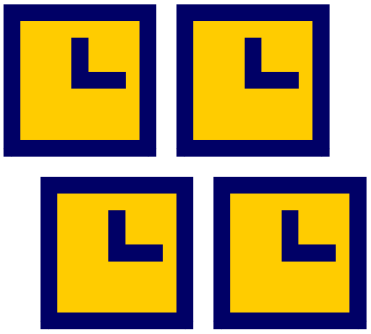

P-SOCRATES

Parallel Software Framework for Time-Critical many-core Systems

Grant Agreement
FP7-ICT-611016

Deliverable type	Report
Deliverable name	Resource allocation requirements
Deliverable number	D3.1
Work Package	3
Responsible partner	UoM
Report status	Draft
Dissemination level	Public
Version number	Version 0.1
Due date	3/31/2014
Actual delivery date	3/24/2014

P-SOCRATES Partners

Name	Short name	Country
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO	ISEP	Portugal
BARCELONA SUPERCOMPUTING CENTER – CENTRO NATIONAL DE SUPERCOMPUTACION	BSC	Spain
UNIVERSITA DEGLI STUDI DI MODENA E REGGIO EMILIA	UoM	Italy
EIDGENOESSISCHE TECHNISCHE HOCHSCHULE ZURICH	ETH Zurich	Switzerland
EVIDENCE SRL	EVI	Italy
ACTIVE TECHNOLOGIES SRL	AT-ITALY	Italy
ATOS SPAIN SA	ATOS SPAIN SA	Spain

Project Coordinator
Dr. Luis Miguel
Email: lmp@isep.ipp.pt

Project Manager
Dra. Sandra Almeida
Email: srca@isep.ipp.pt

Contact information
CISTER Research Centre
Instituto Superior de Engenheria do Porto (ISEP)
Rua Dr. Antonio Bernardino de Almeida, 431
4200-072 Porto, Portugal

Phone: +351 228340502
Fax: +351 228340509

Document History

Version	Date	Author	Description
0.1	2/24/2014	Marko Bertogna	First version.
1.0	4/4/2014	Marko Bertogna	Complete version.

Table of Contents

1 Summary

Deliverable 3.1 summarizes the state-of-the-art in the scheduling and schedulability analysis for real-time multi-core systems, in order to drive the selection of the most appropriate mapping and scheduling algorithms for executing parallel applications on the many-core embedded platforms considered in the project. Also, the main requirements are identified to characterise the desired performance of allocation strategies, including application-specific performance metrics, processor utilization and resource augmentation bounds. The description of the scheduling algorithms is accompanied by considerations on the implementation and run-time overhead of the proposed techniques with particular relation to cache-related penalties, pre-emption and migration overhead, synchronisation delays, scheduling complexity, etc.

The document is the result of Task 3.1. The main existing scheduling strategies are presented, distinguishing between global, partitioned, semi-partitioned and clustered scheduling approaches. The associated schedulability analyses are also outlined, characterizing the relative performance of each proposed algorithm. Finally, the information required from other Work Packages is specified, namely:

- WP1 -- Application Requirements and Evaluation;
- WP2 -- Programming Models and Compiler Techniques;
- WP4 – Timing and Schedulability Analysis
- WP6 -- COTS Many-core platform.

2 Purpose and Scope

2.1 Purpose

The purpose of this document is to drive the selection of the most suitable scheduling and mapping algorithms to reach the final objective of the P-SOCRATES project, determining the input needed from other work packages.

2.2 Scope

The document is public. For the topics covered, the document may be used as a reference for the state-of-the-art in the multi-core scheduling and schedulability analysis for real-time systems.

2.3 Abbreviations and Acronyms

Note that the term *processor* and *core* will be interchangeably used along the document. Also, the term "task" will refer to the concept of *real-time task* adopted in the real-time community, i.e., a recurring process that has to be completed within a specified deadline.

Abbreviation Acronym	Description
RTOS	Real-Time Operating System
EDF	Earliest Deadline First
FP	Fixed Priority
RM	Rate Monotonic
DM	Deadline Monotonic

2.4 Structure of this document

The document consists of 5 chapters. The chapter *"Summary"* provided a brief summary about the content of the deliverable. The chapter *"Purpose and Scope"* explains the purpose, scope, and structure of this document. The chapter *"Real-time scheduling problem"* provides a survey of the scheduling techniques and task models proposed in the real-time literature. The chapter "*Schedulability problem*" details the state-of-the-art in the schedulability analysis of the principal multiprocessor scheduling algorithms. The chapter *"Requirements from other WPs*" contains a list of requirements from the other Work Packages of the project. Chapter *"Bibliography"* close the document by providing a list of references.

3 Real-time scheduling problem

3.1 Introduction

A real-time system has been correctly defined by Alan Burns and Andy Wellings [BW01] as an "information processing system which has to respond to externally generated input stimuli within a finite and specified period: the correctness depends not only on the logical result but also on the time it was delivered; the failure to respond is as bad as the wrong response". These systems are nowadays present in a wide variety of fields, such as control systems, environmental monitoring, avionic and automotive applications.

A real-time *scheduling algorithm* is basically a function defined in the time domain, that determines which task or job has to be executed on each processor. The particular instance produced applying a given scheduling algorithm to a known set of jobs is called *schedule*. The target of a real-time scheduler is to find a schedule in which every job can successfully complete its execution before its deadline. Classical scheduling algorithms used for general purpose systems, like FIFO (processes executed in a first-in-first-out manner) or Round Robin (processes sequentially executed for fixed time-intervals) are not suitable for real-time applications. Even if they can assure good average performances, these algorithms can give very limited guarantees to a system having hard timing constraints. This aspect of the problem has been deeply analysed, and several different scheduling algorithms have been proposed. Below we summarize the most popular ones, classifying them according to their main properties and mechanisms.

Real-time systems designers are typically interested in the following problems:
• The **run-time scheduling problem**: given a set of tasks with real-time requirements, find a schedule that meets all timing constraints.
• The **schedulability problem**: given a set of tasks and a scheduling algorithm, determine before run-time whether all tasks will always meet their deadlines.

Both problems are equally important, since a good scheduling algorithm is almost useless without an associated schedulability test that could tell if some deadline will be missed. Predictability is one of the key properties a good real-time system designer must ensure. The correctness of results strictly depends on the time they are delivered: it is essential to be able to guarantee that results are produced within a certain and known time instant. In other words, the system designer must be able to predict, at least partially, the evolution of the system. Due to the many uncertainties that can modify the behaviour of a computing system, it is often necessary to

sacrifice some precision, deriving worst-case estimations instead of more time-consuming exact parameters.

3.2 Model and Terminology

We will consider platforms composed of one or more computing units. The term processor and core are interchangeably used along the document. Depending on the kind of processors used, Multiprocessor Platforms (MP) are divided into identical, uniform and heterogeneous MPs:

- An **identical** multiprocessor platform is composed of m processors, each one having identical processing capabilities and speed; more specifically, each processor is identical in terms of architecture, memory topology and access times, cache size, I/O interface, resource access and every other mechanism that can influence the overall processor speed. Due to the simplicity of this model, the majority of works in literature address identical multiprocessors.
- A **uniform** multiprocessor platform can be composed of processors of different speeds. The only requirement is that all tasks are executed at the same speed when scheduled on the same CPU.
- A **heterogeneous** (or unrelated) multiprocessor platform represents a set of cores with different capabilities and different speeds. In this model, a processor can execute distinct tasks at different speeds, or even being unable to execute some tasks.

Since the many-core accelerator considered in the project consist of multiple identical cores, this deliverable will mainly focus on the identical multicore platform model.

3.2.1 Task model

The computing workload imposed by a set of processes with real-time requirements can be modelled in many different ways. One of the most used solutions is to model the workload as a set of recurring real-time tasks. In this model, each task executes a sequence of jobs that must complete within a specified deadline. Depending on the frequency at which a task activates its jobs, tasks are distinguished between periodic and sporadic tasks. Every job of a periodic task is activated a fixed amount of time after the previous job. This fixed interarrival time is called *period* of the task. For a sporadic task, instead, only a *minimum interarrival time* is specified, meaning that only a lower bound on the temporal separation between two consecutive jobs must be satisfied.

Each task τ_k is characterized by a three-tuple (C_k, D_k, T_k) composed of:

- A **worst-case computation time** (WCET) C_k. Since the time a task needs to complete its execution depends on branches, cycles, conditional structures, memory states and other factors, we need to consider worst-case values when checking the temporal constraints of an application. As a consequence, C_k represents the maximum number of time-units for which a job of τ_k needs to be scheduled to complete its execution, when the task is executed in isolation (see Deliverable D4.1 for further details on the timing analysis to compute the WCET).
- A **relative deadline** D_k. Since each job has an absolute deadline, which represents the latest time instant at which we can accept the job to complete execution, the relative deadline of task τ_k is simply the maximum acceptable distance between each job activation and the end of its execution.
- A **period** or **minimum inter-arrival time** T_k. Depending on the periodic or sporadic task model used, T_k will denote the exact or minimum time interval between two successive activations of a job of task τ_k.

Depending on the relation between period and deadline parameters of a task system τ , we can distinguish between:

- **implicit** deadline systems, if every task has a deadline equal to its period;
- **constrained** deadline systems, if every task has a deadline less than or equal to its period;
- **unconstrained** (or **arbitrary**) deadline systems, if no constraint is imposed on the values of periods and deadline, that is deadlines can be less than, equal to or greater than periods.

The utilization of a task is defined as the ratio between the task worst-case execution time and period, representing the maximum processor share required by the task. The total utilization of a task set is obtained by summing the utilization of all tasks in the set.

The response time of a task is the longest time that any job of the task may take from its release to its completion. The response time differs from the WCET as it factors in the inteference from the other tasks and from the system. Note that when a task set is schedulable, each task has a response time lower than or equal to its deadline.

Further generalizing the task structure, other models have been proposed to characterize the different sections of a task. In particular:

- **Multi-frame model** [MC96], where each task is composed of different “frames” (with worst-case execution times $C_1,\ldots,C_N$), separated by the same minimum period T.
- **Generalized multi-frame (GMF) model** [BCGM99], where the task frames may be separated by different periods ($T_1,\ldots,T_N$) and deadlines ($D_1,\ldots,D_N$).

- **Recurring Real-Time task (RRT) model** [Bar03], where each task is modeled as a *directed acyclic graph (DAG)*. Each node represents a subtask with a given worst-case execution time and deadline, while edges represent a *potential flow of execution* with an associated minimum inter-release time. After traversing the DAG, each task starts again from its initial vertex after a given task period.
- **Digraph Real-Time task (DRT) model** [SEGY11], where arbitrary (and not only cyclic) directed graphs are possible.

3.2.2 Parallel task models

When tasks can include parallel structures, e.g., when using parallel programming model like OpenMP, Cilk, etc. (see Deliverable D2.1 for a more detailed discussion on the parallel programming models that will be adopted in this project), or are composed of multiple subjobs with precedence constraints, more general task models have been proposed in the literature to characterize intra-task parallelism.

An initial classification of parallel task models can be made by distinguishing between:

- **Rigid tasks**, when the number of cores assigned to each task is a priori determined and cannot change over time;
- **Moldable tasks**, when the number of cores assigned to each task can be decided on-line by the scheduler, but cannot change once the task start executing. This scheduling model has been also called *gang scheduling*;
- **Malleable tasks**, when the number of cores assigned to each task can be dynamically determined by the scheduler at runtime, and may change over time.

In this project, we will consider the *malleable task model*, as it is the most general and flexible one.

To characterize more in detail the parallel task structure, the following task models have been proposed:

- **Fork/join task model** [LKR10], where each task is divided into sequential and parallel segments. Parallel segments must be preceded and followed by a sequential segment. All parallel segments must have the same number of threads, and the number of threads cannot be greater than the number of processors in the platform.
- **Synchronous parallel task model** [SALG11], which generalises the fork/join model by considering tasks composed of different segments that may have an arbitrary number of parallel threads (even greater than the number of cores).
- **Parallel DAG model** [BBMSW12], further generalizing the model by identifying each task with a *directed acyclic graph (DAG)*, where each node represents a sequential job with a

given worst-case execution time, and each directed edge represents a *precedence constraint* between two jobs. Note that, despite the common use of DAGs, this model differs from the recurring real-time (RRT) model, in that nodes do not have an associated deadline, edges do not have an associated inter-release constraint, and, most notably, they *do not represent* potential flows of execution but precedence constraints.

In this project we will mostly focus on the *synchronous parallel task model*. In summary, the real-time tasks considered in this project will be characterized by a sequence of segments composed of a given number of parallel threads, without restrictions on the number of cores where each segment is to be scheduled. If possible, parallel DAG extensions will be explored if the associated complexity of the scheduling algorithm and schedulability tests will be reasonable to justify such a more detailed task characterization.

3.3 Scheduling real-time task sets on multicore platforms

Even if the concept of multiprocessing has always been present in the real-time community, only recently it is receiving a significant attention. The new architectures that the market is offering give an unprecedented opportunity to test existing multicore scheduling techniques upon real platforms. While the scheduling problem for uniprocessor systems has been widely investigated for decades, producing a considerable variety of publications and applications, there are still many open problems regarding the scheduling and schedulability analysis of multiprocessor systems. The increasing demand for smart methods to schedule the computing workload on parallel platforms pushed researchers to further investigate this rather unexplored area. Analysing multicore systems is not an easy task. Checking timely constraints on these kind of systems is even harder. As pointed out by Liu in his seminal paper [Liu69]: “few of the results obtained for a single processor generalize directly to the multiple processor case: bringing in additional processors adds a new dimension to the scheduling problem. *The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors*”.

To extend the limits of current scheduling theory, it is necessary to improve the performances of classical scheduling algorithms, devising new strategies to overcome the main drawbacks that such algorithms show in a multiprocessor environment. Unfortunately, predicting the behaviour of a multiprocessor system requires in many cases a considerable computing effort. To simplify the analysis, it is often necessary to introduce pessimistic assumptions, particularly when modelling globally scheduled multiprocessor systems, in which the cost of migrating a task from a core to another can significantly vary over time. The presence of shared local memories and the frequency

of memory accesses have a significant influence on the worst-case timing parameters that characterize the system.

3.4 Partitioned vs. global scheduling

When deciding which kind of scheduler to adopt in a multicore system, there are two options (see Figure 1):

- **Partitioned scheduling**: statically assign tasks to processors, and using well-known uniprocessor scheduling algorithms;
- **Global scheduling**: tasks are dynamically assigned to cores, and they may migrate from one core to another.

The first method is particularly efficient when there are no load variations and the tasks can be properly characterized. Every processor has its own queue of ready tasks, from which tasks are extracted for execution, according to a local policy. While the implementation of the local scheduler is pretty easy, partitioning tasks to processors is rather complex: the problem of distributing the load to the computing units is analogous to the bin-packing problem, which is known to be NP-hard in the strong sense [GJ79, LW82]. Finding an optimal solution is therefore highly likely to have an exponential complexity. Even if there are heuristics that are able to find acceptable solutions in polynomial or pseudo-polynomial time [DL78, BLOS95, LMM03, LDG04], partitioning algorithms are not sufficiently flexible to deal with variable loads.

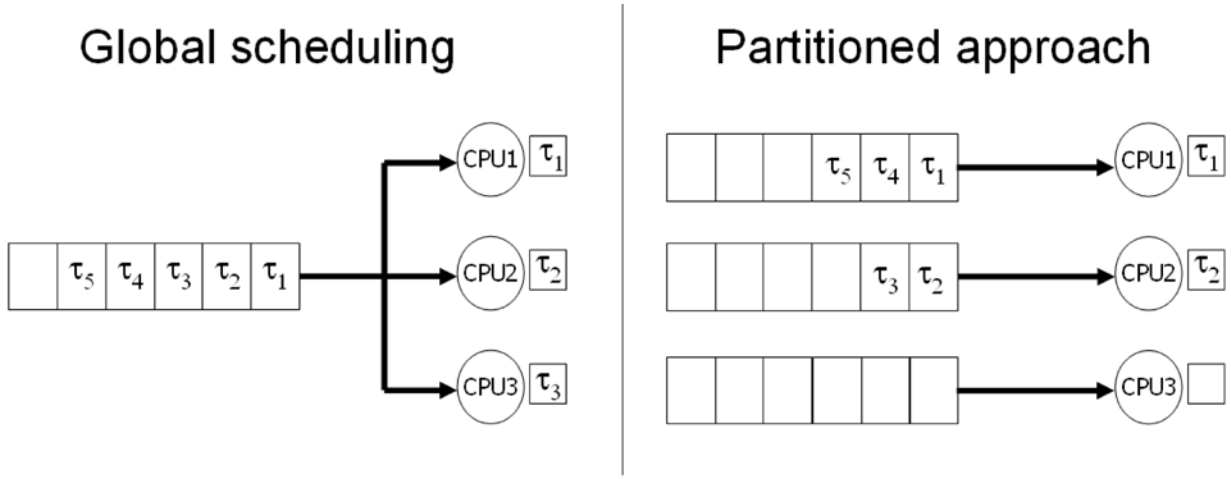

Figura 1. Differences in the implementation between partitioned and global schedulers.

For highly varying computational requirements, a better option is probably to use a global scheduler. With this method, there is a single system-wide queue from which tasks are extracted and scheduled on the available cores. When a processor is idle, a dispatcher extracts from the top

of this queue the highest priority job, and schedules it on the available core until it completes execution, or is preempted by another task. Whenever a task with priority higher than one of the executing tasks is released, the task having lowest priority among the executing ones is preempted and re-inserted in the ready queue. Now, if a processor becomes idle, the preempted task can restart the execution even if the idle processor is different than the original processor upon which the task was originally scheduled. In this case, we say that the task "*migrated*" from a processor to another. This mechanism guarantees that the m highest priority ready tasks (where m is the number of cores in the system) are always the ones executing on the multi-processor platform.

In this way, the load is intrinsically balanced, since no processor is idled as long as there is a ready task in the global queue. The scheduler is said to be work-conserving, in that it doesn't possibly produce a situation in which (i) at least one processor is in idle state, and (ii) a ready task is waiting for execution. Partitioned algorithms are not work-conserving, since it is possible to have a system configuration in which a ready task is assigned to the same processor where a higher priority task is running, while a different processor has an empty ready queue. Instead, a global scheduler would have assigned the waiting task to the available core, allowing a more balanced use of the resources.

A similar argument can be found in queuing theory, where single-queue approaches are known to be superior to multiple queue techniques, with respect to average response time [Kle75]. Even if real-time systems are less interested in average values than in worst-case timely parameters, taking centralized scheduling decision can have interesting advantages for the following reasons:

- There is no need to invoke heavy routines to balance dynamic workloads.
- In overload situations, there are more chances to prevent a task from missing its deadline if we allow the waiting tasks to be scheduled on any processor, instead of statically assigning each task to a fixed processor.
- When a task executes less than its worst-case execution time, the spare bandwidth can be used by any other task that is ready to execute, increasing the responsiveness of the system.
- The more balanced distribution of the workload allows a fairer exploitation of the computing resources, preventing an asymmetric wear of the device and increasing fault tolerance.
- As observed by Andersson et al. in [AJ00a], globally scheduled multiprocessor systems typically have a lower number of preemptions and context changes than partitioned approaches. When a high priority task becomes ready, a global scheduler will preempt some lower priority running task only if there are no idle processors. If instead a partitioned scheduler is used, a high priority task would be scheduled on the assigned

processor even if it is necessary to preempt a task executing with lower priority, and there are other cores in idle state.

However, using a global scheduler there are also complications related to the cost of interprocessor migration. Even if there are mechanisms that can reduce this cost, a significant schedulability loss may be due to tasks having a large memory footprint. Transferring a job context from a processor to another takes some time and may cause a significant number of cache misses and an increased bus load that could have been avoided using a partitioned approach. Therefore, the effectiveness of a global scheduler is rather conditioned by the application characteristics and by the architecture in use.

When analysing the schedulability performances of global and partitioned approaches, no class dominates the other: there are task sets that can be scheduled using a global scheduler but not with a partitioned one, and viceversa [LW82, CFH+03, Bar07]. This complicates the decision on which technique to adopt. There are many factors that can influence this decision. The presence or not of a good (tight) associated schedulability test is very important. *Having a good scheduler without being able to prove the correctness of the produced schedule is almost useless in the real-time field*. Since designers are interested in finding efficient scheduling algorithms at least as much as they are interested in proving that no deadline will be missed, the tightness of existing schedulability tests is key to select which class of schedulers to adopt.

While there are many schedulability tests for partitioned multiprocessor task systems (combining well-known uniprocessor scheduling techniques with solutions for the bin-packing problem) [DL78, BLOS95, LGDG00, LGDG03, LMM03, LDG04], there is still a lot of work to do in the scheduling analysis framework of global approaches. Existing schedulability tests for globally scheduled systems [GFB01, BC07, BCL08, BB11] detect a lower number of schedulable task sets than the existing tests for partitioned systems.

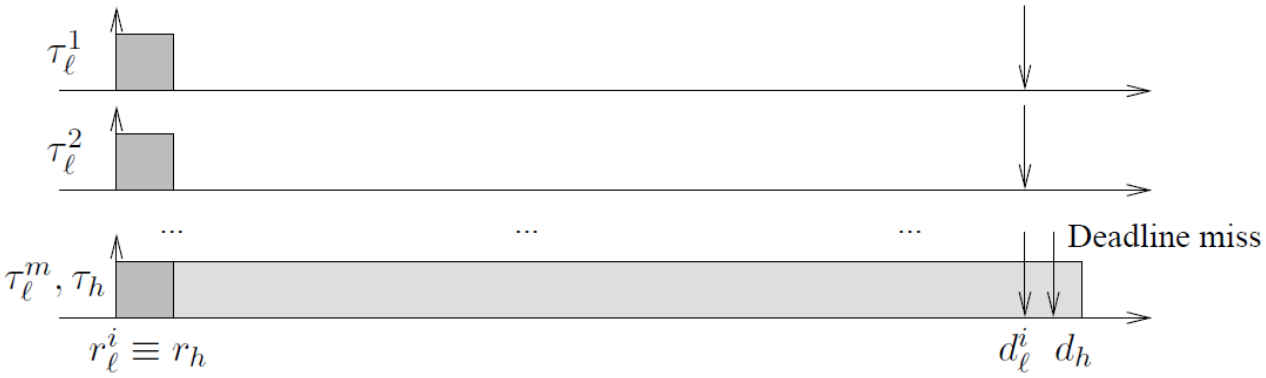

Figure 2. Dhall's effect: a heavy task misses its deadline due to the interference of higher priority light tasks.

A further complication for non-partitioned approaches is given by the poor performances of classic uniprocessor scheduling algorithms. This is due to a particular situation, called Dhall's effect [DL78], that takes place when a bunch of light tasks are scheduled together with a heavy task. As shown in Figure 2, a deadline can be missed even at very low processor utilizations, using either EDF or RM. Therefore, the uniprocessor optimality of EDF and RM does not extend to multiprocessor systems.

3.5 Hybrid scheduling approaches

As an alternative to the global and partitioned scheduling models, it is possible to use **hybrid** approaches that limit the cost of interprocessor migration. In [CAB07], a hybrid migration technique is explored for multiprocessor platforms with a high number of CPUs, allowing a task to be scheduled only on a subset of the available processors (**clustered scheduling**). In this way, every task can migrate among a limited number of processors, reducing cache misses and migration costs. This method is more flexible than a rigid partitioning algorithm without migration, and it is particularly indicated for systems with a large number of cores (as the ones addressed in this project), where it would be very difficult and time-consuming to move a task from a computing unit to a distant one.

Another hybrid scheduling strategy is given by **restricted-migration scheduling**. According to the classification in [CFH+03], this class of schedulers allows task migration, but only at job boundaries, i.e., before a job starts executing or at the end of its execution. In this way, the amount of information that must be transferred while migrating a task from a processor to another is likely to be less than with fully-migrative schedulers [BC05, CFH+03].

A similar method consists in using a global scheduler with **restricted preemptions** [Bar06]: in this way, once a job starts executing on a CPU, it is not possible to preempt it until it completes, so that migration can only take place at job boundaries. However, note that non-preemptable systems can incur significant schedulability losses, due to potential delays caused by long chunks of code that can execute without being interrupted. To sidestep this problem, **limited preemption schedulers** have been proposed to allow preemptions only at the boundaries of properly selected non-preemptive regions of the code [MNPBD13, DBMNPB13].

A final class of hybrid schedulers is given by **semi-partitioned** approaches [AT06, AB08, BA09, KY09, KYI09, GSYY10], which partition most of the tasks to statically assigned processors, and schedule the (few) remaining ones in a global or clustered way.

3.6 Static vs. dynamic priorities

Another possible categorization of existing real-time scheduling algorithms is given by **off-line** and **on-line** schedulers. The first class refers to real-time systems whose schedule is completely computed and stored in memory before run-time. A dispatcher will then simply read from a table in memory the entry corresponding to the current time instant, and schedule tasks to processors accordingly. This kind of scheduling is also called table-driven scheduling [BS89]. To statically compute the schedule, the characteristics of the workload must be completely specified before run-time. For this reason, such class of schedulers is not so appropriate for systems that are partially-specified or have schedules too large to be stored in memory. If this is the case, a better solution is using dynamic (or online) scheduling algorithms. At each time instant, an online real-time scheduling algorithm selects the set of jobs that must execute on the target platform based on prior decisions and on the status of the jobs have been released so far. No specific information is known on future job releases.

A scheduling algorithm that can exploit information on timing parameters (such as arrivals, exact computation times, etc.) of future releases is called clairvoyant[1]. We will mainly focus on *online* real-time scheduling algorithms.

Real-time scheduling algorithms can be also classified according to task and job priorities. Depending on the policy used to sort the ready queue, schedulers are divided into **static-priority**, **fixed job-priority** and **dynamic job-priority** schedulers.

3.6.1 Static priority scheduling algorithms.

This class of algorithms assigns each task a static priority, so that all jobs of a task have the same priority. These algorithms are often shortly called Fixed Priority (FP), omitting to explicitly refer to task priorities. Priorities can be assigned in many different ways. Popular assignment are, for example, Rate Monotonic (RM), assigning priorities inversely proportional to task periods, and

[1] Note that periodic and sporadic task systems define a period or minimum inter-arrival time, which can be seen as information on future releases. Nevertheless, the schedulers that use such information, as well as information on the worst-case execution times, are still conventionally classified as on-line schedulers.

Deadline Monotonic (DM), that assigns priorities inversely proportional to the relative deadlines. Both these priority assignments are optimal for sporadic and synchronous periodic uniprocessor systems with, respectively, implicit and constrained deadlines, meaning that if a sporadic or synchronous periodic task system can be scheduled with FP on a single processor, then it can also be scheduled using RM (for implicit deadlines) [LL73] or DM (for constrained deadlines) [LW82]. For multiprocessor systems, things are much more complicated: RM and DM are no more optimal and can fail even at very low utilizations (arbitrarily close to one) [DL78]. There are however hybrid priority assignment, like Rate Monotonic with Utilization Separation (RM-US) and Deadline Monotonic with Density Separation (DM-DS) [BCL05], that obtain much better scheduling performances by assigning highest priorities to the heaviest tasks (above a given threshold) and scheduling the remaining ones with RM or DM.

The static priority class of algorithm has some particularly desired feature: systems scheduled with static priority algorithms are rather easy to implement and to analyse; they allow reducing the response time of more critical tasks by increasing their priorities; they have a limited number of preemptions (and therefore migrations), bounded by the number of jobs activations in a given interval; they allow selectively refining the scheduling of the system by simply modifying the priority assignment, without needing to change the core of the scheduling algorithm (a much more critical component).

3.6.2 Fixed job-priority scheduling algorithms.

This class allows changing the priority of a task, but only at job boundaries, i.e., every job has a fixed priority. In uniprocessor systems, this increased flexibility in the management of priorities allows significant improvements of the scheduling performances: it has been proved that the Earliest Deadline First scheduling algorithm (EDF) — that schedules at each time-instant the ready job with the earliest absolute deadline — is an optimal scheduling algorithm for scheduling arbitrary collections of job on a single processor [LL73, Der74]. Therefore, if it is possible to schedule a set of jobs such that all deadlines are met, then the same collection of jobs can be succesfully scheduled by EDF as well. Unfortunately, this optimality does not hold for multiprocessors due to
Dhall's effect [DL78]. There are EDF-based hybrid algorithms that overcome the schedulability penalties associated to Dhall's effect, scheduling some task with static priority and some other with EDF. Algorithm EDF-US (Earliest Deadline First with Utilization Separation) [SB02] gives highest (static) priority to the tasks having utilization higher than a given threshold, and schedules the remaining ones with EDF; algorithm EDF-DS (Earliest Deadline First with Density Separation) is an alternative version that uses the densities instead of the utilizations and is more indicated for constrained deadline task systems. Algorithm fpEDF [Bar04b] assigns highest priorities to the first

(m−1) tasks having utilization greater than a half, and schedules the remaining ones with EDF. Algorithm EDFk [GFB01] assigns highest priority to the k heaviest tasks, scheduling again the remaining ones with EDF; for constrained deadlines systems, a better approach seems to be ordering tasks according to their densities instead of utilizations.

The benefits of using a fixed job-priority algorithm is in the relative simplicity of the scheduler implementation, as well as in the bounded number of preemptions and migrations. Since a task can change priority only at job boundaries, the number of preemptions in a fixed time-interval is bounded by the number of job activations in the same interval. This characteristic is common to both static priority and fixed job-priority scheduling algorithms, which are often also called *priority-driven schedulers*.

3.6.3 Dynamic job-priority scheduling algorithms.

These algorithms remove the restriction on the fixed priorities of the jobs, allowing the priority of a job to change at any time. These algorithms are not so popular for single processor applications, since usually a simpler priority-driven scheduler is preferred. For example, the optimality of EDF allows obtaining the best schedulability performances, with an easier implementation and a lower number of context switches than with a dynamic job-priority scheduler. Nevertheless, changing the priority of a job during execution turns out to be much more useful in the multiprocessor case, overcoming Dhall's effect and increasing substantially the number of schedulable task sets.

The Pfair class of algorithms is known to be optimal for scheduling periodic and sporadic real-time tasks with migration when deadlines are equal to periods [BCPV96, AS00], with a schedulable utilization equal to the number of processors. Such algorithms are based on the concept of *quantum* (or slot): the time line is divided into equal-size intervals called *quanta*, and at each quantum the scheduler allocates tasks to processors. A disadvantage of this approach is that all processors need to synchronize at quantum boundaries, when the scheduling decision is taken. Moreover, if the quantum is small, the overhead in terms of number of context switches and migrations may be too high. Solutions to mitigate this problem have been proposed in the literature [HA05]; however, the complexity of the implementation increases significantly.

More recently, other dynamic job-priority algorithms have been proposed to achieve optimal schedulability performances, at a lower preemption cost: BF [ZMM03], LLREF [CRJ06], EKG [AT06], E-TNPA [FKY08], LRE-TL [FN09], DP-fair [LFSPB10], BF^2 [FNGMN10, NSGZNG14], RUN [RLMLB11], U-EDF [NBNGM12]. Note that optimality refers only to a very restricted task model, namely sequential sporadic tasks with implicit deadlines. When task may have deadlines different from periods the problem requires clairvoyance [FGB10].

An interesting algorithm that has the same worst-case number of preemptions of EDF, but much better (even if not optimal) scheduling performances for multiprocessor systems is EDF with zero laxity (EDZL) [LE92]. Algorithm EDZL adds one additional rule to EDF: whenever a job reaches zero laxity, it is scheduled for execution even if there are other jobs with an earlier deadline waiting for execution.

One last algorithm that is worth mentioning is the Generalized Processor Sharing (GPS) server [PG93], which has been initially developed for packet scheduling, but can be easily adapted to multiprocessor real-time systems. It serves all active tasks simultaneously, assigning each one of them a processor share proportional to its utilization. The GPS scheduler is an optimal algorithm for implicit deadline multiprocessor systems, but cannot be implemented on real devices, since it is impossible to arbitrarily divide the processing resources among all requests. Anyway, it is a good reference model for more practical scheduling algorithms, because of its perfectly fair allocation of the available bandwidth.

4 Schedulability problem

As mentioned in the previous section, the schedulability problem for multi-core real-time system is often difficult to solve. Schedulability conditions can be distinguished among:

- **Necessary** test if all schedulable task sets pass the test;
- **Sufficient** test if passing the test implies the schedulability of the task set;
- **Exact** test: a tests that is both necessary and sufficient.

Exact results are available only for restricted task models and algorithms. We hereafter provide a brief summary of the state-of-the-art, with particular relation to schedulability tests that are of interest to the project.

4.1 Performance metrics

To characterize the performance of a scheduling algorithm or a schedulability test, different metrics have been proposed to characterize the distance from an optimal solution. Such a distance may be due to the non-optimal performance of the scheduling algorithm or to the schedulability test being only sufficient.

4.1.1 Utilization-based metrics

For implicit deadline systems, the performance of a given scheduling algorithm can be characterized by computing its *schedulable utilization*, i.e., the minimum utilization among the task sets that are not schedulable according to the algorithm, so that if a task set has a utilization below the given bound, it is deemed schedulable with the considered algorithm. For implicit deadline systems, a necessary and sufficient feasibility condition is given by checking whether the task set utilization is lower than or equal to the number of processors. Therefore, the schedulable utilization of a given scheduling algorithm represents a meaningful measure of the distance from a theoretically optimal solution. The lower the schedulable utilization with respect to the number of processors, the worse the algorithm performance.
For constrained and arbitrary deadline tasks, for which deadlines may differ from periods, the schedulable utilization is no longer a meaningful metric, as there are task sets that are not feasible even with utilization close to zero. Density-based bounds are used instead, where a task density is defined as the ratio between the task worst-case execution time and the deadline (or the minimum between the deadline and the period, in the arbitrary deadline case). A sufficient feasibility test can be derived checking whether the density of a task set is smaller than or equal to

the number of processors. However, density does not allow properly characterizing the distance from an optimal solution, as there are task sets with density tending to infinity, that can be feasibly scheduled even on a single processor system.

4.1.2 Load-based metrics

For constrained and arbitrary deadline systems, load-based metrics appear more suitable to characterize the performance of a scheduling algorithm, at least on a single processor setting. The *load* of a real-time task set is defined as the maximum cumulative execution requirement of the task set in any interval, normalized by the interval length. In other words, the load measures the ratio between the maximum execution demand of jobs that have arrival times and deadlines within an interval, over the execution time available in the considered interval, taking the maximum over all possible intervals. For a single processor system, a necessary and sufficient feasibility condition can be derived by checking whether the load of a task set is smaller than or equal to one. The corresponding test for multiprocessor systems, i.e., checking if the load does not exceed the number of processors, is however only a necessary condition. In fact, it is possible to find task sets with a load arbitrarily close to one that cannot be feasibly scheduled on a multiprocessor platform.

4.1.3 Processor speed-up factor

Another metric that can be used to characterize the performance of a schedulability test is the *processor speedup factor* (also known as *resource augmentation bound*). This is a lower bound on the speedup factor s that guarantees that each feasible task set on a platform composed of identical processors will pass the considered schedulability test on a platform in which each processor is s times as fast. In other words, this means that if the schedulability tests fails, the task set cannot be scheduled with any algorithm on a platform which is at most 1/s times as fast.
Note that the resource augmentation bound “measures” the distance in terms of processor speed from an *ideal* scheduler. However, such an ideal scheduler may be impossible to implement in practice. This is for instance the case with constrained deadline sporadic task systems scheduled upon an identical multi-core platform: an optimal scheduler for these systems would need to be clairvoyant [FGB10], something that cannot be implemented in practice.

4.1.4 Capacity augmentation bounds

A somewhat similar way to characterize scheduling performance is using *capacity augmentation bounds.* A scheduler is said to provide a capacity augmentation bound *c* if it can schedule any task set having a total utilization *m/c*, and the worst-case critical-path length of each task (execution time of the task on an infinite number of processors) is at most *1/c* of its deadline.

The capacity augmentation bound provides more information than a resource augmentation bound (so that a capacity augmentation bound implies an identical processor speed-up bound), and allows deriving a simple schedulability test.

4.1.5 Experimental characterization

A widely adopted method to characterize the performance of a schedulability test is to run experiments using randomly generated sets of tasks. The various tests are then compared in terms of number of generated task sets that are deemed schedulable by each algorithm. In order for the experimental characterization to be meaningful, the task generation method should not be biased in favour of one test or the other, selecting different task distributions to properly highlight the behaviour of the algorithms in each possible scenario.
Typical parameters that are varied among different simulation runs are

- Number of tasks
- Distribution of task utilizations
- Task periods and deadlines
- Number of task segments
- Degree of parallelism of task segments

Unfortunately, it is often impossible to compare the performance of a given schedulability test with that of an exact test, since the few existing exact tests for global scheduling algorithms have a significant complexity that limits their applicability to very limited sets, with small values for the task parameters. An alternative is to compare sufficient tests with simple necessary conditions, evaluating the difference between the number of task sets deemed schedulable by the sufficient test and those not deemed unschedulable by the necessary condition. The larger the gap, the less precise the analysis.

4.2 Partitioned algorithms

The schedulability analysis for partitioned approaches is much simpler than it is for global schedulers, since, after the static partitioning step, it reduces to a single-processor schedulability problem. We hereafter mention the main results in the single-processor case for independent tasks with no precedence constraints:

- EDF is optimal for scheduling generic sets of jobs (including on-line arrivals, periodic tasks, sporadic tasks, etc.) [Der74].

- EDF is able to schedule all implicit deadline periodic or sporadic task sets with utilization not exceeding 1 [LL73]. Exact schedulability analyses for EDF with constrained and arbitrary deadline tasks are available with pseudo-polynomial time [BMR90].
- RM is an optimal fixed priority assignment for synchronous periodic or sporadic tasks with implicit deadlines [LL73].
- DM is an optimal fixed priority assignment for synchronous periodic or sporadic tasks with constrained deadlines [LW82].
- DM and RM are not optimal for arbitrary deadline systems, or when periodic tasks are released with a given offset. Instead, Audsley's priority assignment is optimal in these cases [Aud01].
- Exact schedulability analysis for FP systems are available with pseudo-polynomial complexity, based on the response-time analysis [ABR+93].

Unfortunately, the problem of partitioning a set of tasks to a set of processors is NP-hard in the strong sense [GJ79], so that sub-optimal heuristics are typically used, like First Fit, Best Fit, Worst Fit, etc.

A known lower bound on the achievable utilization with a partitioned scheduler is given by *(m+1)/2* , and is easily derived considering a situation in which there are *m+1* identical tasks, each one having a worst-case execution time slightly larger than half its period. In this case, there is no possible partitioning of the tasks on the m available processors (as the maximum schedulable utilization of a single processor is 1). There are partitioning algorithms that are able to achieve the above bound for fixed priority systems [AB03].

For EDF-based partitioned systems, the best possible utilization bound that can be achieved is given by *(x/m+1)/(x+1)*, where *x* is the floor of $1/U_{max}$, being U_{max} the maximum utilization among all tasks. Allocation policies that are able to achieve such a bound are Best Fit and First Fit [LDG04].

4.3 Global algorithms

The schedulability analysis for global algorithms is much more complicated. The problem is that no worst-case task release pattern is known in this case, since the synchronous arrival of all tasks with jobs released as soon as possible is proven not to be a critical instant when more than one processor is used (even though it is a critical instant on a single processor platform).
Different necessary conditions are known in the global case, such as:

- Total utilization less than or equal to the number of processor;

- Maximum total task load (maximum execution request in a given time-interval over the time interval itself) less than or equal to the number of processor [FB07,FB08];
- Total force-forward load (a refinement of the above defined load) less than or equal to the number of processor [BC06a].

Various sufficient schedulability tests are also available for fixed-priority and EDF-based global schedulers. Due to the complexity and variety of these tests, a detailed description is here omitted. A survey on existing EDF schedulability tests is available in [BB11]. The corresponding extensions for FP are often straightforward. Notably, the schedulability analysis for FP systems is much tighter than that for EDF-based systems, owing to the possibility of limiting the interfering contributions to the higher priority tasks only (instead of all tasks, which is the case for EDF).

The best performing schedulability tests are based on response-time analysis techniques [BC07b], upper bounding the interfering contributions by considering the densest possible packing of jobs in a window of interest. However, there is still a large gap between such sufficient schedulability tests and an exact test. The few existing exact tests available in the literature can only be applied to very limited task sets, with small values for the task parameters.

Among hybrid priority schedulers, algorithm fpEDF is able to guarantee a schedulable utilization of *(m+1)/2* assigning highest priority to tasks having a utilization higher than *½* and scheduling the remaining ones with EDF [SB02]. Among fixed priority schedulers, algorithm RM-US[1/3] guarantees a schedulable utilization of *(m+1)/3* assigning highest priority to tasks having a utilization higher than 1/3 and scheduling the remaining ones with RM.

The analysis for clustered schedulers reduces to the global analysis, plus a partitioning stage. In general, to perform a sound, quantitative comparison, in terms of generated overhead on a real systems, the actual overhead generated by each solution should be measured. An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers is provided in [BBA10], where it is shown that global EDF is the most effective solution with up to 8 processors, whereas partitioning becomes more effective with a higher number of processors.

Finally, only few works address the schedulability analysis of globally scheduled parallel task models, like the fork/join model, synchronous parallel task model, and DAG-based model. The tighter empirical results seem to be again given by response-time analysis based techniques [CLPES13]. Other works are concerned with theoretical speed-up augmentation bounds and capacity augmentation bounds, which, however, provide weaker results in terms of schedulable task sets detected among randomly generated workloads [LKR10, SALG11, BBMSW12, NBGM12, AN12, LALG13, BMSW13].

4.4 Tardiness bounds

Another scheduling problem on which the real-time community has focused is instead related to soft real-time (SRT) tasks, i.e., tasks whose deadlines may be missed without compromising the correctness of the applications they represent, provided that some quality-of-service requirement is still met. One such requirement is that job tardiness be not higher than some application-specific threshold, where tardiness is defined as the maximum between 0 and the difference between job deadlines and completion times. For example, this guarantee may allow typical SRT applications, such as video players, to perfectly conceal deadline misses through buffering.

This problem is in fact the following SRT problem: “is it possible to guarantee a bounded tardiness on a multiprocessor platform?”. This problem can be generalized as: “is it possible to guarantee bounded job-completion (response) times?”. This thread of research basically starts with [DA05], where global EDF is proved to guarantee bounded tardiness with implicit deadline task sets, under the only constraint that the utilization of each task and the total utilization of the task set do not exceed, respectively, one and the capacity of the platform.

Another technique for computing tighter worst-case tardiness bounds for implicit-deadline tasks has been proposed in [EDB10], based on compliant-vector analysis (CVA). Such bounds have then been generalized for task sets with arbitrary deadlines in [EGB10]. Other tardiness bounds have been provided for different EDF-like scheduling algorithms [LA07,EA12].

The tardiness and lateness bounds mentioned so far are all computed assuming worst-case job execution times. This is unfortunately a serious issue, because, as the number of processors increases, worst-case execution times can become orders of magnitude higher than average ones. This problem becomes particularly relevant on many-core platforms. In this respect, timing analysis tool are unfortunately not yet able to provide reasonably tight upper bounds on execution times on multiprocessors.

4.5 Memory-aware scheduling

The task models analysed in the previous sections made the implicit assumption that all the overhead due to the concurrent access to the shared memory be already included in the worst-case execution times of the tasks. However, such an assumption may be too pessimistic, leaving wide room for improving the system performance.

More precisely, the memory access latency can be divided into two main categories:

- physical access delay: representing the required time to physically retrieve the data from memory, considering the actual task running in isolation;
- memory contention: modelling the time during which a task is suspended as the memory is used by another task or device.

Note that hard-coded bus arbitration policies do not consider task priorities for the memory accesses, meaning that if a low priority task commands the DMA to copy a wide memory area from a location to another, higher priority tasks which need to query the memory will have to wait for a long time.

Memory-aware scheduling techniques have been proposed to optimize memory accesses in order to reduce the related latency which may affect the real-time feasibility of the system. Yun et al. [YYP+13] introduced MemGuard, a framework which aims at guaranteeing memory performance isolation similarly to what happens for CPUs. More precisely, performance isolation means that the average memory access latency of a task is no longer than when the task runs on a dedicated memory system. Such a property is achieved by assigning a fraction of bandwidth to each core. Any time a core communicates with the memory, its budget is decreased accordingly and, when the budget is exhausted, memory requests are delayed until next period, when the budget is refilled.

Pellizzoni et al. [PBCS08] studied the impact of the peripherals on the task execution times and provided a solution for preserving real-time feasibility. More precisely, they empirically tested that peripherals with heavy I/O load increase the overall memory access latency, making jobs last even 44% longer. First, a hardware referee is introduced to regulate the device access to the bus. Then, a WCET analysis is introduced to split the code into a sequence of atomic computational chunks. At runtime, when the running chunk terminates, if the difference between its worst-case length and the actual duration is not shorter than the maximum delay introduced from the device, then the slack time is granted to the latter. In other words, all the devices are seen as an additional best-effort task which runs when there are no pending real-time jobs.

Bak et al. [BYPC12] investigated through exhaustive simulations which memory-aware scheduling policy works better when PREM compliant tasks are involved. The PREM (PRedictable Execution Model) model considers tasks whose execution is explicitly divided into a retrieving phase which copy data from the main memory into the cache and an execution phase which runs computation using previously-cached data. Two opposite classes of algorithms are considered: TDMA-based and Prioritized memory schedulers. The first approach allows cores to access the main memory only during predefined periodic slots, according to a TDMA fashion, in order to provide core isolation. Moreover, tasks in memory phase are promoted rather than those in computation

phase. The other approach lets the running tasks compete for the use of memory at any time and, if the access is denied, they are suspended and queued. Whenever the actual memory access terminates, a global arbiter grants the memory to one of the pending jobs according to a predefined policy (FIFO, EDF, shortest-memory phase first and an approximated least-laxity first). The algorithm which guarantees the highest schedulability rate is M-LAX which exploits EDF for scheduling the tasks on each core, while promoting memory phases rather than CPU computation, and the bus arbiter grants the memory access according to the least laxity heuristic (the shorter the task slack time, the higher its priority).

In [PSCCT10] the worst-case interference due to simultaneous accesses to the main memory on multicore systems is studied. The workload is executed periodically and each program is divided into a set of superblocks (containing condition and loop statements internally) which are executed sequentially. Tasks running on the same core are scheduled according to fixed time slots and the cache is invalidated when a slot starts. Given a task partition, the algorithm starts computing, for each processing unit, the upper bound of the amount of memory traffic generated by the assigned workload. Then, for each task the longest delay is computed while considering the interference due to the tasks on the other cores and the peripherals.

Kim et al. [KNAKMR14] presented an analysis for tightly bounding the memory interference delay in multicore systems. The model takes into account a DDR3 SDRAM main memory with shared/private banks, while the memory controller implements the First-Ready First-Come First-Serve (FR-FCFS) policy. The analysis starts considering the required time to retrieve data in memory when the core shares or not the memory bank with other cores. Then, the interference due to concurrent memory accesses related to other tasks running on other cores is considered. Finally, such overheads are combined to provide an overall latency upper bound which is included in the classical response time analysis.

5 Requirements from other WPs

The requirements of WP3 from the other workpackages are hereafter detailed.

5.1 Application timing requirements (WP1)

For all three application scenarios, the timing requirements have to be properly characterized. This includes workload-based performance metrics, such as the average and worst-case amount of computing workload that needs to be processed in a given amount of time, as well as real-time requirements, such as application deadlines and desired response-times.

5.2 Programming model (WP2)

In order to comply with the task model adopted in the scheduling and schedulability analysis (malleable synchronous parallel task model), the programming model should allow easily deriving the real-time task structure[2]. Precedence constraints among different task segments should be properly modelled. The parallel structure of each segment has to be well characterized in terms of number of parallel regions and degree of allowed parallelism. The recurring behaviour of each task has to be modelled through a period or minimum inter-arrival time. Also, the timing requirements of each task have to be characterized by specifying a relative deadline.

5.3 Timing analysis and platform characterization (WP4 and WP6)

The timing analysis should be able to provide an efficient characterization of the worst-case execution time of each task segment when executed in isolation. Also, an accurate analysis should be made of all sources of overhead in the considered platform, i.e., memory access time (at all memory levels), inter- and intra-cluster communication, bus contention, hard-coded arbitration of buses and NoC connections, cache misses, pre-emption and migration overhead.

A proper characterization of the system will allow selecting the most suitable scheduling algorithm to improve the predictability, lending the system to an efficient and tight schedulability analysis.

[2] Note that the term task here refers to the real-time notion of tasks (a recurring process with a given deadline), and not to the notion adopted in the HPC community (e.g., OpenMP task).

5.4 Operating System (WP5)

The impact of preemptions and migrations on the system overhead has to be properly characterized and measured. In order to design the most suitable scheduling algorithms for the adopted platform, it is necessary to know how scheduling decisions affect the overall system overhead. The decision on where to store the context of the pre-empted threads will be taken in close collaboration with WP5, in order to limit as much as possible the context-switch overhead. Also, hybrid techniques like limited preemptive approaches may be considered to limit both the schedulability and the pre-emption overhead. In order to select the most suitable technique, it is necessary to know which possible implementations are allowed by the OS to enable/restrict preemptions. Also, the decision on the migration model to implement will be taken in close collaboration with WP5.

Another input needed from WP5 is related to the policies to adopt for arbitrating the access to mutually exclusive shared resources. Depending on the adopted policy, particular synchronization mechanisms, thread queues and blocking primitives may be needed. Moreover, the project will also evaluate novel techniques to schedule not only threads to cores, but also memory accesses to bring instructions and data to the local memory of each core. A co-scheduling approach of core computations and DMA accesses will be implemented on the considered platform. In order to do this, a detailed study will be conducted along with WP5 to evaluate the most suitable option to coordinate and enforce the scheduling decisions among the various cores and DMAs.

6 Bibliography

[AB03] B. Andersson and J. Jonsson 2003. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In Proceedings of the 15th Euromicro Conference on Real-Time Systems.

[AB08] Bj¨orn Andersson and Konstantinos Bletsas, Sporadic multiprocessor scheduling with few preemptions. Proceedings of the Euromicro Conference on Real-Time Systems. 2008.

[ABR+93] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings, Applying new scheduling theory to static priority preemptive scheduling, Software Engineering Journal 8 (1993), no. 5, 285–292.

[AJ00a] Bj¨orn Andersson and Jan Jonsson, Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition, Proceedings of the International Conference on Real-Time Computing Systems and Applications (Cheju Island, South Korea), IEEE Computer Society Press, December 2000.

[AN12] B. Andersson and D. de Niz. “Analyzing Global-EDF for Multiprocessor Scheduling of Parallel Tasks”. In: Principles of Distributed Systems. 2012, pp. 16–30

[AS00] James Anderson and Anand Srinivasan, Pfair scheduling: Beyond periodic task systems, Proceedings of the International Conference on Real-Time Computing Systems and Applications (Cheju Island, South Korea), IEEE Computer Society Press, December 2000.

[AT06] Bj¨orn Andersson and Eduardo Tovar, Multiprocessor scheduling with few preemptions. Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA), 2006.

[Aud01] N. Audsley 2001. On priority assignment in fixed priority scheduling. Inform. Process. Lett. 79, 1, 39–44.

[BA09] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. Proceedings of the 30th Real-Time Systems Symposium, 2009.

[Bar03] S. K. Baruah, “Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks,” Real-Time Syst., vol. 24, no. 1, pp. 93–128, 2003.

[Bar04b] Sanjoy Baruah, Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors, IEEE Transactions on Computers 53 (2004), no. 6.

[Bar06] Sanjoy Baruah, The non-preemptive scheduling of periodic tasks upon multiprocessors, Real-Time Systems: The International Journal of Time-Critical Computing 32 (2006), no. 1-2, 9–20.

[Bar07] Sanjoy Baruah, Techniques for mutliprocessor global schedulability analysis, Proceedings of the IEEE Real-time Systems Symposium (Tucson), IEEE Computer Society Press, December 2007.

[BB11] Marko Bertogna, Sanjoy Baruah, "Tests for global EDF schedulability analysis", Journal of Systems Architecture. 57(5): 487-497. 2011.

[BB11] Marko Bertogna, Sanjoy Baruah, "Tests for global EDF schedulability analysis", Journal of Systems Architecture. 57(5): 487-497. May 2011.

[BBA10] A. Bastoni, B. Brandenburg, and J. Anderson, “An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers,” in Proceedings of the 31st IEEE Real-Time Systems Symposium, 2010, pp. 14–24.

[BBMSW12] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, “A generalized parallel task model for recurrent real-time processes,” in RTSS, 2012.

[BC05] Sanjoy Baruah and John Carpenter, Multiprocessor fixed-priority scheduling with restricted interprocessor migrations, Journal of Embedded Computing 1 (2005), no. 2, 169–178.

[BC06a] Theodore P. Baker and Michele Cirinei, A necessary and sometimes sufficient condition for the feasibility of sets of sporadic hard-deadline tasks, Proceedings of the Work-In-Progress (WIP) session of the 27th IEEE Real-Time Systems Symposium (RTSS’06) (Rio de Janeiro, Brazil), December 2006.

[BC07] M. Bertogna and M. Cirinei, Response time analysis for global scheduled symmetric multiprocessor platforms. Proceedings of the Real-Time Systems Symposium. 149–158. 2007.

[BC07b] Marko Bertogna and Michele Cirinei, Response-time analysis for globally scheduled symmetric multiprocessor platforms, 28th IEEE Real-Time Systems Symposium (RTSS) (Tucson, Arizona (USA)), 2007.

[BCGM99] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, “Generalized multiframe tasks,” Real-Time Syst., vol. 17, no. 1, pp. 5–22, 1999.

[BCL05] M. Bertogna, M. Cirinei and G. Lipari, New schedulability tests for real-time tasks sets scheduled by deadline monotonic on multiprocessors, Proceedings of the 9th International Conference on Principles of Distributed Systems (Pisa, Italy), IEEE Computer Society Press, December 2005.

[BCL08] M. Bertogna, M. Cirinei and G. Lipari, Schedulability analysis of global scheduling algorithms on multiprocessor platforms. IEEE Trans. Paral. Distrib. Syst. 20, 4, 553–566. 2008.

[BCPV96] Sanjoy Baruah, Neil Cohen, Greg Plaxton, and Donald Varvel, Proportionate progress: A notion of fairness in resource allocation, Algorithmica 15 (1996), no. 6, 600–625.

[BLOS95] Almut Burchard, Jorg Liebeherr, Yingfeng Oh, and Sang H. Son, New strategies for assigning real-time tasks to multiprocessor systems, IEEE Transactions on Computers 44 (1995), no. 12, 1429–1442.

[BMR90] Sanjoy Baruah, Aloysius K. Mok, and Louis E. Rosier, Preemptively scheduling hard-real-time sporadic tasks on one processor, Proceedings of the 11th Real-Time Systems Symposium (Orlando, Florida), IEEE Computer Society Press, 1990.

[BMSW13] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. “Feasibility Analysis in the Sporadic DAG Task Model”. In: ECRTS. 2013, pp. 225–233.

[BS89] Theodore P. Baker and Alan C. Shaw, The cyclic executive model and ada, Real-Time Systems 1 (1989), no. 1, 7–25.

[BW01] Alan Burns and Andy Wellings, Real-time systems and programming languages, 3rd ed., Addison-Wesley, 2001.

[BYPC12] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo, “Memory-aware scheduling of multicore task sets for real-time systems,” in Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, ser. RTCSA ’12, Aug 2012, pp. 300–309.

[CAB07] John M. Calandrino, James H. Anderson, and Dan P. Baumberger, A hybrid real-time scheduling approach for large-scale multicore platforms, Proceedings of the Euromicro Conference on Real-Time Systems (Pisa), 2007.

[CFH+03] John Carpenter, Shelby Funk, Phil Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah, A categorization of real-time multiprocessor scheduling problems and algorithms, Handbook of Scheduling: Algorithms, Models, and Performance Analysis (Joseph Y.-T Leung, ed.), CRC Press LLC, 2003.

[CFH+03] John Carpenter, Shelby Funk, Phil Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah, A categorization of real-time multiprocessor scheduling problems and algorithms, Handbook of Scheduling: Algorithms, Models,and Performance Analysis (Joseph Y.-T Leung, ed.), CRC Press LLC, 2003.

[CLPES13] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, “Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms,” in ECRTS, 2013.

[CRJ06] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen, An optimal real-time scheduling algorithm for multiprocessors, 27th IEEE Real-Time Systems Symposium (RTSS) (Rio de Janeiro, Brazil), December 2006.

[DA05] U. C. Devi and J. H. Anderson, “Tardiness bounds under global edf scheduling on a multiprocessor.” in RTSS. IEEE Computer Society, 2005, pp. 330–341.

[DB11] RI Davis, A Burns. A survey of hard real-time scheduling for multiprocessor systems ACM Computing Surveys (CSUR) 43 (4), 35. 2011.

[DBMNPB13] Robert I. Davis, Alan Burns, Jose Marinho, Vincent Nelis, Stefan M. Petters, Marko Bertogna, “Global Fixed Priority Scheduling with Deferred Pre-emption”, Proceedings of 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2013), Taipei (Taiwan), August 2013. Best paper award.

[Der74] Michael Dertouzos, Control robotics : the procedural control of physical processors, Proceedings of the IFIP Congress, 1974.

[DL78] Sudarshan K. Dhall and C. L. Liu, On a real-time scheduling problem, Operations Research 26 (1978), 127–140.

[EA12] J. P. Erickson and J. H. Anderson, “Fair lateness scheduling: Reducing maximum lateness in g-edf-like scheduling,” in ECRTS, 2012, pp. 3–12

[EDB10] J. P. Erickson, U. Devi, and S. K. Baruah, “Improved tardiness bounds for global edf,” in ECRTS, 2010, pp. 14–23.

[EGB10] J. Erickson, N. Guan, and S. K. Baruah, “Tardiness bounds for global edf with deadlines different from periods,” in OPODIS, 2010, pp. 286–301.

[FB07] Nathan Fisher and Sanjoy Baruah, The global feasibility and schedulability of general task models on multiprocessor platforms, Proceedings of the EuroMicro Conference on Real-Time Systems (Pisa, Italy), IEEE Computer Society Press, July 2007.

[FB08] Nathan Fisher and Sanjoy Baruah The feasibility of general task systems with precedence constraints on multiprocessor platforms, Real-Time Systems: The International Journal of Time-Critical Computing (2008), In submission.

[FGB10] N. Fisher, J. Goossens, and S. Baruah. “Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible”. In: Real-Time Systems Journal 45.1-2 (2010), pp. 26–71.

[FKN08] K. Funaoka, S. Kato and N. Yamasaki, Work-conserving optimal real-time scheduling on multiprocessors. In Proceedings of the Euromicro Conference on Real-Time Systems. 13–22. 2008.

[FN09] S. Funk and V. Nadadur, LRE-TL: An optimal multiprocessor algorithm for sporadic task sets. In Proceedings of the Real-Time Networks and Systems Conference. 159–168. 2009.

[FNGFN10] S. Funk, V. Nelis, J. Goossens, D. Milojevic, G. Nelissen and V. Nadadur, “On the design of an optimal multiprocessor real-time scheduling algorithm under practical considerations (extended version)“. arXiv preprint arXiv:1001.4115. 2010.

[GFB01] Jo¨el Goossens, Shelby Funk, and Sanjoy Baruah, Priority-driven scheduling of periodic task systems on multiprocessors, Real Time Systems 25 (2001), no. 2–3, 187–205.

[GJ79] Michael R. Garey and David S. Johnson, Computers and intractability : a guide to the theory of NP-completeness, W. H. Freeman and company, NY, 1979.

[GSYY10] N. Guan, M. Stigge, W. Yi, and G. Yu, Fixed-priority multiprocessor scheduling with Liu and Layland’s utilization bound. In Proceedings of the 16th Real-Time and Embedded Technology and Applications Symposium, 2010.

[HA05] Philip Holman and James H. Anderson, Adapting pfair scheduling for symmetric multiprocessors, J. Embedded Comput. 1 (2005), no. 4, 543–564.

[Kle75] Leonard Kleinrock, Queueing systems. volume 2: Computer applications, John Wiley & Sons, New York, 1975.

[KNAKMR14] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar, “Bounding memory interference delay in cots-based multi-core systems,” in Proceedings of the IEEE International Conference on Real-Time and Embedded Technology and Applications Symposium, ser. RTAS ’14, 2014.

[KY09] S. Kato and N. Yamasaki, Semi-partitioned fixed-priority scheduling onmultiprocessors. Proceedings of the Real-Time and Embedded Technology and Applications Symposium. 2009.

[KYI09] S. Kato, N. Yamasaki and Y. Ishikawa, Semi-partitioned scheduling of sporadic task systems on multiprocessors. Proceedings of the Euromicro Conference on Real-Time Systems. 2009.

[LA07] H. Leontyev and J. H. Anderson, “Generalized tardiness bounds for global multiprocessor scheduling,” in In Proc. of the 28th Real-Time Systems Symp, 2007, pp. 413–422.

[LALG13] J. Li, K. Agrawal, C.Lu, and C. Gill. “Analysis of Global EDF for Parallel Tasks”. In: Euromicro Conference on Real Time Systems. 2013.

[LDG04] Jos´e M. L´opez, Jos´e L. D´iaz, and Daniel F. Garc´ia, Utilization bounds for EDF scheduling on real-time multiprocessor systems, Real-Time Systems: The International Journal of Time-Critical Computing 28 (2004), no. 1, 39–68.

[LE92] Suk Kyoon Lee and Donald Epley, On-line scheduling algorithms of real-time sporadic tasks in multiprocessor systems, Tech. Report 92-3, University of Iowa, Computer Science, 1992.

[LFSPB10] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, Scott Brandt. DP-Fair: A Simple Model for Understanding Multiprocessor Scheduling. Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS), Brussels, Belgium, Pages 1 - 10, July, 2010

[LGDG00] J. M. Lopez, Manuel Garcia, Jose L. Diaz, and Daniel F. Garcia, Worst-case utilization bound for EDF scheduling in real-time multiprocessor systems, Proceedings of the EuroMicro Conference on Real-Time Systems. 2000.

[LGDG03] Jos´e M. L´opez, Manuel Garc´ia, Jos´e L. D´iaz, and Daniel F. Garc´ia, Utilization bounds for multiprocessor rate-monotonic scheduling, Real-Time Systems: The International Journal of Time-Critical Computing. 2003.

[Liu69] C. L. Liu, Scheduling algorithms for multiprocessors in a hard real-time environment, JPL Space Programs Summary 37-60 II (1969), 28–31.

[LKR10] K. Lakshmanan, S. Kato, and R. R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in Proceedings of the 2010 31st IEEE Real-Time Systems Symposium, RTSS ’10, 2010, pp. 259–268.

[LL73] C. L. Liu and James Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM 20 (1973), no. 1, 46–61.

[LMM03] Sylvain Lauzac, Rami Melhem, and Daniel Moss´e, An improved rate-monotonic admission control and its application, IEEE Transactions on Computers 58 (2003), no. 3.

[LW82] Joseph Y.-T Leung and Jennifer Whitehead, On the complexity of fixed-priority scheduling of periodic, real-time tasks, Performance Evaluation 2 (1982), 237–250.

[LW82] Joseph Y.-T Leung and Jennifer Whitehead, On the complexity of fixed-priority scheduling of periodic, real-time tasks, Performance Evaluation 2(1982), 237–250.

[MC96] Mok, A. K., and Chen, D. 1996. A multiframe model for real-time tasks. Proceedings of the 17th Real-Time Systems Symposium.

[MNPBD13] Jose Marinho, Vincent Nelis, Stefan M. Petters, Marko Bertogna, Robert I. Davis, “Limited Pre-emptive Global Fixed Task Priority”, Proceedings of 34th IEEE Real-Time Systems Symposium (RTSS 2013), Vancouver, Canada, December 2013.

[NBGM12] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, “Techniques optimizing the number of processors to schedule multi-threaded tasks,” in ECRTS, 2012.

[NBNGM14] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, “U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks“, 24th Euromicro Conference on Real-Time Systems (ECRTS), 2012.

[NSGZNG14] G. Nelissen, H. Su, Y. Guo, D. Zhu, V. Nelis and J. Goossens, “ An optimal boundary fair scheduling“, Real-Time Systems Journal, 2014.

[PBCS08] R. Pellizzoni, B. Bui, M. Caccamo, and L. Sha, “Coscheduling of cpu and i/o transactions in cots-based embedded systems,” in Real-Time Systems Symposium, 2008, Nov 2008, pp. 221–231.

[PG93] Abhay K. Parekh and Robert G. Gallager, A generalized processor sharing approach to flow control in integrated services networks: the single node case, IEEE/ACM Transactions on Networking 1 (1993), no. 3, 344–357.

[PSCCT10] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, “Worst case delay analysis for memory interference in multicore systems,” in Proceedings of the Conference on Design, Automation and Test in Europe, ser. DATE ’10. European Design and Automation Association, 2010.

[RLMLB11] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, “RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor,” in IEEE 32nd Real-Time Systems Symposium (RTSS), 2011.

[SALG11] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” Real-Time Systems Symposium, IEEE International, vol. 0, pp. 217–226, 2011.

[SB02] Anand Srinivasan and Sanjoy Baruah, Deadline-based scheduling of periodic task systems on multiprocessors, Information Processing Letters 84 (2002), no. 2, 93–98.

[SEGY11] M. Stigge, P. Ekberg, N. Guan, and W. Yi, “The Digraph Real-Time Task Model,” in Proc. of RTAS 2011, pp. 71–80.

[YYP+13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms,” in Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), ser. RTAS ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 55–64.

[ZMM03] Dakai Zhu, Daniel Moss´e, and Rami G. Melhem, Multiple-resource periodic scheduling problem: how much fairness is necessary?, 24th IEEE Real-Time Systems Symposium (RTSS) (Cancun, Mexico), 2003