $H_{E,A} = \{\tau_E \rightarrow R_2 \rightarrow \tau_C \rightarrow R_1 \rightarrow \tau_A\}$. For the same reason S_D and S_E are subject to interference either by busy waiting or executing τ_A until it releases R_1. This is a blocking-chain situation similar to what happens with priority inheritance in single processor systems.

The algorithm has two important properties:

- It guarantees **temporal isolation**: a task cannot receive interference from independent tasks;
- It is possible to compute an upper bound on interference, therefore it is possible to assign the budget Q_i to a task such that it will meets all its deadlines.

5.3 Proof of Concept

In the course of the JUNIPER project, we will extend the SCHED_DEADLINE patch by integrating it with the M-BWI protocol. That is, we will implement the M-BWI rules and we will apply them on the implementation of the periodic server algorithm that comes with SCHED_DEADLINE (see 4.2). As already stated, the M-BWI protocol is quite complex in its original formulation; thus, we developed a proof of concept implementation (see Parri et al. [19]) to foster discussion about runtime

details at an early stage. We give here a brief overview of experimental findings related to our design decisions (we refer the reader to the original paper [19] for more information), advocating that an efficient implementation of the M-BWI protocol on a Linux system is certainly viable, even if challenging.

The majority of our modifications consists in the integration of the SCHED_DEADLINE patch with Linux Priority Inheritance infrastructure (PI from now on). Indeed, SCHED_DEADLINE currently implements an approximated version of deadline-inheritance, in which the relative deadlines of the tasks are inherited but without control on the corresponding bandwidths; on the other hand, the implementation of PI was designed and optimized for fixed-priority tasks, hypothesis which can not be assumed for tasks scheduled with SCHED_DEADLINE policy.

We decided to keep at the minimum the modifications to original data structures in Linux and in SCHED_DEADLINE, and to maintain the original functions semantics. For these reasons, our implementation deviates from the original M-BWI protocol as previously described, in that it does not consider tasks busy-waiting: like for PI, a lock-holding task inherits the scheduling parameters of a blocked task, *either* if the lock-holding task is executing or not.

A first implementation has been recently presented at the 15^{th} Real Time Linux Workshop ([19]), starting a fruitful technical discussion with kernel developers and researchers. This has lead to a set of improvements that will be implemented in the next months. Thus, in the following we present some preliminary experimental results obtained with the actual implementation that validate viability of the solution on a real system.

We first performed simple tests (on an Intel®Core2™ quad-core machine (Q6600) with 4GB of RAM and running at 2.4GHz) to validate the implementation. In the first test two threads are run that operate on the same mutex (denoted as `A`). A third thread has nothing to do with the first two, its only intent is to demonstrate that the M-BWI mechanism (when enabled) works properly. All threads are restricted to execute on the same CPU.

Figure 9 shows a visual representation[1] of a run when M-BWI mechanism is disabled. Threads τ_1 and τ_3 share a resource for which mutual exclusion is achieved through the use of a mutex. Both threads are periodic with periods of $24ms$ and $72ms$ respectively (deadline are set equal to periods). τ_1 executes entirely inside the critical section for $8ms$ every period, τ_3 has an execution time of $24ms$ of which $20ms$ are spent inside the critical section. τ_2 has no critical section and executes for $8ms$ every $24ms$. Thread τ_3 is the first to be activated, after a while it acquires the mutex (`L(A)` in the figure). Then τ_1 is woken up, it tries to lock the same mutex and blocks on `A` queue, waiting for τ_3 to release it. Since it has a shorter deadline than τ_3, when τ_2 is activated it preempts τ_3 causing unexpected delay inside the critical section. τ_3 can only resume its execution once τ_2's job has finished. When τ_3 releases the mutex (`U(A)`) τ_1 executes inside the critical section and then both threads' jobs complete.

The same configuration is executed with M-BWI mechanism enabled, as depicted in Figure 10. In this case, when τ_1 is activated and blocks on mutex `A`, τ_3 can start executing in τ_1's server (highest priority server among τ_3 waiters), as highlighted with a darker blue shade in the figure. Since τ_3 has inherited also τ_1's deadline, when τ_2 arrives it doesn't immediately preempt τ_3. Mutex owner

[1] Execution diagrams in this section are created through the KernelShark (https://lwn.net/Articles/425583/) utility from execution traces extracted from the kernel via `ftrace` (Documentation/trace/ftrace.txt).

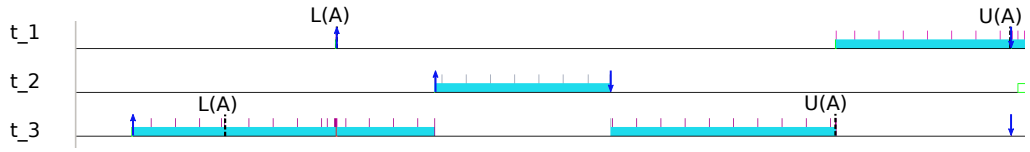

Figure 9: Two task (τ_1 *and* τ_3) sharing one resource (protected by a normal mutex). A third independent task (τ_2) arrives and preempts τ_3 even if τ_1's server has higher priority than τ_2's.

is actually preempted only when τ_1's server budget is exhausted and its deadline postponed (τ_2's deadline becoming the earliest), event `Rep(S1)` in the figure. After that the execution proceeds like in the previous situation.

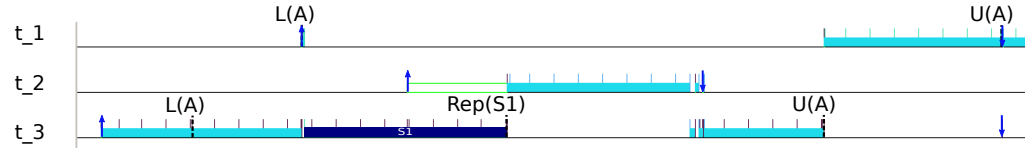

Figure 10: Two task (τ_1 *and* τ_3) sharing one resource (protected by a M-BWI-enabled mutex). A third independent task (τ_2) has to wait τ_1's server replenishment event to start executing.

The second test is performed on a 2 CPUs system. Two tasks (τ_1 and τ_3) share a resource protected by a M-BWI-enabled mutex `A` (we omit the standard case for brevity) and are free to execute on every CPU. Other two tasks (τ_2 and τ_4) are pinned each one on a different CPU and are independent from the others and between themselves (they are thought to create interference). Figure 11 zooms in a particular execution window. A job of task τ_3 arrives on CPU1 and gets scheduled, τ_3 acquires mutex `A` and enters the critical section. A few instants after a job of τ_1 arrives, τ_1 tries to acquire mutex `A` and blocks, donating its server to τ_3. The interesting part comes when τ_4 is activated. Having an earliest deadline than τ_3's original one, τ_4 should have preempted it, but its execution is delayed until τ_3 releases mutex `A` and is consequently deboosted. After this instant of time execution continues with original parameters. Without the M-BWI mechanism working τ_3 would have been preempted inside the critical section by τ_4, delaying τ_1 execution.

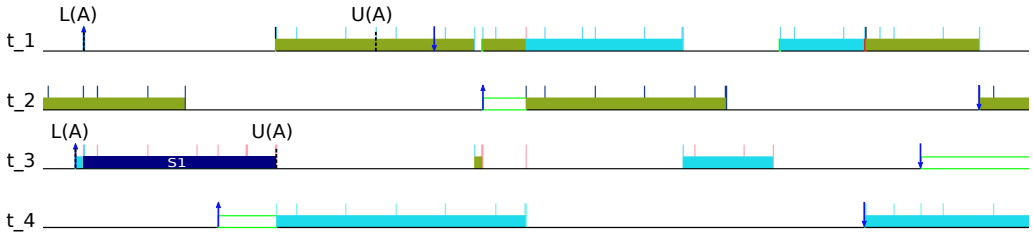

Figure 11: System with two CPUs. Two task (τ_1 *and* τ_3) sharing one resource (protected by a M-BWI-enabled mutex). Other two independent tasks (τ_2 and τ_4) are pinned each one on a different CPU.

We then measured runtime overheads comparing execution of the same benchmark with the M-BWI mechanism activated, with simple deadline inheritance, and against the stock fixed priority Linux

scheduler. We launched on each core four tasks with periods $1ms$, $25ms$, $100ms$, $1000ms$ and execution times of $0.1ms$, $2ms$, $15ms$, $600ms$. The one-millisecond tasks did not access any shared resources. All other tasks shared the same lock (one lock for each core) with an associated maximum critical section length of $1ms$, and each of their jobs acquired the lock once.

We ran the task set once using the stock Linux scheduler (SCHED_FIFO with priority inheritance enabled, called `pi` in what follows), once using the original SCHED_DEADLINE implementation (deadline inheritance, `dl`) and once with the M-BWI mechanism enabled (`bwi`), for 60 seconds each. Although the same task sets can be run with priority inheritance mechanisms turned off, we don't report results coming from that configurations here as they are hardy comparable to cases when priority inheritance (or M-BWI) is enabled. In fact, execution paths inside the kernel are completely different, and unrelated functions get called, thus making the comparison of little interest for the present discussion.

Figure 12 shows the measurements of kernel functions, obtained using `ftrace`, that could be ill-affected by the mechanism implementation:

a) `schedule()`, scheduler core, it decides which task to run next and performs the context switch;
b) `do_futex()`, `sys_futex()` system call entry point;
c) `enqueue_task_dl()`/`enqueue_task_rt()`, enqueue a task, respectively, on the `dl` or the `rt` runqueues;
d) `rt_mutex_slowlock()`, work required to acquire a mutex;
e) `rt_mutex_slowunlock()`, work required to release a mutex.

Results show that overheads of `dl` (yellow, oblique lines, boxes) and `bwi` (red boxes) are comparable. Differences between `bwi` and `pi` (blue, small circles, boxes) measurements remain in the same order of magnitude (even if `bwi` doubles `pi` in some case). These differences can be ascribed to the slightly higher complexity of `bwi` implementation, but also to the fact that tasks interactions can be modified by scheduling the same task set using different scheduling policies (this can have an impact on runtime overheads).

We have then modified the previous example in order to create longer PI chains: a new task with period $2000ms$ and execution time $700ms$, and two more mutexes were added to the above task set. Like in the previous example, there is a task that does not use any resource; no task accesses more than two mutexes, but the resulting PI chain can reach a depth of 4:

$$\tau_1 \rightarrow R_1 \rightarrow \tau_2 \rightarrow R_2 \rightarrow \tau_3 \rightarrow R_3 \rightarrow \tau_4$$

We replicated this task set 3 times for a total of 15 tasks, due to bandwidth constraint. The results displayed in Figure 13 show that the effect of the chain's depth contributes in an equivalent amount for the three implementations.

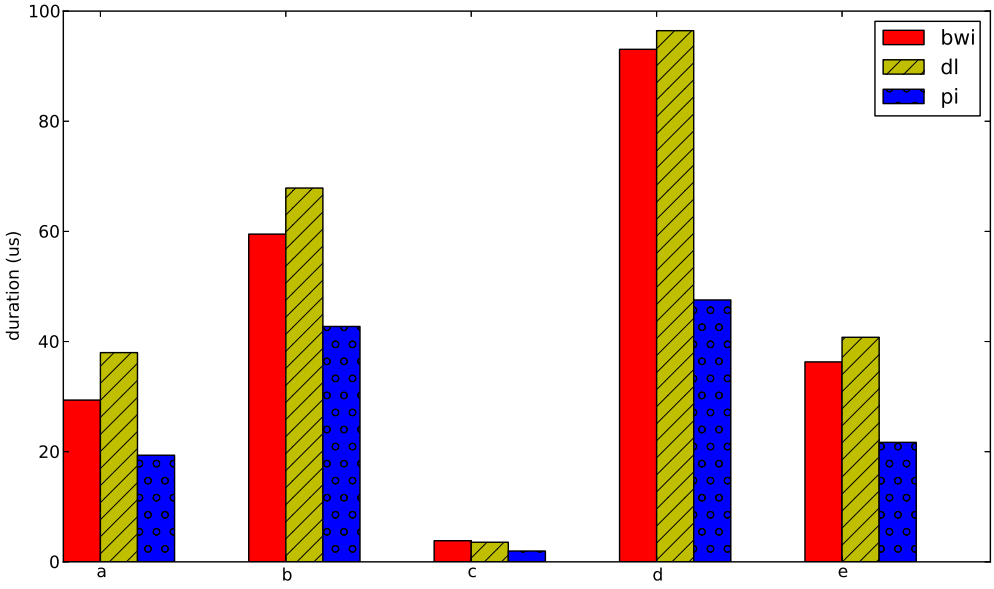

Figure 12: Kernel functions durations (in μs) from a run on a real machine.

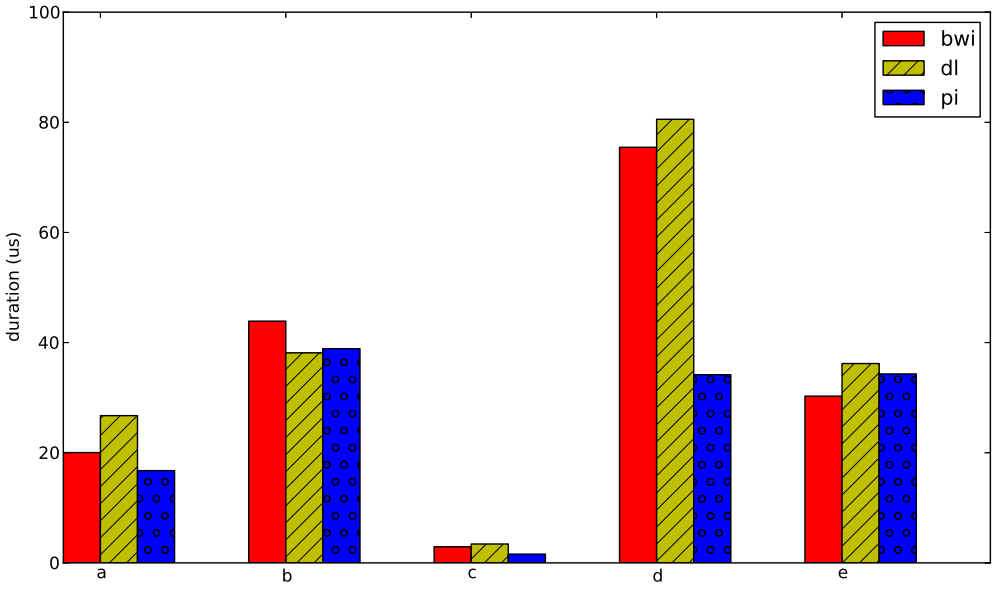

Figure 13: Kernel functions durations (in μs) with nested critical sections, from a run on a real machine.

6 Disk I/O Reservation

Managing data on disk devices is a requirement for most applications. However, meeting the opposing requirements of per-request delay and high throughput is not an easy task. Three main difficulties can be highlighted:

- the time needed to serve a request once dispatched to the disk device, is highly variable; causes are seek and rotational latencies, variation of the transfer transfer rate with the sector position, caching.
- almost any commercial controller exports these physical parameters, which makes it difficult to predict service times.
- most applications usually issue one batch of requests then block waiting for completion; the delayed arrival of these requests may jeopardize the timing constraints.

In this scenario, the simplest way to guarantee a predictable and short request service time would be *over-provisioning*. Unfortunately, it entails high purchase, powering and cooling costs, and purposely wastes disk bandwidth. A wiser solution is a better scheduling policy for disk requests.

In this section we present a proportional share timestamp-based disk scheduler, called Budget Fair Queueing (BFQ). We introduce the algorithm and its service properties to show the advantages that could provide to an Operating System like the one under development in the JUNIPER Project.

6.1 Budget Fair Queueing

The BFQ approach has been proposed by Valente and Checconi [29] to provide a fail allocation to application concurring for the disk access. In the model considered by BFQ, a *storage system* is composed by a disk device and the scheduler, as in Figure 14. The disk, is represented as a sequence of contiguous fixed size sectors, each identified by its *position* in the sequence. The disk device services two types of *disk requests*: the reading and the writing of a set of contiguous sectors from/to the disk. After receiving the *start command* or completing a request, the disk device asks for the next request to the BFQ scheduler.

The input are the requests issued by the N applications served by the storage system. The system could serve *read* and *write* requests, where each request is characterized by a size and a position on the disk. Two requests are called *sequential* if the position of the second request is just after the end of the first one. Each request has an *arrival* time at which the request is issued by the application, a *start* time when it begin to be served and a *completion* time when it is completely served by the disk device. A request is considered *synchronous* if it can be issued by an application only after the completion of its previous request. Otherwise the request is denoted as *asynchronous*.

An application is *receiving service* from the storage system if one of its requests is currently being served. Both the amount of service $W_i(t)$ received by an application and the total amount of service $W(t)$ delivered by the storage system are measured in number of sectors transferred during the time interval $[0, t]$. For each application is possible to configure the maximum budget, in number of sectors, that BFQ can assign to it.

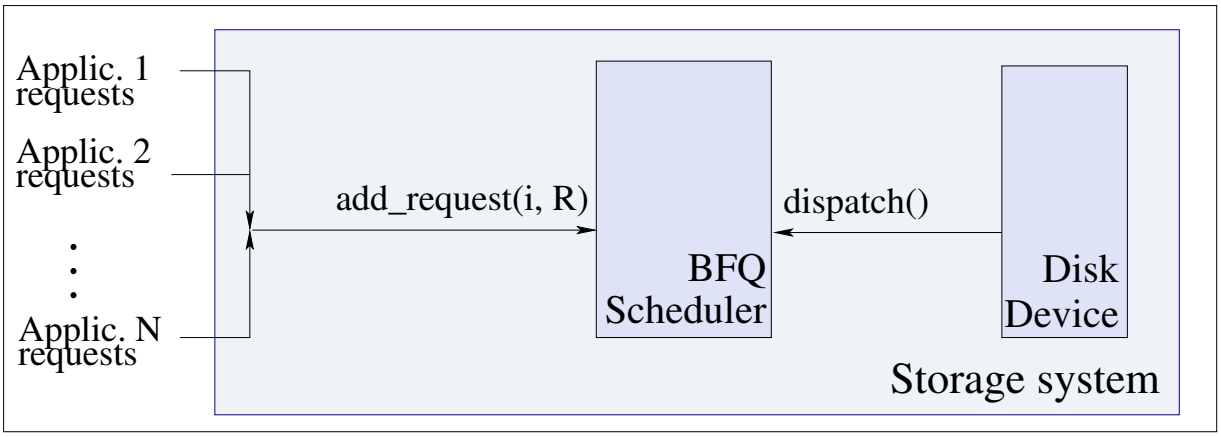

Figure 14: Reference Storage System.

The logical scheme of the BFQ scheduler is depicted in Figure 15, where solid arrows represent the paths followed by requests until they reach the disk device and dashed arrows represent flows of information or internal commands. Finally, circles represent algorithms or operations. There is a *request queue* for each application.

At any time each application has a *budget* assigned to it, measured in number of sectors. At system start-up, all applications are assigned the same *default* budget. Disk access is granted to one application at a time, denoted as the *active application*. When the new active application is selected, its current budget is assigned to a special *remaining budget* counter. Each time a request of the active application is dispatched, the remaining budget counter is decreased by the size of the request. Moreover, if the request queue gets empty, a timer is set to wait a predefined interval for the possible arrival of the next request. This may leave the disk idle, but prevent BFQ from switching to a different application if the active application is deceptively idle. However, waiting for the arrival of non-sequential requests provides no benefit. Hence, BFQ automatically reduces this interval to a very low (configurable) value for applications performing random IO. If the active application issues no new request before the timer expiration, it is deemed idle. The active application is exclusively served until either the remaining budget is exhausted, i.e., there is not enough remaining budget to serve the next request, or the application becomes idle. At this point the next budget of the application is computed. The next active application is then chosen according to the B-WF^{2}Q+, which schedules applications as a function of their budgets.

6.2 Service Guarantees

An important features of BFQ is the capability to maintain a high throughput while providing time guarantees for the incoming requests. As demonstrated in [29], the strong similarities between BFQ and the fail packet scheduling (WF^2Q+) allows to inherit the theorems regarding the guarantees. One of those theorems says that the difference between BFQ and the ideal fair allocation is bounded by a constant which is function of the application parameters. Another demonstration shows that also the time needed to complete an application request is bounded, thus its impact on the application execution time.

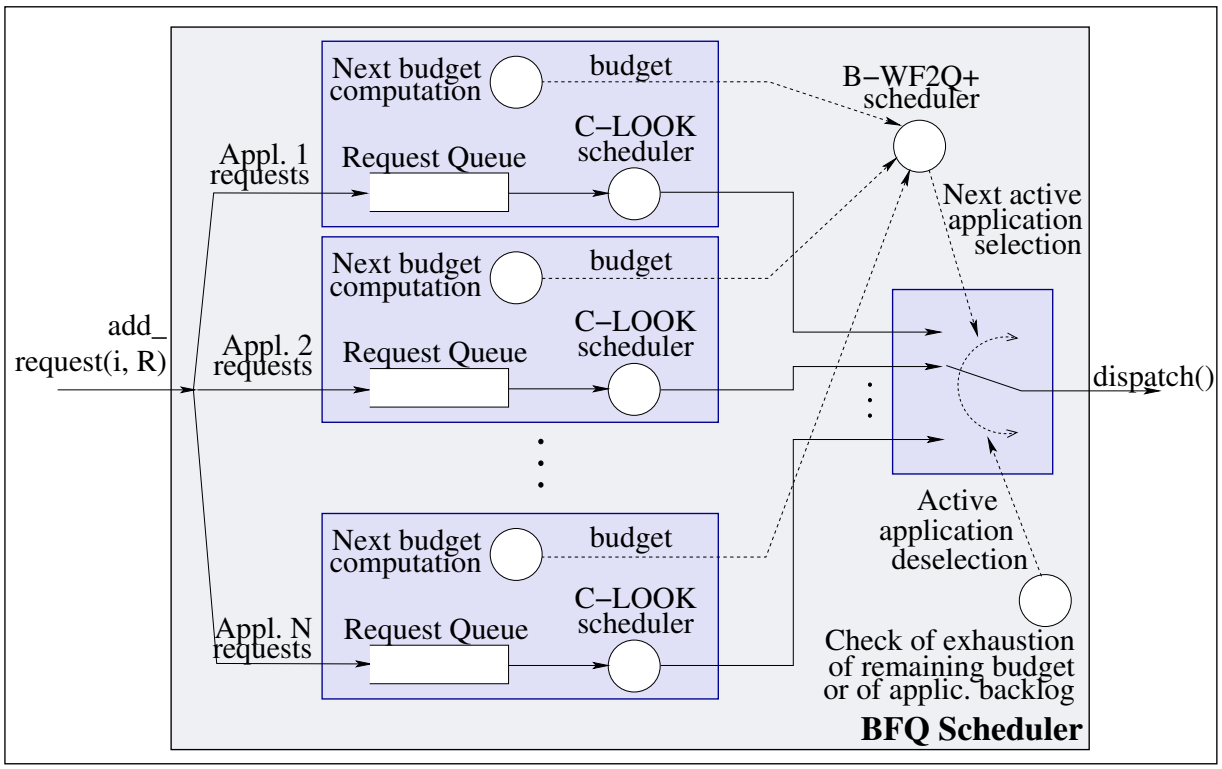

Figure 15: BFQ Logical Scheme.

A set of theorems is provided in [29] to show the temporal properties of BFQ. From those derive the possibility of enforcing the time constraints for the disk requests from an application. The aggregate throughput must be known at some extent to assess the actual time guarantees. Unfortunately, it is in its turn a function of the many user-configurable parameters. The desired trade-off between completion times and the aggregate throughput can be achieved by iteratively tuning the values either of each of these parameters or of just the throughput boosting level parameter. The accuracy of the actual completion times depends on how accurately the aggregate throughput is known (e.g., worst-case or average value, variance, confidence interval).

Once known the worst-case throughput, the requirements of an application requesting a long term throughput and no other type of guarantee can be fulfilled by assigning to it a proportional weight. In contrast, to provide guarantees on single request completion times to an application, the request arrival pattern of the application needs to be modelled too. A general request arrival model is the periodic or sporadic pattern: the application issues requests with a known maximum size, and with a period or minimum inter-arrival time. BFQ properties allow to compute for these requests the maximum worst-case delay that the application must tolerate. This analysis applies under the assumption that the aggregate requests from all application are not overloading the disk. So, an acceptance test for incoming applications or the possibility to dynamically adjust applications requirements must be available to actually enforce the timing constraints.

Larger budgets increase the probability of serving larger bursts of close or even sequential requests, and hence of achieving a higher throughput. Besides, a larger interval after which queued requests must be served in FIFO order may boost the throughput in presence of asynchronous requests. In

this respect, also recall that most applications issues only synchronous requests. Hence, in such applications this parameter has no impact either on the throughput or on the time guarantees. In contrast the idle interval before considering the application idle may be just set to the most effective value for the target disk device, equal to the device-dependent average cost of seek and rotational latencies.

BFQ exports a last low-level configuration parameter related to disk throughput, namely the system-wide maximum *time budget*. Once assigned access to the disk, each active application must consume all of its time budget or backlog within no more than the maximum *time budget*, otherwise it is unconditionally (over)charged for a extra service and the next active application is selected. This additional mechanism prevents applications performing random IO from substantially decreasing the disk throughput. Hence it guarantees practical bandwidths and delays, which are inversely proportional to the aggregate throughput, to applications performing mostly sequential IO. In contrast, applications performing random IO virtually receive no service guarantees.

All these parameters directly or indirectly influence guarantees. Hence, to set the desired trade-off between guarantee granularity and throughput boosting, the values of these parameters must be (iteratively) tuned by (iteratively) measuring the resulting throughput. BFQ also provides a simplified interface, which allows a user not interested in full control over all the parameters to avoid the resulting tuning complexity.

Whatever interface is used, to evaluate both the (worst-case) throughput and the (worst-case) guarantees as a function of the parameters, and to tune the latter, it is necessary to measure the aggregate throughput against some (worst-case) benchmark pattern. Such a pattern may be defined as a function of the information available on the expected request pattern.

7 The JUNIPER framework User Interface

In this section, we briefly present a possible user interface for the JUNIPER framework. The intent of such a specification at an early stage is twofold: on one side it is fundamental to tie up different mechanisms together providing a usable user level abstraction to them; on the other side this will also provide an early assessment on the usability of the proposed mechanisms.

We recall here that our objective is to provide the developers of JUNIPER applications and the upper layers and libraries with the possibility to reserve resource bandwidth on Commercial Off-The-Shelf (COTS) components. In particular, we provide reservation capabilities for processors (in a multi-core environment), a synchronisation protocol for mutually exclusive resources and reservation on disk bandwidth.

As described in Section 4, in the course of the JUNIPER project, we will adopt and implement the BDM model (4.5) for reserving processor bandwidth . Users of the run-time OS support will be thus required to specify a number of parameters related to reservations. For any given (group of) tasks, the user will be able to define a BDM interface $(\Delta, \beta_1, \ldots, \beta_m)$ consisting of:

- the maximum number of virtual processors m;
- the worst-case delay Δ;
- the list of cumulative bandwidth $\beta_1, \ldots, \beta_m$ for each level of parallelism $j = 1, \ldots, m$.

At the low level, such interface will be translated into a set of server reservations with parameters (Q_j, P_j), by using some optimisation technique such as the one described in Section 4.5.

Notice that this interface is very flexible but indeed quite complex, so it may be simplified during the course of the project in exchange for some additional pessimism.

For what concerns shared resources, we will rely on standard Linux PI for intra-server resource access (i.e., resources shared by tasks within the same group), since tasks belonging to the same server are scheduled by standard RT policies. Inter-server resource sharing (i.e., resources shared among tasks of different groups) will instead be managed by the M-BWI protocol (5.2) by default, as is natural for SCHED_DEADLINE entities.

Since Big Data applications are expected to be run on large-SMP NUMA systems, we will allow users to define locality information. In particular, for each server it will be possible to specify:

- the **node**(s) the server is allocated to;
- the server **CPU**(s) affinity (for each node);
- and the **memory policy** (see D1.2).

Regarding disk access, BFQ reservations will be tied to BDM servers. In this way, will be possible to specify two parameters for each server:

- a **priority class**, the I/O scheduling class the server belongs to;
- a **priority**, of the server inside the class;
- and an associated **weight** (6.2), to fine tune servers shares at each priority.

Furthermore, there will be the possibility for users to specify BFQ related parameters for the whole platform. For each request queue, configurable parameters will be:

- a **maximum budget** (6.1), expressed in number of disk sectors;
- and a **timeout** (6.2), to put an upper bound to the latencies imposed by the scheduler.

References

[1] Luca Abeni and Giuseppe Lipari. Implementing resource reservations in linux. In *Real-Time Linux Workshop*, Boston (MA), December 2002.

[2] M. Awasthi, D. Nellans, D. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *PACT*, 2010.

[3] Enrico Bini, Marko Bertogna, and Sanjoy Baruah. Virtual multiprocessor platforms: specification and use. In *Proceedings of the* 30^{th} *IEEE Real-Time Systems Symposium*, pages 437–446, Washinghton, DC, USA, December 2009.

[4] T. Brecht. On the importance of parallel application placement in numa multiprocessors. In *USENIX SEDMS*, 1993.

[5] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of IEEE Real-Time Systems Symposium*, 2009.

[6] Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems*, 48:789–825, 2012.

[7] Xiang Feng and Aloysius K. Mok. A model of hierarchical real-time virtual resources. In *Proc.* 23^{rd} *IEEE Real-Time Systems Symposium*, pages 26–35, December 2002.

[8] N. Fisher, M. Bertogna, and S. Baruah. The design of an EDF-scheduled resource-sharing open environment. In *Proceedings of the 28th IEEE Real-Time System Symposium*, 2007.

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

[10] R. Knauerhase, P. Brett, T. Hohlt, and S. Hahn. Using os observations to improve performance in multicore systems. In *IEEE Micro*, 2008.

[11] R.P. LaRowe, S. Ellis, and M.A. Holliday. Evaluation of numa memory management through modeling and measurements. In *ASPLOS*, 1996.

[12] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global edf in linux. In *Proceedings of the* 7^{th} *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2011)*, Porto, Portugal, 7 2011.

[13] Giuseppe Lipari and Enrico Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2), 2004.

[14] Giuseppe Lipari and Enrico Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *RTSS*, pages 249–258. IEEE Computer Society, 2010.

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

[16] Z. Majo and T.R. Gross. Memory management in numa multicore sustems: Trapped between cache contention and interconnect overhead. In *ISMM*, 2011.

[17] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys*, 2010.

[18] Farhang Nemati, Moris Behnam, and Thomas Nolte. Sharing resources among independently-developed systems on multi-cores. *ACM SIGBED Review*, 8(1), March 2011.

[19] Andrea Parri, Juri Lelli, Mauro Marinoni, and Giuseppe Lipari. An implementation of the bandwidth inheritance protocol in the linux kernel. In *Proceedings of the 15th Real Time Linux Workshop*, RTLWS '13, 2013.

[20] QFQ+. Quick fair queueing plus. `http://algo.ing.unimo.it/people/paolo/agg-sched/`.

[21] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Proc. Conf. on Multimedia Computing and Networking*, January 1998.

[22] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[23] Insik Shih and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proc.* 24^{th} *Real-Time Systems Symposium*, pages 2–13, December 2003.

[24] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering multiprocessors. In *Proceedings of the* 20^{th} *Euromicro Conference on Real-Time Systems*, pages 181–190, Prague, Czech Republic, July 2008.

[25] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, 1989.

[26] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[27] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhancing aperiodic responsiveness in hard-real-teime environments. *IEEE Transactions on Computers*, 44(1), January 1995.

[28] Paolo Valente. Providing near-optimal fair-queueing guarantees at round-robin amortized cost. *Proceedings of the 22nd International Conference on Computer Communication and Networks.*, 2013.

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

[30] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on cc-numa compute servers. In *ASPLOS*, 1996.

[31] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing contention on multicore processors via scheduling. In *ASPLOS*, 2010.