Noname manuscript No.
(will be inserted by the editor)

Workload assignment for global real-time scheduling on unrelated clustered platforms

Antoine Bertout · Joël Goossens · Emmanuel Grolleau · Roy Jamil · Xavier Poczekajlo

the date of receipt and acceptance should be inserted later

Abstract Heterogeneous MPSoCs are being used more and more, from cellphones to critical embedded systems. Most of those systems offer heterogeneous sets of identical cores. In this paper, we propose new results on the global scheduling approach. We extend fundamental global scheduling results on *unrelated processors* to results on *unrelated multicore* platforms, a more realistic model. We introduce several methods to construct the workload assignment of tasks to cores taking advantage of this new model. Every studied result is optimal regarding schedulability, and all the proposed methods but one have a polynomial time complexity. Thanks to the model, the produced schedules have a limited degree of migrations. The benefits of the methods are demonstrated and compared using synthetic tasks sets. Practical limitations of the global scheduling approach on unrelated platforms are discussed, but we argue that it is still worth investigating considering modern MPSoCs.

Keywords real-time scheduling, global scheduling, multiprocessor, heterogeneous platform

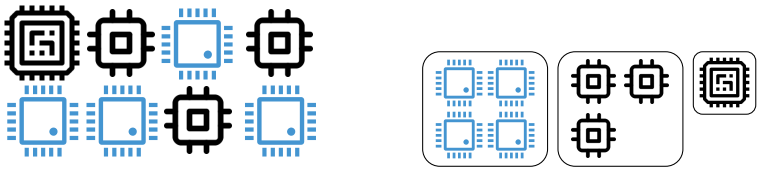
A. Bertout
LIAS, Université de Poitiers, ISAE-ENSMA, Poitiers, France E-mail: antoine.bertout@univ-poitiers.fr

J. Goossens
Université libre de Bruxelles, Brussels, Belgium E-mail: joel.goossens@ulb.be

E. Grolleau
LIAS, ISAE-ENSMA, Université de Poitiers, Chasseneuil Futuroscope, France E-mail: grolleau@ensma.fr

R. Jamil
AC6, LIAS, ISAE-ENSMA, Courbevoie, France E-mail: roy.jamil@ac6.fr

X. Poczekajlo
Université libre de Bruxelles, Brussels, Belgium E-mail: Xavier.Poczekajlo@ulb.ac.be



(a) flat platform model (b) unrelated multicore platforms model

Fig. 1: Illustration of flat versus clustered platform model

1 Introduction

1.1 Motivation

In the keynote speech [15] of the 2019 RTNS edition, Marko Bertogna exposed how heterogeneous architectures found in current and incoming safety-critical innovations such as autonomous vehicles open promising opportunities for the real-time community to bridge practical application with theoretical aspects. *Heterogeneous MultiProcessor System-on-Chip platforms* (MPSoCs) are now widely spread in most embedded systems domains, from infotainment and automated driving system in cars to smartphones and drones. These platforms usually offer several different sets of identical computing cores, called *clusters*, and may also contain specific hardware like specialised processing units (e.g. GPU, NPU) or programmable logic tiles (FPGA). The cluster architectures are typically inspired from multicore architectures and, through a hypervisor, allow a single operating system (*OS*) to globally schedule tasks easily and efficiently. Nevertheless on heterogeneous MPSoCs, different clusters may have different instruction set architectures (*ISAs*), and may host different OSs. For example, the STM32MP157C-DK2® MPSoC from STMicroelectronics is composed of two clusters. On one hand its Cortex-A7® multicore cluster has a Memory Management Unit (*MMU*) allowing memory virtualisation, and can host a multi-purpose Linux OS. On the other hand, its microcontroller Cortex-M4® is a single-core cluster without MMU that can only support lightweight OS (e.g. FreeRTOS or minimal single process RTOS compliant with the POSIX 1003.13 PSE51 [3] profile) or be used directly bare metal. While belonging to the same ARM Cortex® family, these cores have different ISAs. Thus, performing a task migration from one cluster to another requires the task code to be compiled for both types of architecture. Moreover, preemption cannot be allowed between every instruction since the low level instructions are different, and can even be executed out of order on some architectures. On these platforms, migrating a task from a cluster to another would therefore require to determine specific migration points in the code.

This paper is focused on real-time scheduling of a set of tasks. Each task releases a —potentially— infinite set of jobs which have the same worst-case execution-time (WCET) on a *fictional core*. A fictional core is used as a reference to express core processing rates. It is modelled so that all tasks may be executed on it. Jobs must be completed by a given deadline to respect the real-time constraints. In the literature, a platform is often viewed as "flat", as represented in Figure 1(a). In extenso, there is no hierarchy between cores and all migrations are considered

as having the same cost. This is an abstraction since most modern platforms are composed of one or several *clusters* of cores, as represented in Figure 1(b). Cluster cores are identical, but may differ from one cluster to another in the case of *unrelated multicore platforms*. In this paper, the jobs are executed on a computing platform of *unrelated clusters*. Each cluster is characterised by its number of *identical* cores, and each task has a specific processing rate on each cluster. In the literature, multiprocessor systems are generally classified into three categories [23, 12]. (1) *Identical*: all the processors are identical and execute the tasks at the same processing rate; (2) *Uniform*: each processor is characterised by a speed, e.g., a processor of speed 2 executes any task twice faster than a processor of speed 1; (3) *Unrelated*: the processing rate depends on both the processor and the task. There exists a fourth category: *consistent architecture*. This is a particular case of *unrelated* architecture where the heterogeneity is *consistent*. Informally, whenever a processor executes a task *faster* than the others processors then it is also faster at executing the other tasks. While this notion was already considered in the literature, see for instance [7,18], we provide in Section 2.3 a formalisation of this kind of architecture.

The scheduler on a multiprocessor platform can be *global* or *partitioned*. In *global* scheduling, any job may be executed on any core, i.e. migrate without restriction. By contrast, in *partitioned* scheduling each task is assigned to a single core and migration is not allowed. The multicore cluster model allows for an intermediate category: in *clustered* scheduling [12], each task is assigned to a single cluster and jobs can only migrate between cores within the cluster. In this paper, we assume a *global* scheduling on *unrelated* multicore platforms. The migrations between cores of the same cluster are defined as *intra-cluster migrations* while the *inter-cluster migrations* correspond to migrations between cores of different clusters. On most platforms, *inter-cluster migrations* require software support and a specific development effort. This is very costly because of the online execution overhead and time consuming as it requires specific development effort. Today, popular scheduler implementations support symmetrical multiprocessing (SMP) that allows intra-cluster migrations (e.g. the Completely Fair Scheduler (CFS) of the Linux kernel). They are therefore transparent to the application developer, and the online overhead generated is smaller than the *inter-cluster migration* one. Finally, as the scheduler presented in this work is partly based on a template schedule that is repeated over time, it can be classified as an *offline* scheduling algorithm. On the contrary, *online* scheduling defines algorithms whose decisions are taken dynamically during the system lifetime to react to unforeseeable events (e.g. when jobs complete before their worst-case execution time).

1.2 State of the art

Partitioned scheduling on heterogeneous platforms is a NP-hard problem and has been studied in several works [10,38,13]. Global scheduling on heterogeneous platforms, also known as unrelated multiprocessor platforms, was initiated by the seminal paper [9]. Since then, the global scheduling on unrelated platforms has received less attention. This may be due to the fact that hardware platforms generally do not support *inter-cluster* migration of tasks, that may require a full software support. However, global scheduling allows theoretically a full utilisation of the

platform, moreover when neglecting migration cost, the problem of feasibility on unrelated platforms can be addressed in polynomial time [9,30] for independent, implicit deadlines (see Section 2.1 for a formal definition), tasks scheduled globally. In the literature, e.g. in [9,19], the global scheduling upon unrelated platforms is performed in two phases. All the computations are done offline. First, a workload assignment matrix is computed. The workload assignment decides which fraction of processing capacity of a core has to be assigned to each task. Secondly, giving this workload assignment, a template schedule is built. The template schedule is then directly used online.

Recently, MPSoCs with unrelated clusters sharing the same ISA, like the ARM big.LITTLE®architecture, have motivated some work [19] on the optimal global scheduling. Indeed, sharing the same ISA makes the inter-cluster migrations more realistic. In the latter work, the authors adopt a novel strategy, taking into account the hierarchical nature of the set of clusters. They first focus on the assignment of tasks to clusters, and then on cores, which limits the number of inter-cluster migrations. Nevertheless, this method, called Hetero-Split, is limited to a platform with only *two types* of clusters. These two-types platforms also motivated clustered approach with intra-migration like in [36]. New platforms, integrating more than two types of clusters like the Mediatek Helio X20®are developed. This MPSoC includes three clusters (two fast Cortex-A72®cores, four middle speed Cortex-A53®cores and four slow Cortex-A53®cores) sharing the same ISA with a hardware support for inter-cluster migration. This revives interest in the global scheduling of unrelated clusters.

1.3 Contributions and organisation

This paper is an extended version of [17] published in the RTNS 2020 conference. Its content has been enriched on the following points.

- We added experiments on several real heterogeneous MPSoCs showing that the order of magnitude of the time needed for a task to migrate between two cores of the same cluster is smaller than the time required for a task to migrate between two different clusters. These experiments justify the hypothesis that considering a hierarchy of core clusters has many advantages compared to considering a flat platform model, like the seminal work on unrelated multiprocessor platforms do.
- We conducted experiments on real heterogeneous MPSoCs to show that, while the uniform model (processors are characterised by a uniform speed, acting as a global accelerator or decelerator of execution time) does not apply to current technology for several reasons (different instruction sets, impact of cache memory, impact of front side bus speed, etc.), an intermediate platform model between uniform and unrelated is a realistic compromise. This model, that we call *consistent*, simply states that some processors are faster than others, but allows the speed gain to be task dependent rather than global.
- We have included additional experiments to compare the proposed methods among each-other as well as with the state of the art methods.

In this work, we introduce a new model with a two-levels hierarchical platform: a platform has a set of heterogeneous clusters, each cluster is composed by a set of

identical cores. To the best of our knowledge, this hierarchical platform model has only been addressed in the context of optimal global scheduling with two types of clusters. We start with practical considerations and physical experiments on heterogeneous MPSoCs platforms showing that this hierarchical model is closer to the reality than the flat model. We then define the notion of *consistent* platforms, expressing that some clusters are faster than others, but that the speed ratio is not global (unlike the uniform model) but rather per task. We measure the execution time of various tasks on a heterogeneous MPSoC to show that this assumption is realistic on some real platforms. Then, we take advantage of the hierarchical model by proposing several new workload assignment methods derived from former methods applied on flat models. We show that a system is feasible if, and only if, a workload assignment exists. The proof is an elegant mathematical formulation inspired from [30], extending the proof given in [9] for the flat model to both the flat and hierarchical model. These new methods are then tested by simulation on synthetic tasks sets, showing their advantages over the existing methods. These workload assignment methods show a reduced amount of inter-cluster migrations thus improving their applicability. Finally, we discuss the gap between the current theoretical approaches to schedule tasks on heterogeneous platforms and the reality.

Section 2 introduces the task and platform model which is justified in Section 3 by practical experiments on a heterogeneous MPSoC. Section 4 presents the new workload assignment methods. We then evaluate the performances of the new methods in Section 5, and discuss the practicability of global scheduling on heterogeneous platforms in Section 6.

2 Task and platform model

2.1 Task model

The workload is modelled by a set of n *periodic tasks* $\Gamma \doteq \{\tau_i \mid i = 1, \ldots, n\}$ (the symbol $\doteq$ means *is equal by definition to*). Each task τ_i is defined by two parameters (C_i, T_i) where C_i is the *worst-case execution time* on a *fictional processor* capable to execute it —chosen arbitrarily—, and T_i is the release *period*. Each task releases a *job* every period T_i. The first job of a task is released at $t = 0$, the k^{th} at $t = k \times T_i$ and has to complete by $(k+1) \times T_i$ (tasks are said to have implicit deadlines). The utilisation of a task τ_i is $u_i \doteq \frac{C_i}{T_i}$.

2.2 Platform model

An unrelated multicore platform is modelled by a set Π of $\dot{m}$ clusters $\Pi \doteq \{\dot{\pi}_h \mid h = 1, \ldots, \dot{m}\}$. Each cluster $\dot{\pi}_h$ contains $\dot{m}_h$ *identical* cores $\dot{\pi}_h \doteq \{\pi_{h_1}, \ldots, \pi_{h_{\dot{m}_h}}\}$. A job of τ_i that is executed on a core π_{h_k} for t time units will progress by $\dot{r}_{i,h} \times t$ units of its execution time. Within the cluster $\dot{\pi}_h$, every core has the same processing rate $\dot{r}_{i,h}$ for each task τ_i. If $\dot{r}_{i,h} = 0$, then τ_i cannot be executed on the cluster $\dot{\pi}_h$, this couple task/cluster is said to be incompatible. A job of τ_i is completed when its progress reaches its worst-case execution time (WCET) C_i.

τ_i	$\dot{r}_{i,1}$	$\dot{r}_{i,2}$
τ_1	2	1
τ_2	2	1

(a) Uniform clusters

τ_i	$\dot{r}_{i,1}$	$\dot{r}_{i,2}$
τ_1	4	3
τ_2	2	1

(b) Consistent clusters

τ_i	$\dot{r}_{i,1}$	$\dot{r}_{i,2}$
τ_1	4	2
τ_2	1	3

(c) Unrelated clusters

Table 1: Platform classification example

2.3 Consistent clusters

The unrelated model captures the heterogeneity of platforms made of different types of processing units (e.g. CPU, GPU or NPU). In this setting a task τ_1 may be executed faster on a cluster (or processor) $\dot{\pi}_1$ than a task τ_2, that in turn may be executed faster than τ_1 on a cluster $\dot{\pi}_2$. When a task set is exclusively executed on a unique type of processing unit (e.g. CPU clusters with different micro-architecture), there may be a particular setting of the unrelated model that better reflects this kind of platform. In this case, there is still a speed by cluster and task but some clusters are always faster than the others. This particular setting, here named *consistent*, is more general than the uniform multiprocessor model where some clusters are faster than the others, but always proportionally to their speed.

This section provides a formalisation of the notion of *consistent* clusters. First, to be *consistent* the platform must have a *relative order* on the clusters.

Definition 1 (Faster cluster) A cluster $\dot{\pi}_k$ is *faster* than cluster $\dot{\pi}_\ell$ ($\dot{\pi}_k \geq \dot{\pi}_\ell$) if

$$\forall 1 \leq i \leq n \qquad \dot{r}_{i,k} \geq \dot{r}_{i,\ell}$$

Now we introduce a tie-breaker to have the notion of the *fastest* cluster:

Definition 2 (Fastest cluster) $\dot{\pi}_k$ is defined to be *the fastest* processor if k is the smallest index such that $\forall 1 \leq \ell \leq m \quad \dot{\pi}_k \geq \dot{\pi}_\ell$

Without loss of generality (by reordering the clusters) we can assume that $\dot{\pi}_1$ is the fastest cluster. By repeating the same definition on the remaining clusters and without loss of generality we can assume, if the platform is *consistent*, that $\dot{\pi}_1 > \dot{\pi}_2 > \cdots > \dot{\pi}_m$, i.e., we have a *total* order on the clusters.

Scheduling tasks on consistent clusters is a particular case of the unrelated setting but is more general than the uniform setting.

The Table 1 illustrates this classification with three examples of platform. In example 1(a), cluster $\dot{\pi}_1$ is twice faster than cluster $\dot{\pi}_2$ for every task, the clusters are uniform. The clusters are consistent in example 1(b) because $\dot{\pi}_1$ is faster than $\dot{\pi}_2$ but non-uniformly for every task. The cluster speeds are not globally comparable in example 1(c), the clusters are unrelated.

2.4 Assumptions

In this paper, we make the following assumptions. We consider the time as continuous, and that a job may be preempted at any time (fluid schedule). The tasks are sequential so they cannot be executed in parallel. Preemptions and migrations are performed at no extra cost. Also, a task set is *feasible* on a given platform if, and only if, there exists a schedule where every job of every task can be completed by its deadline.

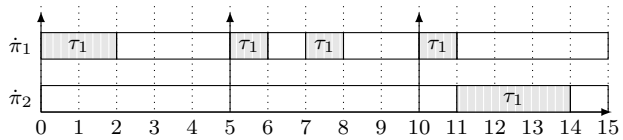


Fig. 2: One task in an offline global schedule

3 Experimental justification of the platform model

In this section, we investigate the migration cost of tasks within and between clusters (intra- and inter-cluster migrations). First, we investigate how inter-cluster migrations may be done in practice (Section 3.1) and estimate the possible cost of such migrations by measuring the communication time needed to share data. Then, in Section 3.2, we compare the results against intra-cluster migrations in a heterogeneous MPSoC to demonstrate the interest of the clustered model. Finally, by measuring the execution time of task upon different CPU clusters (Section 3.3), we show that the *consistent* setting fits well to actual heterogeneous MPSoCs.

Notice that the presented methods are deployed to give a reasonable approximation of the impact of migrations in a heterogeneous platform. It does not aim at providing fined-grained and precise measurement techniques.

3.1 Soft-migration points between heterogeneous clusters

Consider on Figure 2 some offline schedule that is to be executed on any two cores belonging to different clusters. This figure represents the execution of three jobs of a single task τ_1 of period $T_1 = 5$. The complete schedule is repeated every 15 time units and τ_1 is first activated at $t = 0$. The first two jobs are completely executed on cluster $\dot{\pi}_1$. However, the third job starts its execution on cluster $\dot{\pi}_1$, is half executed, and then migrates to finish its execution on $\dot{\pi}_2$. Given the Figure 2, the cores of $\dot{\pi}_2$ are three times slower for executing task τ_1, than the cores of $\dot{\pi}_1$. Indeed, if the task τ_1 is executed *exclusively* on $\dot{\pi}_1$ the cumulative duration before completion is 2 time units. But if the task τ_1 is executed during one time unit on $\dot{\pi}_1$ we see that the remaining computation time on $\dot{\pi}_2$ is three time units, while the remaining computation time on $\dot{\pi}_1$ is one time units. The fourth job will be executed back on $\dot{\pi}_1$, therefore the task has to migrate back at the end of the execution of the job on $\dot{\pi}_2$.

Within a cluster, a multicore OS manages the tasks intra-cluster migrations transparently. Nevertheless, inter-cluster migration is, at least on current technology present in heterogeneous MPSoCs[1], not supported and would require an effort from the programmer.

Somehow, the compiler would have to generate different codes depending on the cores, not only as a matter of instruction sets, but also as a matter of communication and control. This is illustrated on Listing 1, using a pseudo-C++ like code, using a POSIX-inspired API.

[1] Except for heterogenous platform sharing the same instruction set architecture like the ARM big.LITTLE.

```
Task Original_tau_1 {
    Local variables declarations
    Initialization code
    for(;;) {
        wait activation
        First&Second half of the code
    }
}
Task tau_1_on_cluster1 {
    static unsigned count=0;
    static bool first=true;
    Local variables declarations
    Initialization code
    for(;;) {
        wait activation
        if (!first && count==0) {
            // migration from cluster2 to cluster1
            deserialize(wait_AMP_msg(),&local vars);
        }
        first=false;
        First half of the code
        if (count==2) {
            // migration from cluster1 to cluster2
            send_AMP_msg(serialize(local vars));
        } else {
            Second half of the code
            count=(count+1)%3;
        }
    }
}
Task tau_1_on_cluster2 {
    static unsigned count;
    Local variables declarations
    for(;;) {
        // migration from cluster1 to cluster2
        deserialize(wait_AMP_msg(),&local vars);
        Second half of the code
        count=(count+1)%3;
        // migration from cluster2 to cluster1
        send_AMP_msg(serialize(local vars));
    }
}
```

Listing 1: Implementation of soft-migration points

The first task is the original task, that would remain the same if the task was always executed on the same cluster. The second task is the task executed

on cluster $\dot{\pi}_1$, and the third is the task executed on cluster $\dot{\pi}_2$, in order for the schedule to follow pattern of Figure 2. Every three jobs, the task is migrating after half of its computation to $\dot{\pi}_2$, and every time a job is executed on $\dot{\pi}_2$, it will migrate back at the end to $\dot{\pi}_1$. In the Listing 1, the migration is performed as follows: local variables of the tasks are serialized[2] in a byte array, that is sent from a cluster to another, using the available inter-cluster communication protocol.

In order to estimate inter-cluster migration duration on actual platforms, we therefore measured how much time it takes from a cluster to transfer 512 bytes of data (that could correspond to the serialization of a middle-size task's local variables) and the control to another cluster. The experiments are done in both directions, because as it can be seen on the results, they are far from identical.

AMP platforms. We consider two different heterogeneous MPSoCs, the i.MX 8®, and the STM32MP157C-DK2®from STMicroelectronics (STM32MP1 in the sequel). Both platforms are ARM based architectures, composed of a cluster of high performance, fast Cortex-A®cores, and a single Cortex-M®microcontroller.

AMP messaging protocol. Most of the heterogeneous platforms provide an implementation of AMP messaging protocol. Their processors communicate through an inter-processor communication controller providing interrupt signalling and messaging status flags to manage data exchange through a shared memory. The sender core sets a status flag to occupied, writes the message to the shared memory, then generates an interrupt on the receiving core, which reads the message, and then clears the status flag. The protocol's firmware on the microcontroller's side is a library to be included, but on the Linux side we use a dedicated driver to receive or send messages, using standard read/write system calls. In the following experiment, we used RPMsg and RPMsg-Lite [2] frameworks on STM32MP1 and i.MX 8, respectively. Both frameworks are actually the only available implementations of an AMP messaging protocol on the given platforms.

AMP measurement technique. The method employed is proposed in [26]. It is based on a common clock on the board: the sender triggers it when calling the AMP send function, and then polls its value until it is stopped. The second core, as soon as it receives an AMP message, stops the clock. Thus, the first core obtains at its next poll of the clock value the elapsed time, and exports the values. Each measurement represents in an order of magnitude, the communication time in addition to the interference with other Linux background threads in the normal state.

Results. The measurements are displayed in Figure 3 the duration for 512 bytes to pass between core clusters as an AMP message. 512 bytes represents the maximum size of the AMP buffer in the protocol's driver (we observed that the size of the message has a little impact on the measurement time). We made 50,000 measures represented in four box plots. The fast cluster is running Linux, patched with PREEMPT_RT, with a fixed operating frequency, while the microcontroller is used bare metal. Each box plot represents the measures on one platform, either

[2] In our context, serialization handles potential endianness differences, memory alignment depending on underlying architecture and compilation options, or differences in type representation

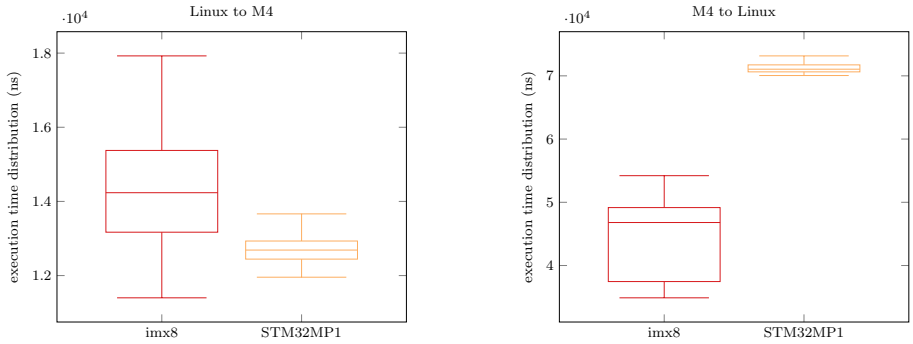


Fig. 3: Inter-cluster communication time for 512 bytes

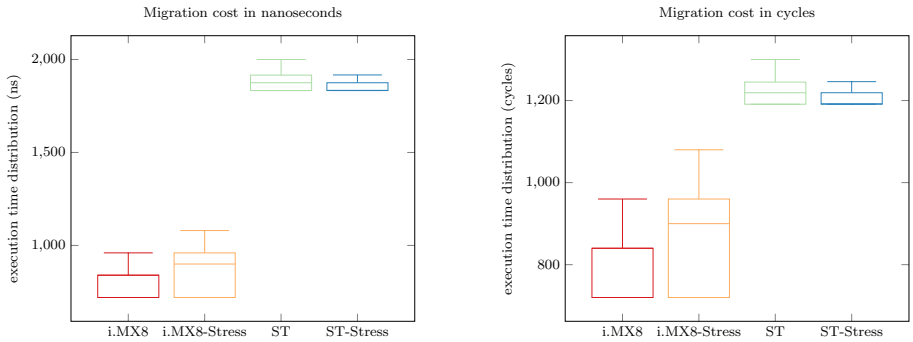


Fig. 4: Intra-cluster migration time

from the fast cluster to the slow cluster (left hand graph), or from the slow cluster to the fast cluster. For example, the leftmost box plot states that in order for 512 bytes to be transferred from the fast cluster to the slow cluster on an i.MX 8, we need between 11.5 and 18 microseconds (μs), with an median of 14.1 μs, a lower quartile of 13.3 and an upper quartile of 15.5 μs. The other platform showed a less variable duration between duration between 12 and 14 μs. This is not to be used to compare both platforms, because it relies on the AMP library implementation. The communication between the microcontroller and the Linux operated cluster is much longer, since on one platform it takes from 30 to 55 μs, and around 70 μs on the other platform.

3.2 Cost of inter- versus intra-cluster migration

To be able to compare the order of magnitude inter- versus intra-cluster migration, we setup some experiments on the same platforms to measure the (intra-cluster) migration time within the Linux cluster.

Linux side configurations. We set a fixed CPU frequency, which avoids any frequency variation due to the processor's load, and we used the highest priority with the sched_fifo scheduling policy.

Measurement technique. The measurement method is described in [26] and based on the CLOCK_MONOTONIC POSIX clock, which is a system wide clock [1]. The execution time by is simply the clock difference between before and after a migration.

Results. We measured 10,000 times the migration time between two cores, while stressing the environment or not. Stressing the environment consists in creating 50 low priority processes communicating through sockets to keep the cores busy. On the left hand side of Figure 4, we can see that on the i.MX 8, the intra-cluster migration time is under 1 μs, while for the STM32MP1, it is always under 2 μs.

This shows that the cost of an inter-cluster migration is, at least, *between 10 and 55 times longer* on i.MX 8, and between 6 and 35 times longer on STM32MP1 than the cost of an intra-cluster migration. This is showing how useful it can be to differentiate both types of migrations on the platform model.

In the sequel, while we showed experimentally that migration costs exist, we will still assume a zero-cost migration of preemption, as this hypothesis is mandatory to find a *polynomial time* feasibility test. Indeed, feasibility is NP-hard in the strong sense for a single-core processor platform as soon as preemption delay is taken into account [34,33]. Nevertheless, we will keep in mind that it is better to favor intra-cluster migration while avoiding if possible inter-cluster migration.

3.3 Variability of the execution time of task in a heterogeneous MPSoC

In this section, we measure the execution time of six different test programs to observe the behaviour of a heterogeneous MPSoC platforms, the STM32MP1. Computing a WCET on such platform is out of the scope of this paper, here we consider the order of magnitude of the measured execution times to compare the different results. We measured 10 000 executions on a *single* core of the Cortex-A®, and 1 000 executions on the Cortex-M®since it shows much less variation in execution time.

In the related figures, the measure distribution is depicted using box plots. We here remove the extreme values (outliers) from the data to improve the general trend visualisation using the following standardised values: any value that was 1.5 lower or larger than the interquartile range —difference between upper and lower quartiles— was discarded. It contains, from bottom to top: the lowest value, the lower quartile, the median quartile, the upper quartile and the highest value. Notice that the percentage of outliers is generally below 0,1% and reaches exceptionally 5% for the stressed programs.

The test programs description follows and their execution times distributions are illustrated in Figure 5.

– BigNum: uses the BigNum library [28], which implements operations on large integers represented as arrays. Depending on its parameters, it may be computationally intensive. Our test BigNum1 executes 5 additions, BigNum2 executes 8 divisions, and BigNum3 is more computationally intensive as it executes 12 additions and multiplications. When the Linux system is stressed, we observe major variations of the execution times. For example, the Tukey box for BigNum1 ranges from around 37.2 microseconds (μs) to 37.4 μs without stress,

when it can be measured up to 38.6 μs when stressed. The behaviour shows much less variation on the bare metal Cortex-M®core, but is close to 4 times slower than on the Cortex-A®core. The same ratio —approximately 4— can be observed for the three variants of BigNum.
- n-body: computes the movement of planets using a symplectic integrator [29]. It starts with a short initialisation phase and then does intensive floating point operations to determine the planet movements. For this program, we observe a ratio greater than 10 between the execution times on the A core compared to the execution on the M core. This is probably due to the presence of a Co-Processor ARM Neon®on the fast cores. This ratio is getting close to 15 for the 50 iterations of this program, which contains more floating point computationally intensive operations.
- FFT: a Fast Fourier Transform, computationally intensive with floating point operations. Similarly to Nbody-50 moves, this task is executed around 15 times faster on a Cortex-A®than on a Cortex-M®.

These samples illustrate the fact that the acceleration factor of a processor is not global on heterogeneous MPSoCs platforms, but rather based on the task usage of specific hardware parts of the processor (like here, a FPU). Usually, a faster processor tends to be better equipped in supplementary hardware co-processors than a slower processor, enforcing the fact that most actual heterogeneous MPSoCs can be called consistent on most types of tasks.

4 Workload assignment methods

In this section, we focus on different methods to assign the workload of tasks to a platform. As far as we know, every optimal scheduling method of the literature [38] for unrelated multiprocessor platforms (from real-time [9] or operational research [30] areas), starts with a workload assignment phase (made offline). From an input made of tasks parameters and platform rates, this phase decides the fraction of processing capacity of each core assigned to tasks. The tasks have to be completed within their period thanks to this assignment, without overloading the cores. With the exception of [19], presented in this section to serve as a comparison for the experiments of Section 5, most of the existing works have expressed the workload assignment phase as a LP problem. A Linear Programming (LP) problem can be solved in polynomial time [27].

The solution of the LP problem is a *cluster workload assignment* matrix $X = [x_{i,h}]_{i=1,\dots,n}^{h=1,\dots,\dot{m}}$ where $x_{i,h}$ is the fraction of a core in the cluster $\dot{\pi}_h$ used by a task τ_i.

As shown experimentally in Section 3, inter-cluster migrations are more costly in terms of time overhead and task programming effort than intra-cluster migrations. We quantify the impact of such migrations by the definition of the *presence* of a task on a cluster, introduced in [9]. Formally, a task τ_i has a *presence* on a cluster $\dot{\pi}_h$ iff $x_{i,h} > 0$. The *number of presences* $\dot{\Pr}_i$ corresponds to the number of times where τ_i has a presence in a cluster: $\dot{\Pr}_i \doteq |\{x_{i,h} > 0 \mid h = 1, \dots, \dot{m}\}|$

A task τ_i will have to migrate between clusters if, and only if, $\dot{\Pr}_i > 1$. Therefore, any presence greater than one is a *presence in excess* that will generate at least one inter-cluster migration at runtime.

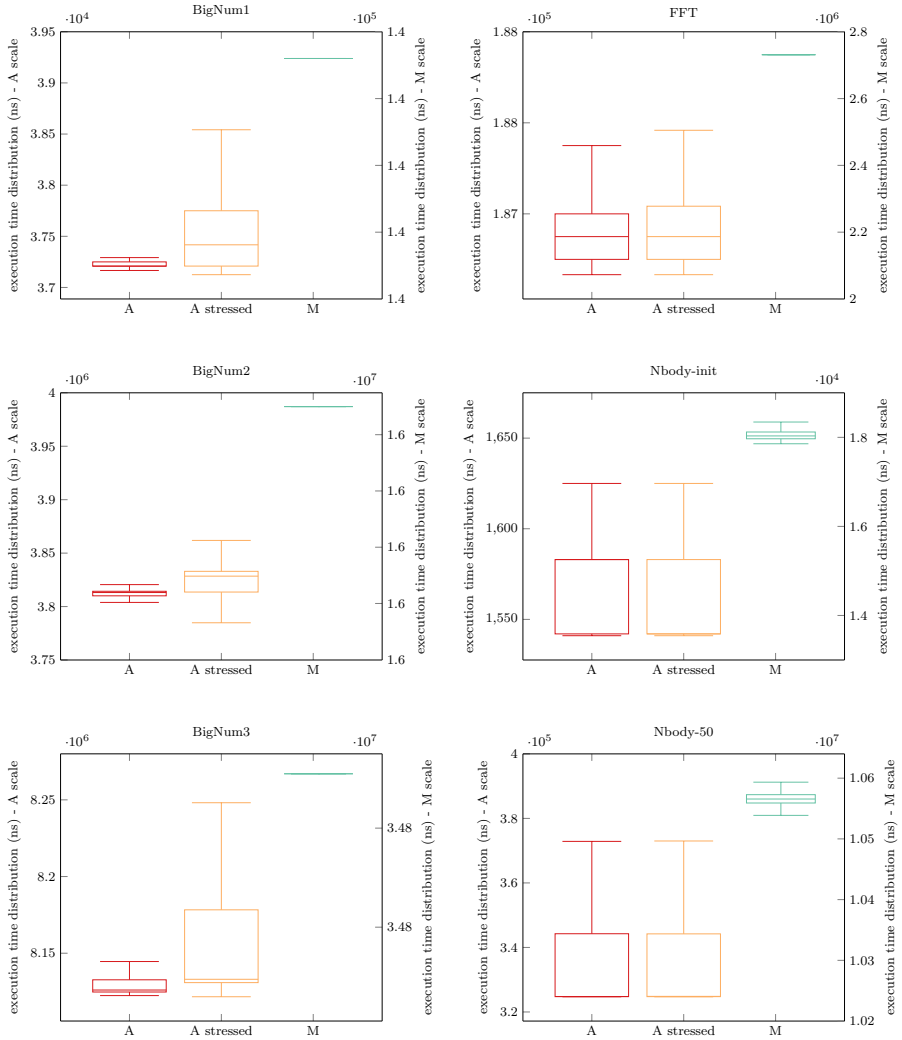


Fig. 5: Execution time distribution of different programs

In this section, we first formulate the cluster workload assignment as a LP problem and show that it is an exact feasibility test. This formulation extends the seminal LP problem of [9]. Then, we present a Mixed-Integer Linear Programming (*MILP*) formulation minimising the number of presences of tasks on clusters. Finally, we present succinctly a method from the literature limited to two types of clusters that will be experimentally compared to the other LP-based solutions.

4.1 Workload assignment as a LP problem

Assigning the workload of tasks on clusters can be expressed using three sets of constraints, defined in LP-Cluster:

LP 1 (LP-Cluster)**.**

$$\sum_{h=1}^{\dot{m}} x_{i,h} \times \dot{r}_{i,h} = u_i \qquad i = 1, 2, \ldots, n \tag{1}$$

$$\sum_{h=1}^{\dot{m}} x_{i,h} \leq 1 \qquad i = 1, 2, \ldots, n \tag{2}$$

$$\sum_{i=1}^{n} x_{i,h} \leq \dot{m}_h \qquad h = 1, 2, \ldots, \dot{m} \tag{3}$$

objective constraint: any

Equation 1 ensures that enough processing capacity is allocated to each task by reserving a processing capacity fraction on each cluster. Equation 2 constrains the total capacity fraction allocated to a task to be less than or equal to one. This ensures that the task can be scheduled without being executed on two cores at the same time (see Theorem 1). Equation 3 states that the used capacity of a cluster $\dot{\pi}_h$ is less than or equal to its total capacity, which is the capacity of its $\dot{m}_h$ cores.

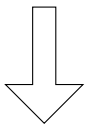
If the *LP-Cluster* is successfully solved, the $x_{i,h}$ represent a successful cluster workload assignment (or assignment of tasks on clusters).

Theorem 1 *A system is feasible on the platform if, and only if, LP-Cluster has a solution.*

Proof. First we prove that (i) if there is no solution to the LP, then the system is not feasible. This will occur if Equation 2 or Equation 3 are not satisfied. In the first case, there would be at least one task τ_i such that $\sum_{h=1}^{\dot{m}} x_{i,h} > 1$. It means that τ_i must be executed in parallel which is forbidden in our model of sequential tasks. In the second case, a cluster $\dot{\pi}_h$ would need a processing capacity higher than its total capacity $\dot{m}_h$.

Now we prove that (ii) finding a solution to this LP problem guarantees that the system is feasible. The proof sketch is depicted in Figure 6. The cluster workload assignment matrix X is of dimension $n \times \dot{m}$. Indeed, by construction of the LP problem, it has n rows with a sum of coefficients less than one, and $\dot{m}$ columns with a sum of coefficients less than $\dot{m}_h$, for each column $h = 1, \ldots, \dot{m}$. First, we replace each column h, corresponding to the task assignment to cluster $\dot{\pi}_h$ by $\dot{m}_h$ columns, one for each core, such that the sum of the coefficients on each of the columns is not greater than one. For the sake of the proof, we simply consider, on each row i of the new columns $k = 1, \ldots, \dot{m}_h$, $x'_{i,h_k} \doteq \frac{x_{i,h}}{\dot{m}_h}$, such that the total capacity fraction allocated to each task on each cluster is evenly distributed on each of its cores. In this manner, we obtain a workload assignment matrix on the cores X_c of dimension $n \times M$, where $M \doteq \sum_{h=1}^{\dot{m}} \dot{m}_h$ is the total number of cores. On each column, $x'_{i,h_k} \doteq \frac{x_{i,h}}{\dot{m}_h}$ represents the capacity fraction of core π_{h_k} allocated to task τ_i. By construction, since originally the used capacity of cluster $\dot{\pi}_h$ to tasks was $\sum_{i=1}^{n} x_{i,h} \leq \dot{m}_h$, we have on each column for π_{j_k}, $\sum_{i=1}^{n} x'_{i,h_k} \leq 1$ (see X_c on Figure 6). From this cluster cores workload assignment matrix, we can easily create a bistochastic matrix B of size $(n + M) \times (n + M)$, as done in [30]. A *bistochastic* (or doubly stochastic) matrix is a square matrix of non-negative real numbers, having each of its rows and columns summing to 1.

$$
X = \begin{array}{c} \\ \tau_1 \\ \vdots \\ \vdots \\ \tau_n \\ \\ \end{array}
\begin{array}{c}
\begin{array}{ccc} \dot{\pi}_1 & \cdots & \dot{\pi}_{\dot{m}} \end{array} \\
\left(\begin{array}{ccc} x_{1,1} & \cdots & x_{1,\dot{m}} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ x_{n,1} & \cdots & x_{n,\dot{m}} \end{array}\right)
\begin{array}{c} \leq 1 \\ \vdots \\ \vdots \\ \leq 1 \end{array} \\
\begin{array}{ccc} \leq \dot{m}_1 & \cdots & \leq \dot{m}_{\dot{m}} \end{array}
\end{array}
$$



Cluster to cores extension

$$
X_c = \begin{array}{c} \\ \tau_1 \\ \vdots \\ \vdots \\ \tau_n \\ \\ \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} \pi_{1_1} & \ldots & \pi_{1_{\dot{m}_1}} & \pi_{2_1} & \ldots & \pi_{2_{\dot{m}_2}} & \cdots & \cdots & \pi_{\dot{m}_{\dot{m}_{\dot{m}}}} \end{array} \\
\left(\begin{array}{cccccccccc} x_{1,1}/\dot{m}_1 & \ldots & x_{1,1}/\dot{m}_1 & x_{1,2}/\dot{m}_2 & \ldots & x_{1,2}/\dot{m}_2 & \cdots & \cdots & x_{1,\dot{m}}/\dot{m}_{\dot{m}} \\ \vdots & & \vdots & \vdots & & \vdots & & & \vdots \\ \vdots & & \vdots & \vdots & & \vdots & & & \vdots \\ x_{n,1}/\dot{m}_1 & \ldots & x_{n,1}/\dot{m}_1 & x_{n,2}/\dot{m}_2 & \ldots & x_{n,2}/\dot{m}_2 & \cdots & \cdots & x_{n,\dot{m}}/\dot{m}_{\dot{m}} \end{array}\right)
\begin{array}{c} \leq 1 \\ \vdots \\ \vdots \\ \leq 1 \end{array} \\
\begin{array}{cccccccccc} \leq 1 & \ldots & \cdots & \cdots & \ldots & \cdots & \cdots & \cdots & \leq 1 \end{array}
\end{array}
$$

Fig. 6: Proof sketch for Theorem 1

Formally, $\forall i = 1, \ldots, n : \sum_{h=1}^{n+M} B_{i,h} = 1$ and $\forall h = 1, \ldots, M : \sum_{i=1}^{n+M} B_{i,h} = 1$. B is constructed as follows:

$$
B \doteq \left(\begin{array}{c|c} X_c & B_n \\ \hline B_M & X_c^t \end{array}\right)
$$

B_n is a $n \times n$ diagonal matrix, such that $B_n(i,i) \doteq 1 - \sum_{h=1}^{\dot{m}} \sum_{k=1}^{\dot{m}_h} x'_{i,h_k}$ $\forall i$. The diagonal coefficients of B_n correspond to the *laxity* of the task τ_i, i.e. the fraction of time during which τ_i is left idle. B_M is a $M \times M$ diagonal matrix, such that $B_M(h_k, h_k) \doteq 1 - \sum_{i=1}^{n} x'_{i,h_k} \forall h_k$. The diagonal coefficients of B_M correspond to the *slack* of the core π_{h_k}, i.e. the fraction of time during which π_{h_k} is left idle. X_c^t is the transpose of the core workload assignment matrix X_c, and has a dimension $M \times n$. By construction, we obtain a square bistochastic matrix B of dimension $(n+M) \times (n+M)$ expressing the fraction of each core that has to be allocated to each task, as well as the slack of the cores and the laxity of the tasks. Following the Birkhoff-von Neumann (BvN) theorem, such a matrix can be decomposed into a convex combination of permutation matrices $A \doteq \delta_1 P_1 + \delta_2 P_2 + \cdots + \delta_k P_k$ [30],

where δ_i is a real coefficient $\in (0,1]$, $\sum_{i=1}^{k} \delta_i = 1$, and P_i is a permutation matrix. A *permutation matrix* is a binary square matrix where there is exactly one 1 on each row and each column. This can be seen as a matching between tasks (rows) and cores (columns). Indeed, one and only one coefficient $P_i(h,k) = 1$ means that task on column k will be assigned to the core of the row h for a duration δ_i. The assignment matrix X_c states that assigning a ratio of x'_{i,h_k} of core π_{h_k} to task τ_i during each of its periods ensures that its jobs will be completed. However, we need to ensure that a job is never executed on two different cores at the same time.

For each time window $[t_1, t_k)$, between two successive releases at times t_1 and t_k (or deadlines since tasks have implicit deadlines), we can use the BvN decomposition to create such a schedule. We use the matching P_1 on the time window $[t_1, \delta_1 \times (t_k - t_1))$, by definition of a permutation matrix, this matching ensures that a task is assigned to at most one core in this time windows. Similarly, we can use the following permutation matrices obtained in the BvN decomposition, each permutation matrix P_i covering a sub-interval of duration $\delta_i \times (t_k - t_1)$. Since by the BvN theorem, $\sum_{i=1}^{k} \delta_i = 1$, we can completely schedule every task on the interval $[t_1, t_k)$, ensuring that a task is never executed on more than one core at the same time. This one time unit schedule can then be *stretched* to fit into intervals of time delimited by successive task release dates. This technique is also referred to as deadline partitioning [32]. □

4.2 About linear algebra for scheduling purposes

Theorem 1 shows that finding a solution to *LP-Cluster* asserts the feasibility of the system. Moreover, it shows that building a schedule from a workload assignment matrix is exactly equivalent to finding a BvN decomposition of this matrix. This result indicates that linear algebra results could be used to improve the schedule construction.

One may note that minimising the number of permutation matrices in a BvN decomposition is similar to minimising the number of scheduling decisions. Indeed, each different permutation matrix corresponds to a different schedule decision (i.e. which jobs are executed at a given instant, and on which cores). Taking schedule decisions lead to preemptions and/or migrations (both inter- or intra-cluster). Therefore, minimising the number of scheduling points may be a solution to reduce the number of preemption and migrations. This is an example of optimisation of the template schedule construction [9,19,16].

The next property concerns the complexity of the BvN decomposition:

Theorem 2 (Dufossé 2016 [24]) *The problem of deciding if there is a BvN decomposition of a given doubly stochastic matrix with k permutation matrices is NP-complete in the strong sense.*

Since the decision problem is NP-complete in the strong sense, the optimisation problem of minimising the number of permutation matrices in a BvN decomposition is NP-hard in the strong sense. Thus, optimising the number of scheduling decisions cannot be done efficiently.

In the remainder, we focus only on modifying the workload assignment to reduce the number of preemptions and migrations. However, using linear algebra

techniques to *sub-optimally* reduce the number of scheduling decisions will be explored in future works.

4.3 LP-Feas and LP-CFeas

In [9], author presents a LP-Feas, a LP model for assigning the workload on an unrelated real-time multiprocessor platform. This work was primarily focused on feasibility, and does not aim at minimising the number of presences. It is very close to the LP formulation of the makespan minimisation in job shop scheduling on unrelated single-core processors given in [30]. In that work, the model is using a *flat* platform model. To fit our model notations, we consider a hierarchical hardware with one core per cluster, i.e. $\forall h, \dot{m}_h = 1$. The different LPs are named LP-α (where α is the name of the LP) for LP addressing the flat platform model (i.e., models not considering clustered), while LP-Cα is used to qualify a LP variant that considers the hierarchical model.

LP 2 (LP-Feas [9])**.** *The workload assignment is solution of the following LP:*

$$\sum_{h=1}^{\dot{m}} x_{i,h} \times \dot{r}_{i,h} = u_i \qquad i = 1, 2, \ldots, n \tag{4}$$

$$\sum_{h=1}^{\dot{m}} x_{i,h} \leq \ell \qquad i = 1, 2, \ldots, n \tag{5}$$

$$\sum_{i=1}^{n} x_{i,h} \leq \ell \qquad h = 1, 2, \ldots, \dot{m} \tag{6}$$

Minimise makespan objective: Minimise ℓ, the system is feasible if, and only if, $\ell \leq 1$.

The immediate extension of LP-Feas to clusters is the following:

LP 3 (LP-CFeas)**.** *The workload assignment is solution of the following LP:*

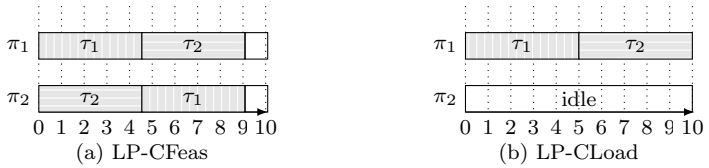
$$\sum_{h=1}^{\dot{m}} x_{i,h} \times \dot{r}_{i,h} = u_i \qquad i = 1, 2, \ldots, n \tag{7}$$

$$\sum_{h=1}^{\dot{m}} x_{i,h} \leq \ell \qquad i = 1, 2, \ldots, n \tag{8}$$

$$\sum_{i=1}^{n} x_{i,h} \leq \dot{m}_h \times \ell \qquad h = 1, 2, \ldots, \dot{m} \tag{9}$$

Minimise makespan objective: Minimise ℓ, the system is feasible if, and only if, $\ell \leq 1$.

LP-Feas and LP-CFeas differ in Equations 6 and 9: since on a unrelated multicore platform, a cluster $\dot{\pi}_h$ has $\dot{m}_h$ cores, a total capacity of $\dot{m}_h$ can be allocated to tasks. It is straightforward that the condition $\ell \leq 1$ constrains solutions of LP-CFeas to be solutions of LP-Cluster. Therefore by Theroem 1, a solution of LP-CFeas with $\ell \leq 1$ can be used to build a feasible schedule.



(a) LP-CFeas (b) LP-CLoad

Fig. 7: Rectangle schedule computed from LP-CFeas versus schedule favouring fast cores utilisation computed from LP-Cload

4.4 LP-Load and LP-CLoad

LP-Feas and LP-CFeas tend to reduce the makespan of the schedule that will be stretched between successive releases. As an example, let two tasks be scheduled on two very different cores: one being ten times faster than the other one for all the tasks. Consider the system of two tasks $\Gamma = \{\tau_1, \tau_2\}$, with both WCET given by $C_1 = C_2 = 5$ and both periods given by $T_1 = T_2 = 10$. The platform is composed of two clusters of one core each, with $\Pi = \{\dot{\pi}_1, \dot{\pi}_2\}$, both clusters having only one core $\dot{m}_1 = \dot{m}_2 = 1$, and having respective rates $\dot{r}_{1,1} = \dot{r}_{2,1} = 10$ for $\dot{\pi}_1$, and $\dot{r}_{1,2} = \dot{r}_{2,2} = 1$ for $\dot{\pi}_2$. The workload assignment matrix computed by LP-CFeas (or equivalently LP-Feas since clusters have one core) is given by $X_{\text{LP-CFeas}} = \begin{bmatrix} 5/11 & 5/11 \\ 5/11 & 5/11 \end{bmatrix}$ ($5/11 \approx 0.4545$). This would lead to a schedule repeated between every successive release (which is every ten time units in our simple example since both tasks have a period of 10), as shown in Figure 7(a).

When considering the number of presences of tasks on clusters, a more interesting workload assignment would favour a high utilisation, or load, on faster cores: $X_{\text{LP-CLoad}} = \begin{bmatrix} 1/2 & 1/2 \\ 0 & 0 \end{bmatrix}$. Such workload assignment could lead to a schedule such as Figure 7(b), which does not produce any inter-cluster migration. LP-CLoad is a LP formulation with the same constraints as LP-Cluster, with the objective of minimising the used capacity of the system. On the unrelated multicore platforms problem, it is defined for LP-Cluster as: ***LP-CLoad:*** Minimise $\sum_{i=1}^{n} \sum_{h=1}^{m} x_{i,h}$. LP-CLoad can be used in the context of a flat platform model. To do so, one simply has to assume that each core is a cluster of size one, i.e. $\dot{m}_h = 1$ for every cluster $\dot{\pi}_h$.

4.5 Minimal number of presences: ILP-CMig

Even if non polynomial, an optimal method minimising the number of presences of tasks on clusters can be useful. Indeed, a system designer may prefer spending a couple of hours waiting for the assignment to be computed rather than spending development time and facing the complexity to implement an inter-cluster migration. Since we are working at the cluster level, the size of the problem, at least in the number of clusters, can be as small as two or three in practice. We propose a Mixed Integer Linear Programming (*MILP*) formulation called *ILP-CMig*, based on the LP-Cluster. In addition, we introduce a boolean variable $b_{i,h}$. Variable $b_{i,h}$ is 1 if task τ_i is present on cluster $\dot{\pi}_h$, and 0 otherwise. The objective is to minimise the total number of presences.

LP 4 (ILP-CMig)**.** *The workload assignment is solution of LP-Cluster (Equations 1, 2, 3) with the following additional constraints:*

$$b_{i,h} \in \{0,1\} \qquad i = 1,\dots,n; h = 1,\dots,\dot{m} \tag{10}$$

$$x_{i,h} \leq b_{i,h} \qquad i = 1,\dots,n; h = 1,\dots,\dot{m} \tag{11}$$

$$b_{i,h} < 1 + x_{i,h} \qquad i = 1,\dots,n; h = 1,\dots,\dot{m} \tag{12}$$

Minimise presence objective: Minimise $\sum_i^n \sum_h^{\dot{m}} b_{i,h}$

The non-clustered version ILP-Mig has the same set of constraints than LP-Cluster where each core is considered as a cluster with a single core. Of course, the complexity of ILP-Mig will grow rapidly with the number of cores, which is higher than the number of clusters of the platform, and will be more and more in the future. While ILP-CMig will minimize the number of inter-cluster migrations, ILP-Mig will minimize the total number of migrations (intra- and inter- without distinction).

4.6 Hetero-Split

In order to compare the previous methods to an efficient existing one, we consider Hetero-Split [19]. This algorithm solves an equivalent expression of LP-Cluster with a $\mathcal{O}(n \log n)$ time complexity. It is restricted to systems having only two different types of clusters. A property of this algorithm is to limit the number of tasks having a number of presences in excess higher than one (i.e., assigned to the two different clusters) to the total number of cores. Since there are only two clusters, tasks are classified into two categories: either cluster $\dot{\pi}_1$ is more efficient, or it is $\dot{\pi}_2$ that is more efficient for their execution. The method exploits this dual property and thus cannot be easily extended to more than two types of clusters. Nevertheless, it allows the use of McNaughton wrap-around rule to efficiently create a schedule conforming to the workload assignment.

5 Experimental comparison of workload assignment methods

When neglecting the migration cost, every workload assignment method presented in Section 4 is optimal regarding the feasibility. Since we know that this hypothesis is unrealistic, we compare the number of presences in excess $\dot{\Pr}_i - 1$ for the six presented methods. The number of presences in excess is a lower bound on the number of inter-cluster migrations. The LP based methods, as well as Hetero-Split, are polynomial time methods, while the ILP-CMig method has an exponential time complexity regarding the number of clusters and the number of tasks. In this section, we compare the following methods:

- LP-Feas is the method minimising the makespan proposed in [9] considering the "flat" core model, while its clustered version LP-CFeas presented in Section 4.3 considers the hierarchical clustered model;
- LP-Load (see Section 4.4), whose objective is to minimise the total core utilisation, its clustered version is LP-CLoad;
- Hetero-Split ([19]) a linear algorithm limited to two types of clusters;

– ILP-Mig is the "flat" core-based version of ILP-CMig, a MILP problem minimising the number of presences on clusters. In practice, ILP-CMig uses significantly fewer variables than ILP-Mig.

In Section 2.3 we formalised the notion of *consistent* clusters.

Thus, the methods cited above are also compared using systems generated with consistent clusters.

In this experiments, we focus on the average number number of presence in excess but also in the percentage of systems with null inter-cluster presence in excess corresponding to partitioned schedules.

5.1 Experimental setup

For the number of presences and simulation experiments, we have generated the systems as follows. The number of types of clusters $\dot{m}$ is either 2 or 5. The former in order to compare Hetero-Split to the other methods, and the latter because five different types of clusters is considered a large size for a heterogeneous MPSoC nowadays. Then, the number of cores per type of cluster is set in $[2, 5]$. The number of tasks n is arbitrarily bounded as follows: $\dot{m} \leq n \leq 10 \times \dot{m}$. We then generate every task such that its period T_i is determined using [25]. The parameter C_i is based on T_i: $\frac{T_i}{2} \leq C_i \leq T_i$. We then generate the rates randomly and adjust them so that the tasks fit the given utilisation. For experimentation purposes, the clusters (the rates in particular) may be set to consistent.

Using this generator, we generate 1 000 systems per total utilisation range $u \in [p - 0.1, p)$, increasing p from 0.4 to 1, for both $\dot{m} = 2$ and $\dot{m} = 5$. The ratio $p = 1$ corresponds to a full utilisation of the platform by the tasks. Here, p is equal to the value of the LP-CFeas objective function result, which is the maximal platform utilisation. The experimentation compares the different scheduling methods over 28 000 randomly generated test systems. As ILP-Mig and ILP-CMig have an exponential time complexity, they are tested using only a subset of the generated systems.

5.2 Inter-cluster number of presences in excess

The workload assignment methods are compared in terms of inter-cluster presences in Figure 8. First note that the scale is 10^{-2}, meaning that in average, very few tasks are assigned to different clusters, for both two and five types of clusters.

On the graphs a) and b) (with $\dot{m} = 2$), we observe that the consistent feature of the systems has no noticeable impact on the results. Hetero-Split performs close to LP-CFeas for low platform utilisation. At higher platform utilisation, Hetero-Split dominates the other polynomial time assignment methods. We can see that both the *Feas*-based LP solutions perform poorly at low platform utilisation compared to the *Load*-based LP solutions for both two-types and five-types (graphs c) and d)) of cores. This is due to *Feas* objective that tends to create "rectangular" (i.e. all processors tend to be idle at the same instant) schedules by balancing the tasks workload on different cores or clusters, as illustrated in Figure 7. While the platform utilisation increases, the slack left at the right-hand side of this rectangle reduces, and the solutions provided by both objective functions tend to be similar.

At high platform utilisation, we thus see that both clustered versions of the LP outperform both non-clustered versions. When combining the two approaches —both the clustered version and the Load objective function—, we observe two to four times fewer inter-cluster migrations compared to the seminal non-clustered Feas objective function. On the consistent cores (with $\dot{m} = 5$), graph d) shows that LP-Load even dominates LP-CFeas for low platform utilisation but its performances seriously worsen for higher platform utilisation, when LP-CFeas meets LP-CLoad. On graphs e) and f) (g) and h)), we see the proportion of generated systems for which the assignment is completely clustered for two (five) types of clusters. It is close to 100% for the ILP-CMig, while the clustered LP-CLoad dominates all the other methods in terms of ratio of completely clustered workload assignments. For similar utilisation, the greater the number of clusters, the more the tasks are likely to be distributed and the less the systems are partitioned. Again, the consistent feature has generally no impact on the results, except for Hetero-Split. Indeed, comparing graphs e) and f), we observe that LP-CLoad performs better than Hetero-Split regarding systems partitioning on consistent two-types systems. However, Hetero-Split performs better in average on unrelated two-types systems, especially at very high utilisation ([0.9,1]). This behaviour is due to the properties of Hetero-Split. In short, Hetero-Split begins the workload assignment by assigning the maximal fraction of tasks to their most capacity-efficient cluster. Then, the rest of workload is distributed in order to respect the constraints of LP-Cluster. In this manner, the first phase limits the presence of the tasks on both clusters. However, with consistent clusters, all the tasks are more capacity-efficient on the same fastest cluster. Thus, this globally reduces the number of tasks being entirely assigned to only one cluster and increases the number of presences in excess.

5.3 Runtime measurement

The performance of the LP/ILP based solution in terms of execution time are depicted in Table 2. The experiment has been conducted on a Intel I7500® multi-core processor from a prototype written in the Python programming language. The Parma Polyhedra Library [8] was used to solve the linear programs with rational coefficients and CBC [37] to solve the mixed integer linear programs. The left table gives the performances of the LP/ILP based solution with the same test systems. In this experiment, the system utilisations are uniformly distributed in the range [0.3, 1.0]. The rest of the system parameters are generated as in Section 5.1. The table on the right gives the average performances with test systems ordered by number of tasks. Thus, both tables are not comparable because they do not have the same test systems. The left table gives the average computation time, per LP or ILP for both $\dot{m} = 2$ and $\dot{m} = 5$ on unrelated clusters. For example, ILP-Mig took an average of 0.061 second to compute the workload assignment with $\dot{m} = 2$. We observe that the clustered version of a LP or an ILP is always faster than the non-clustered version, which is normal since there are fewer variables in the clustered versions. Also, the execution time from $\dot{m} = 2$ to $\dot{m} = 5$ increases drastically and this affects less the clustered versions, since there are fewer additional cluster variables than core variables. Hetero-Split is limited to two types (NA for $\dot{m} = 5$) while ILP-Mig is not practicable in a reasonable time for $\dot{m} = 5$ (NA) with the system parameters given in Section 5.1. As ILP-Mig is significantly slower for

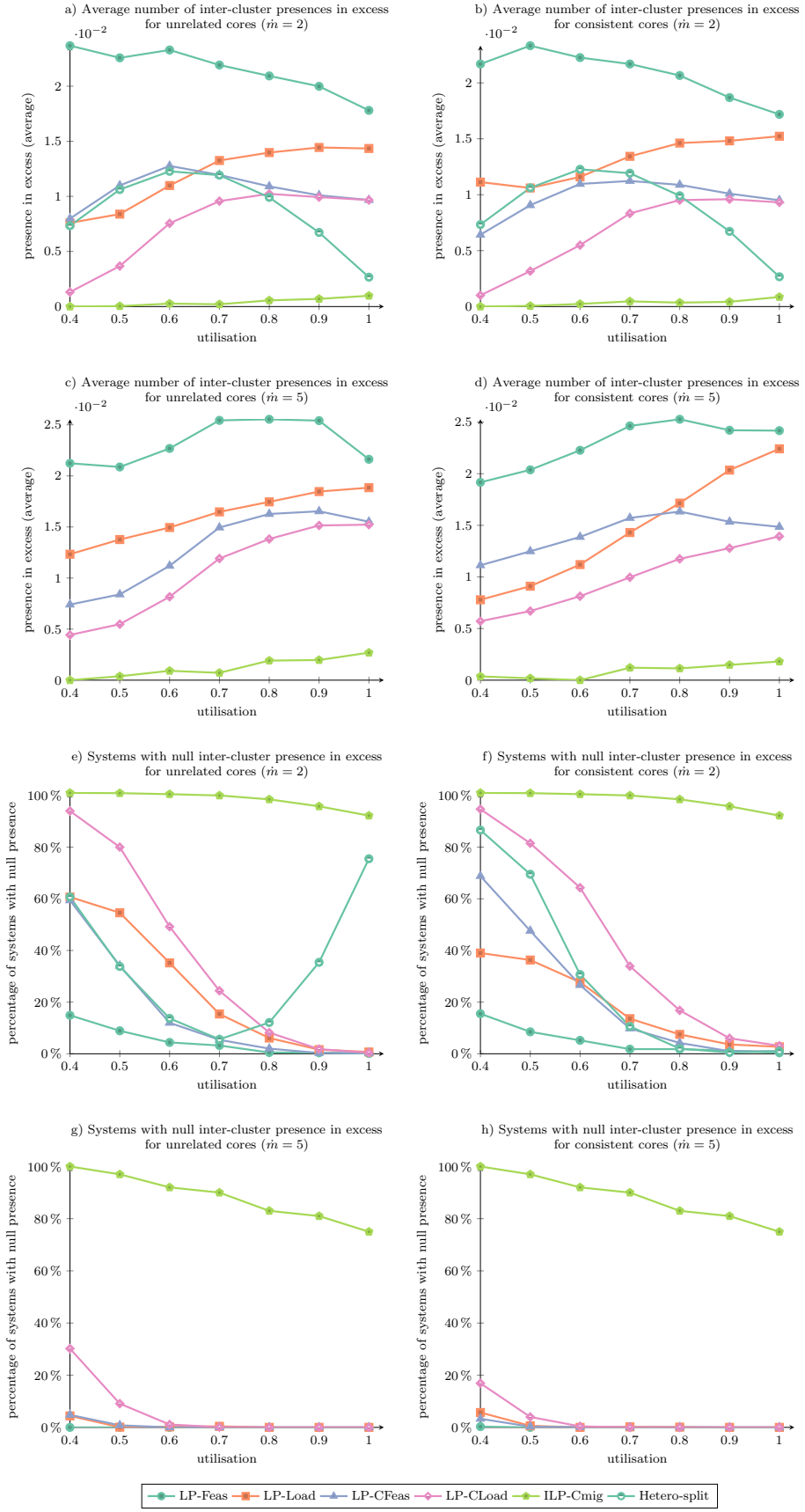
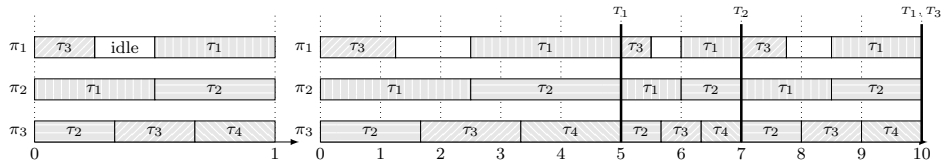


Fig. 8: Number of presences by workload assignment method for unrelated and consistent clusters

	$\dot{m}=2$	$\dot{m}=5$
LP-Feas	0.013	0.464
LP-Load	0.012	0.562
LP-CFeas	0.002	0.027
LP-CLoad	0.002	0.029
Hetero-Split	0.007	NA
ILP-Mig	0.061	NA
ILP-CMig	0.023	0.156

ILP-Mig, $\dot{m}=2$	
n	time (s)
10	0.811
11	1.736
12	3.562
13	8.271
14	16.665
15	28.492
16	69.782
17	130.582

Table 2: Average execution time of the workload assignment methods in seconds



(a) Template schedule example

(b) Stretched template schedule example

Fig. 9: Template-based scheduling

$\dot{m}=2$ than its counterparts, we study how it is sensitive to the number of tasks n on table on the right. It clearly shows that the execution time grows exponentially with the number of tasks, making it intractable for large systems, contrary to ILP-CMig.

6 Global scheduling applicability on heterogeneous platforms

In this section, we examine the applicability of the provided results. Specifically, we discuss the applicability of the global scheduling on unrelated platforms. First, we study the validity of considering migrations upon an unrelated platform. Second, we expose some limitations on the use of a template to schedule tasks in practice.

6.1 How realistic is the migration of tasks upon unrelated platforms?

Actual heterogeneous platforms. The global scheduling theory on unrelated processors assumes that migrations of tasks between processors with *heterogeneous* architectures are possible. As mentioned in [35] to motivate partitioned scheduling, the migration between processors with different instruction sets is at least challenging if not unrealistic. Indeed, tasks code should be compiled for both architectures and migration points determined beforehand if jobs migration is allowed, using soft-migration points as shown in the sample code presented in Section 3.1. Consequently, the context of the migrating task —the state of all active job local variables— should also be saved and transferred to the destination core (e.g. by using the OpenAMP framework) to ensure the continuation of this task at the point it has stopped. Nevertheless, recent platforms as ARM big.LITTLE® are game-changing. Those platforms embed clusters of cores different in their micro-architecture (*asymmetric*) but having similar ISAs with full cache coherence. This

allows migrations between cores belonging to different clusters. Specifically, ARM big.LITTLE® platforms are made of a cluster of fast but energy-consuming (big) cores and a cluster of slower but more energy-efficient (LITTLE) cores. These platforms are unrelated and cannot be classified as uniform as the rate of execution still depends both on the task and the executing core. They are *consistent* (see Section 5.1) in the sense that there exists an order of magnitude between the core processing rate of different clusters: big cores executes tasks always faster than LITTLE cores but not always with the same magnitude.

Applied scheduling in heterogeneous platforms. In practice, the Energy Aware Scheduler (EAS) [21] has been recently integrated in the mainline linux kernel (from version 5.0) and supports migration between single-ISA cores of asymmetric clusters. Based on a model of the cores energy consumption, EAS selects which core to use —the most energy efficient— and at which frequency using the dynamic voltage and frequency scaling (DVFS) mechanism. In line with the existing Completely Fair Scheduler (CFS) [20], EAS aims at providing a fair distribution of the cores utilisation to non real-time tasks while maximising the overall performance but also optimising energy usage. When CFS is designed for identical cores (symmetrical multi-processing or SMP in the OS terminology), EAS takes advantage of recent asymmetric multi-processing (AMP) platforms as ARM big.LITTLE®. SCHED_DEADLINE [31] is an implementation of the global EDF scheduler [11] for real-time tasks in Linux together with the Constant Bandwidth Server (CBS) [6] algorithm to manage non real-time tasks. SCHED_DEADLINE was designed for identical cores (SMP) and may starve all the tasks upon a consistent (AMP) platform. This issue amongst other was discussed by Luca Abeni in talks given during the last two editions of the OSPM summit [22, 5]. He proposed some practical solutions (submitted as a linux kernel patchset in [4]) as adapting the admission control test of tasks and selecting the least energy consuming processor capable of executing the pending task. Interestingly, the latter proposal is valid because the two clusters of cores are consistent. Consequently, studying the global scheduling of unrelated platform is not only theoretical but practicable and the particular case of consistent cores fits well to those *modern* platforms.

6.2 Template-based scheduling

In Section 4, we detail the workload assignment phase of tasks to cores. To the best of our knowledge, any global scheduler on heterogeneous platform starts with such a workload assignment phase. This workload assignment phase is then followed by the construction of a *template schedule*. This *template schedule* contains a feasible schedule of the periodic task set, over a time instant. An example of a template schedule is given in Figure 9(a). We will now discuss the possible usages of this template schedule.

In [9, 19], this template schedule is stretched between every absolute task releases, as illustrated in Figure 9(b). Repeating the template schedule every time unit is acceptable for a feasibility test. It is however not acceptable in practice, due to the number of preemptions and migrations involved. To limit the online overhead, some techniques have been developed to improve the average case, as [14]. This work is designed in the context of an affinity mask model, with sporadic tasks.

In the worst case, however, this optimisation has no effect. Other works, as [39], loosen the hypothesis of hard real-time tasks to soft real-time tasks with bounded tardiness and allow intra-task parallelism via a DAG-task model. This change of paradigm allows to reduce the overhead and thus improves the use in practice. However, this change of paradigm may not be applied in the general case. One could also argue that considering that the rate of a task is constant on a given core is unrealistic. For example, if a task has a first part involving intensive integer computation, followed by a second part of intensive floating point computation, the rate of execution of both parts would differ depending on whether the core has a FPU or not. As a result, the task should be split in more homogeneous sub-tasks, such that each task can be considered as having a constant rate of execution on each core. This requires a DAG-task model or at least a model that handles chains of tasks to be considered.

The use of identical platform scheduling techniques for unrelated platform scheduling has been explored. However, those techniques seem to be hard to generalise to unrelated platforms. In [19], the authors propose a global scheduling algorithm for periodic tasks on a 2-types heterogeneous platform. After the workload assignment phase *hetero-split*, it performs the *2-types McNaughton hetero-wrap*. This *2-types McNaughton* assigns the fractions of processing capacity of a task for both cluster at once while preventing parallelism. To fill both cluster at once, the first cluster is filled from left to right, while the second one is filled symmetrically from the right to the left. Adapting this seminal *McNaughton* to two types platform required several transformations. As indicated in Section 4.6, this transformations based on a symmetrical filling seems hard to generalise to unrelated platforms with any number of types.

As mentioned in [14], the template schedule produced could be optimised in terms of preemptions with heuristics reordering the windows delimited by scheduling points. However, it would not decrease the number of presences and the schedule would remain static.

Repeating the same sequence over and over reduces the adaptability of the system. To avoid this, an option would be to avoid the use of a template schedule. Designing a more dynamic scheduler such as EDF or Last Laxity First (LLF) may be difficult or require a very pessimistic approach. We believe that for consistent platforms, the design of dynamic schedulers will be made much simpler due to the monotonous characteristic of such platforms. The use of consistent platform would ease both the design of the platform and the usability in practice, as those platforms become more and more common on the market.

7 Conclusion

In this work, we have confirmed by practical considerations and experiments on a real heterogeneous platform the intuition whereby inter-processor migrations are more costly than intra-processor migrations. Starting from this observation, we propose a new model to handle those two types of migrations differently. Based on previous works, we use this cluster-based model to design the workload assignment of an optimal scheduler on unrelated platforms. To do so, we propose a LP formulation, with several objective functions. We show that this LP formulation is an exact feasibility test and that its output may be used to construct an of-

fline schedule. We also propose an ILP formulation that is optimal regarding both schedulability and the number of inter-cluster migrations. Using simulation, we demonstrate the impact of our model on the number of inter-cluster migrations. Our new solution outperforms the existing ones. The optimal ILP is used as a reference.

This new workload assignment thus improves the applicability of global scheduling on unrelated platform, by reducing the online overheads. We discuss the applicability of global scheduling in the context of unrelated platform. We show that global scheduling for heterogeneous platforms is actually used in practice. Moreover, we emphasized that a particular case of the unrelated platform model, the *consistent* model, fits well to certain realistic platforms. Consistent platforms are likely to be used in this context, because the theoretical migration model is close to their behaviour for such platforms. We also discuss the existing usages of global scheduling for the general unrelated case and their limitations.

In the future, we intend to evaluate the performance of an online scheduler, such as a global EDF as such an algorithm may be more usable in practice. We also believe that the consistent model is worth investigating. Its particular processing characteristics and its closeness to actual platforms make it an interesting candidate for incoming works. Closing the gap between theory and practice could also be done by observing more complex task model, such as Gang scheduling or a DAG-based task model.

Acknowledgements

First we would like to thank the anonymous reviewers for their valuable suggestions.

The research is done in the context of the SOFIST project, supported by Project ARC (Concerted Research Action) of Federation Wallonie-Bruxelles. This research is also supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 826610.

References

1. clock_getres(2) - linux manual page. URL https://man7.org/linux/man-pages/man2/clock_gettime.2.html
2. RPMsg-lite user's guide: RPMsg component. URL https://nxpmicro.github.io/rpmsg-lite/
3. IEEE standard for information technology- standardized application environment profile (aep)-posix realtime and embedded application support. IEEE Std 1003.13-2003 (Revision of IEEE Std 1003.13-1998) pp. i–164 (2004)
4. Abeni, L.: Rfc patch 0/6] capacity awareness for sched_deadline (2019). URL https://lkml.org/lkml/2019/5/6/11
5. Abeni, L.: Sched_deadline on heterogeneous multicores (2019). URL https://lwn.net/Articles/793281/. Power Management and Scheduling in the Linux Kernel (OSPM summit III)
6. Abeni, L., Buttazzo, G.C.: Integrating multimedia applications in hard real-time systems. In: Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998, pp. 4–13 (1998). DOI 10.1109/REAL.1998.739726
7. Armstrong, R.K.: Investigation of effect of different run-time distributions on smartnet performance. Master's thesis, Naval Postgraduate Scholl, Monterey, California (1997)

8. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming **72**(1-2), 3–21 (2008)
9. Baruah, S.K.: Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In: Real-Time Systems Symposium, pp. 37–46. IEEE (2004)
10. Baruah, S.K.: Partitioning real-time tasks among heterogeneous multiprocessors. In: 33rd International Conference on Parallel Processing (ICPP), pp. 467–474. IEEE (2004)
11. Baruah, S.K., Baker, T.P.: Schedulability analysis of global EDF. Real-Time Systems **38**(3), 223–235 (2008). DOI 10.1007/s11241-007-9047-9
12. Baruah, S.K., Bertogna, M., Buttazzo, G.: Multiprocessor Scheduling for Real-Time Systems. Springer (2015)
13. Baruah, S.K., Bonifaci, V., Bruni, R., Marchetti-Spaccamela, A.: ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. J. Scheduling **22**(2), 195–209 (2019)
14. Baruah, S.K., Brandenburg, B.: Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities. In: Real-Time Systems Symposium, pp. 160–169. IEEE (2013)
15. Bertogna, M.: A view on future challenges for the real-time community (2019). URL https://www.irit.fr/rtns2019/keynote/. RTNS 2019 Keynote
16. Bertout, A., Goossens, J., Grolleau, E., Poczekajlo, X.: Template schedule construction for global real-time scheduling on unrelated multiprocessor platforms. In: Design, Automation and Test in Europe Conference (Grenoble, France, March 2020) (2020)
17. Bertout, A., Goossens, J., Grolleau, E., Poczekajlo, X.: Workload assignment for global real-time scheduling on unrelated multicore platforms. In: Proceedings of the 28th International Conference on Real-Time Networks and Systems, RTNS 2020, p. 139–148. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3394810.3394823. URL https://doi.org/10.1145/3394810.3394823
18. Chen, H., Cheng, A.M.K., Kuo, Y.W.: Assigning real-time tasks to heterogeneous processors by applying ant colony optimization. Journal of Parallel and Distributed computing **71**(1), 132–142 (2011)
19. Chwa, H.S., Seo, J., Lee, J., Shin, I.: Optimal real-time scheduling on two-type heterogeneous multicore platforms. In: Real-Time Systems Symposium, pp. 119–129. IEEE (2015)
20. kernel development community, T.: The linux kernel documentation: Completely fair scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (2019). Accessed: 2019-12-18
21. kernel development community, T.: The linux kernel documentation: Energy aware scheduling. https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html (2019). Accessed: 2019-12-18
22. Corbet, J.: Power-aware and capacity-aware migrations for real-time tasks (2018). URL https://lwn.net/Articles/754923/. Power Management and Scheduling in the Linux Kernel (OSPM summit II)
23. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM computing surveys (CSUR) **43**(4), 35 (2011)
24. Dufossé, F., Uçar, B.: Notes on birkhoff–von neumann decomposition of doubly stochastic matrices. Linear Algebra and its Applications **497**, 108–115 (2016)
25. Goossens, J., Macq, C.: Limitation of the hyper-period in real-time periodic task set generation. In: Proceedings of the RTS Embedded System, pp. 133–148 (2001)
26. Jamil, R., Grolleau, E., Dautrevaux, B., Bertout, A.: Measurement-based timing analysis on heterogeneous mpsocs: A practical approach. In: European Conference on Software Architecture, pp. 279–293. Springer (2020)
27. Karmarkar, N.: A new polynomial-time algorithm for linear programming. Combinatorica **4**(4), 373–396 (1984). DOI 10.1007/BF02579150. URL https://doi.org/10.1007/BF02579150
28. Kokke: tiny-bignum-c (2019). URL https://github.com/kokke/tiny-bignum-c
29. Kostya, M.: crystal-benchmarks-game (2018). URL https://github.com/kostya/crystal-benchmarks-game
30. Lawler, E.L., Labetoulle, J.: On preemptive scheduling of unrelated parallel processors by linear programming. J. ACM **25**(4), 612–619 (1978)
31. Lelli, J., Scordino, C., Abeni, L., Faggioli, D.: Deadline scheduling in the linux kernel. Software: Practice and Experience **46**(6), 821–839 (2016)

32. Levin, G., Funk, S., Sadowski, C., Pye, I., Brandt, S.A.: DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In: 22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010, pp. 3–13. IEEE Computer Society (2010). DOI 10.1109/ECRTS.2010.34. URL https://doi.org/10.1109/ECRTS.2010.34
33. Phavorin, G., Richard, P., Goossens, J., Maiza, C., George, L., Chapeaux, T.: Online and offline scheduling with cache-related preemption delays. Real-Time Systems **54**(3), 662–699 (2018)
34. Phavorin, G., Richard, P., Maiza, C.: Complexity of scheduling real-time tasks subjected to cache-related preemption delays. In: 20th Conference on Emerging Technologies & Factory Automation, pp. 1–8. IEEE (2015)
35. Raravi, G., Andersson, B., Bletsas, K.: Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. Real-Time Systems **49**(1), 29–72 (2013)
36. Raravi, G., Andersson, B., Nélis, V., Bletsas, K.: Task assignment algorithms for two-type heterogeneous multiprocessors. Real-Time Systems **50**(1), 87–141 (2014)
37. Saltzman, M.J.: Coin-or: an open-source library for optimization. In: Programming languages and systems in computational economics and finance, pp. 3–32. Springer (2002)
38. Singh, J., Auluck, N.: Real time scheduling on heterogeneous multiprocessor systems - a survey. In: Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC), pp. 73–78. IEEE (2016)
39. Tang, S., Voronov, S., Anderson, J.H.: GEDF tardiness: Open problems involving uniform multiprocessors and affinity masks resolved. In: 31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany, pp. 13:1–13:21 (2019)