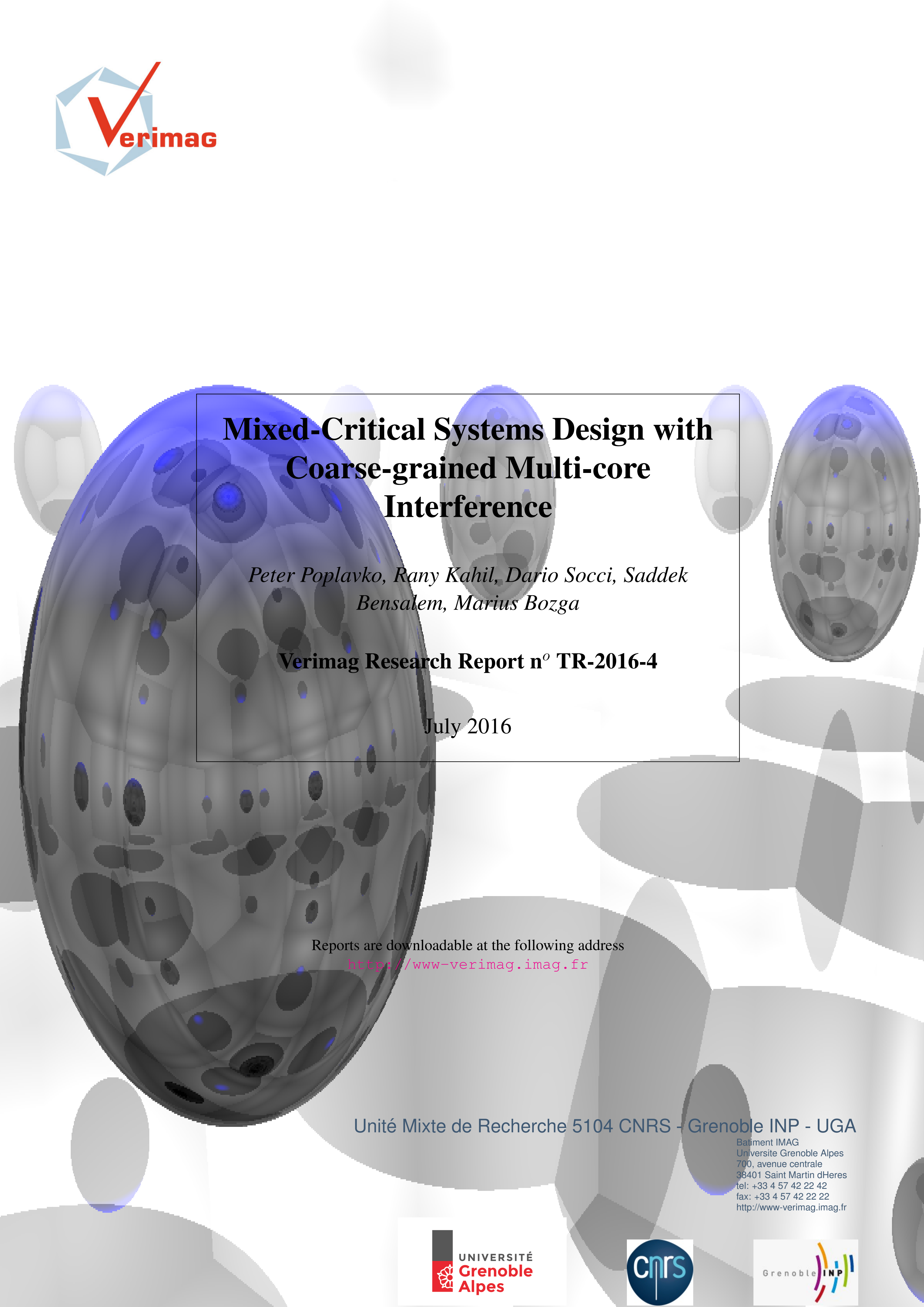


Mixed-Critical Systems Design with Coarse-grained Multi-core Interference

Peter Poplavko, Rany Kahil, Dario Socci, Saddek Bensalem, Marius Bozga

Verimag Research Report n^o TR-2016-4

July 2016

Mixed-Critical Systems Design with Coarse-grained Multi-core Interference[1]

Peter Poplavko, Rany Kahil, Dario Socci, Saddek Bensalem, Marius Bozga

July 2016

Abstract

Autonomic concurrent systems that are timing-critical and compute intensive need special resource managers in order to ensure adaptation to unexpected situations. So-called mixed-criticality managers may be required that adapt system resource usage to critical run-time situations (*e.g.,* overheating, overload, hardware errors) by giving the highly critical subset of system functions priority over low-critical ones in emergency situations. Another challenge comes from the fact that for modern platforms – multi- and many- cores – make the scheduling problem more complicated because of their inherent parallelism and because of "parasitic" interference between the cores due to shared hardware resources (buses, FPU's, DMA's, *etc.*). In our work-in-progress design flow we provide the so-called concurrency language for expressing, at high abstraction level, new emerging custom resource management policies that can handle these challenges. We compile the application into this language and combine it resource manager into a joint representation used to deploy the given solution on the target platform. In this context, we also discuss our current work in progress on scheduling tools for handling the multi-core interference in mixed-critical applications.

Keywords: bandwidth interference, multi-core, embedded multiprocessor, mixed criticality

Reviewers:

How to cite this report:

```
@techreport {TR-2016-4,
  title = {Mixed-Critical Systems Design with Coarse-grained Multi-core Interference},
  author = {Peter Poplavko, Rany Kahil, Dario Socci, Saddek Bensalem, Marius Bozga},
  institution = {{Verimag} Research Report},
  number = {TR-2016-4},
  year = {2016}
}
```

[1]The final publication at ISOLA'16 will be available at Springer via http://dx.doi.org/TO BE PROVIDED

Research supported by ARROWHEAD, the European ICT Collaborative Project no. 332987, and MoSaTT-CMP, European Space Agency project, Contract No. 4000111814/14/NL/MH

1 Introduction

In this paper we present our work-in-progress design flow for scheduling and deployment of software designs for embedded systems. Modern embedded applications constitute so-called *nodes* of *distributed systems*, *i.e.,* they communicate via buses and networks with other applications (nodes). We consider systems that are not only *timing-critical*, *i.e.,* subject to hard real-time constraints, but also *mixed-critical*, *i.e.,* able to sustain highly-critical functions even under harsh compute-resource shortage situations. The latter is desirable if the system has to be *autonomic* [1], *i.e.,* able to operate in open and non-deterministic environments. An example of an autonomic mixed timing-critical system is a "fleet of UAV's (unmanned air vehicles) [2]" that coordinate with the leader UAV within strict time bounds to avoid mutual collision and to ensure timely response to external events, *e.g.,* appearance of hazards. Such systems should not only be correctly specified but also implementable, first of all *schedulable in real-time*. The point is that control tasks in many applications are augmented by complex computations that can load the processor significantly (*e.g.,* computer vision, trajectory/route calculation, image/video coding, graphics rendering). In such cases, to meet the high computational demands inside the nodes while keeping their energy consumption, cost and weight manageable it is important to consider multi- (2-10) or even many-core (x100's cores/'accelerators') platforms.

A major obstacle for schedulability analysis of multi-core applications is 'bandwidth interference' [3], *i.e.,* blocking due to conflicts in simultaneous accesses to shared hardware resources, such as buses, FPU's, DMA channels, IO peripherals. Next to interference, the other dimensions in the scheduling problem are (i) possible lack of preemption support in many-core systems, (ii) inter-task precedences (dependencies), commonly implied from the application's model of computation (MoC) and (iii) switching between normal and emergency mode in mixed-critical scheduling. To be able to address all these dimensions at the same time we propose simplifications which make the scheduling problem amenable for known heuristic methods with some adaptations.

We also put the proposed scheduling approach into the context of our work-in-progress design flow, which offers not only scheduling but also deployment on the platform. The deployment is ensured by a compilation tool-chain that is by construction customizable to various MoCs and online scheduling policies by mapping them to an expressive intermediate 'concurrency' language.

In Section 2 we introduce one-by-one the main pillars of our design flow, such as MoCs and mixed-criticality. Section 3 introduces the structure and assumptions of the proposed flow itself and illustrates it via a small synthetic application example. Section 4 gives a basic explanation of the scheduling models and algorithms and presents some experiments for a large set of random benchmarks. Section 5 concludes the paper and discusses future work.

2 Background

2.1 Models of Computation

To manage concurrency and coordination between tasks in parallel and distributed environments Models of Computations (MoCs) have been proposed in the literature. They permit the application designer to define the structure and organize the tasks and their communication channels in a way that resembles high-level specifications (functional diagrams). MoCs intend to abstract the application's behavior from any implementation detail. Figure 1 shows an example: a part of an industrial avionics application modeled in a MoC called Fixed Priority Process Network (FPPN) [4].

In the figure we see (1) tasks, *e.g.,* 'HighFreqBCP', *etc.*, annotated by periods, (2) inter-task channels, *e.g.,* between 'DopplerConfig' and 'SensorInput', and (3) precedence relation between tasks, *e.g.,* 'HighFreqBCP' has higher precedence than 'BCPConfig'. The application consumes data from *input buffers*, *e.g.,* 'AnemoData', and produces the results to *output buffers*, *e.g.,* 'BCP Data'. The buffers are supposed to hold their input values or output slots during the periodic interval between task arrivals and deadlines. As a MoC, FPPN should define the partial ordering of execution and interaction steps of concurrent activities (tasks), and this is done via the precedence relation, which ensures predictable inter-task communication.

Next to FPPN, many MoCs have been proposed in the literature for embedded multi-core systems, to

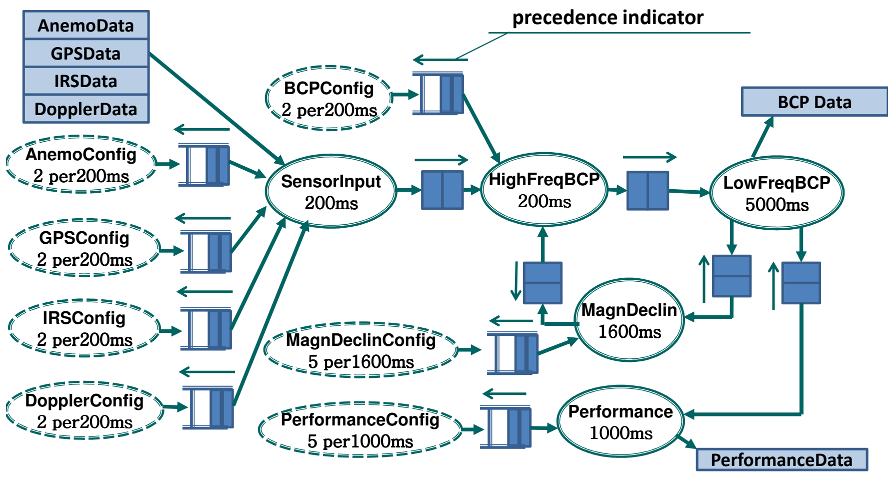


Figure 1: Application modeled in a MoC: Flight Management System in FPPN

name just a few: MRDF (multi-rate data-flow, often named SDF – Synchronous Dataflow) [5], Prelude [6], SADF (scenario-aware data-flow) [7] and DOL-Critical [8].

2.2 Resource Managers and Concurrency Language

An important property of autonomic embedded systems is their ability to adapt themselves to unexpected phenomena [1]. When a system is compute-intensive (which should be the case when a multi-core implementation is necessary) and time-critical it has to be able to adapt itself to exceptional shortage in compute resources. In real-time systems, '*resource managers*' are software functions that monitor utilization of compute resources and ensure such adaptation. For this they apply different mechanisms, such as mixed-criticality, QoS management, DVFS (Dynamic Voltage and Frequency Scaling), *etc.*. Especially the mixed-criticality approaches are gaining more an more interest and have a high relevance for collective adaptive systems [2]. Resource managers are implemented as integral part of *online schedulers i.e.,* middlewares for customized online scheduling policies.

Unfortunately, there is a considerable semantical gap between different online schedulers and middlewares for MoC, even though both define software concurrency behavior. We aim at a common approach that can ensure consolidation, by representing middlewares in a language that is expressive enough such that it can encompass all possible concurrency behaviors for real-time systems, including their timing constraints. We refer to that common language as *concurrency* language (or backbone language) [9].

We believe that for autonomic timing-critical systems a proper choice of concurrency language is a combined procedural and timed automata language extended with tasks, *i.e.,* so-called *task automata* [10, 11]. Timed-automata languages in general are known to be convenient means to specify resource managers, such as QoS [12] and mixed criticality [13]. In our design flow the concurrency language is BIP. Under 'BIP' we mean in fact its 'real-time dialect' [12], designed to express networks of connected timed automata components. In [14] this language was demonstrated useful for modeling distributed autonomic systems. In [8] it was extended from timed to task automata, by introducing the concept of *self-timed* (or 'continuous') automata transitions, *i.e.,* transitions that have non-zero execution time, in order to model task execution.

In our approach, the applications are still programmed in their appropriate high-level MoC because in many cases an automata language, though being appropriate for resource managers, may still be too low-level for direct use in application programming. Instead, we assume automatic *compilation* of higher-level MoCs into the concurrency language. Due to well-known high expressive power of automata to model concurrent systems this must be possible for most MoCs. In ideal case, the compilation would be configured by a user-defined set of grammar rules for automatic translation of his/her preferred MoC into automata.

2.3 Concurrency Language based Representation of System Nodes

Figure 2 gives a generic structure of a concurrency language model of a distributed-system node running an application expressed in a certain MoC. We also zoom into the BIP model of an important part of the system.

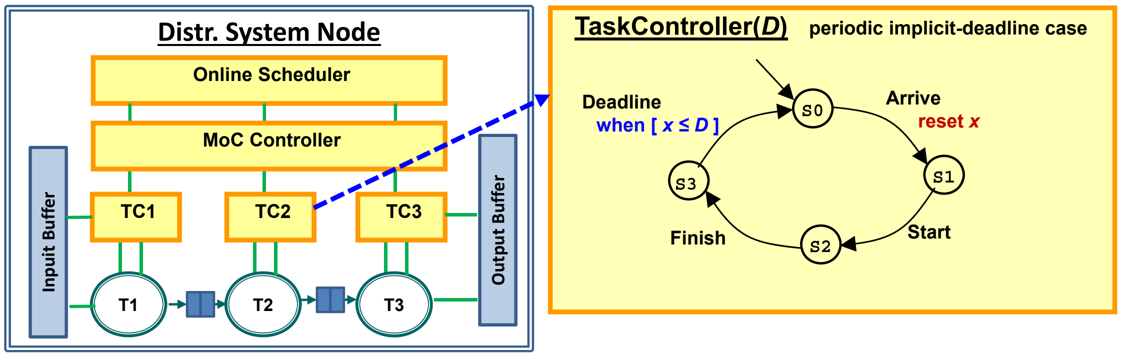


Figure 2: Concurrency Language Representation of a Timing-critical Application

The basic components of the model are automata, *i.e.,* finite-state machines that can interact with other components by participating in a set of interactions with other automata as they make discrete *transitions* (basic steps of execution). In our model, we have one automaton per application task and one per inter-task channel, and also an automaton to control each task – the so-called *task controller*. There is also an automaton that ensures proper task execution order according to model-of-computation semantics, we refer to that automaton as MoC controller. One can also introduce an automaton that further restricts the ordering and timing of task executions – the online scheduler. Note that some automata can be hierarchical, *i.e.,* they may represent a composition of several more primitive automata.

In Figure 2 we zoom into a task controller for periodic tasks whose deadline is equal to the period. It consists of a cyclic sequence of states, with initial state 'S0' and first transition 'Arrive', which models task arrival and is followed by transition 'Start', which corresponds to starting a new iteration of task execution, called a *job*. The 'Start' transition is followed by 'Finish' transition when the job finishes. After the finish, the deadline-check transition 'Deadline' is executed. The deadline is checked as follows: upon task arrival a so-called clock variable x is reset to zero. This variable acts a timer indicating the time elapsed since the last clock reset. After the job has finished we check whether the deadline D was respected, *i.e.,* whether $x \leq D$.

Note that the given task controller example is time-driven, but depending on the employed MoC it can also be event-driven, where events can indicate availability of task input data, or task buffer space or other conditions prescribed by a given MoC for task execution.

2.4 Multi-core Interference Aspects

When dealing with multi-core platform architectures as targets for timing critical applications a particular serious problem arises. Spontaneous unpredictable or hardly predictable 'parasitic' timing delays – '*interference*' – manifest themselves when multiple threads run in parallel in the same hardware system; they are caused by parallel activity of other threads. Interference appears when threads await response from resources that are in use by other threads.

The concerned resources can be either hardware or protected logical (software) resources. Shared hardware resources that can cause interference can be global buses, bus bridges and switches, coprocessors, peripherals, and even FPU's (if they are shared between cores to save on-chip area). Software shared resources are, for example, mutex-lock segments in the source code and calls for mutually exclusive services in the system runtime environments.

Interference can be *coarse-grain* or *fine-grain*. In the former case the accesses to the shared resource occurs in 'coarse' blocks, called superblocks [15], which occur just once or a few times per task execution. Often a task has one superblock to read all the input data from global to local memory at the start and to write the data at the end. Fine-grain interference is sporadic and can occur a large number of times per task execution, *e.g.,* bus accesses due to loads/stores in the memory.

Interference deteriorates the canonical 'worst-case execution time (WCET) analysis' $\rightarrow$ 'schedulability analysis' design process of timing-critical systems due to a feedback influence. Luckily, coarse-grain interference can be *controlled* by scheduling the superblocks in a way that influence of resource conflicts on the WCET is eliminated because the ordering of shared-resource accesses is taken care of by the schedule. Moreover, fine-grained interference can sometimes be partly or completely transformed into coarse-grained one by 'concentrating' the resource-access intensive parts of source code together into coarse-grained blocks.

In our scheduling algorithm we assume fully controlled coarse-grained interference, whereas the remaining fine-grained interference that could not be transformed into coarse-grained one is assumed to be taken into account either via extra WCET margins or, more conservatively, by modeling complete tasks as blocks. Moreover, though different resources (*e.g.,* different FPU's and different memory banks) can be accessed in parallel and though different blocks can have different timing costs, we make a simplifying assumption that there is only one shared resource and the duration of all blocks is the same, we denote it δ. A particular case of such interference that emerges in the context of our concurrency language based design approach is runtime overhead of the centralized 'engine' [8]. In our concurrency model, governed by automata, one can distinguish discrete steps of execution, which correspond to discrete transitions of the automata components that constitute the system. Suppose that these transitions for all system-node components are realized by a single central control thread called the *engine*. This is, for example, the case in the runtime environment for the BIP language employed in our design flow. Suppose that δ is the worst-case time to handle one automaton transition. Then the runtime overhead to execute certain sequence of control operations can be conveniently modeled as the number of discrete transitions times δ [8]. Beyond the necessary accesses to the engine itself, accesses to other interfering resources can be, in principle, also modeled and programmed in a concurrency language as explicit transitions, and this approach is well-known in the literature. For example, in [16], coarse-grained accesses to global bus are explicitly modeled as extra special sub-tasks which are scheduled next to ordinary tasks and which are joined with tasks in a common state-transition model. In principle, our presented scheduling method can handle a general class of coarse-grain interference models that can be reflected in a static task graph. We exemplify this by modeling the interference in the engine. We assume constant block size δ, whereas blocks with duration longer than δ can be modeled by chains with multiple δ-transitions. Therefore, our assumptions can be used in quite general case, though possibly introducing some inefficiency.

We assume single shared resource equal-block duration coarse-grained interference model and refer to it as *δ-interference* model. Generalizing δ-interference for improved efficiency to multiple resources and different block durations is future work.

To manage interference in hard real-time systems we advocate a *time-triggered scheduling* approach, *i.e.,* letting the tasks start at fixed time instances even if previous tasks finish earlier. This approach does not make worst-case response-times of tasks worse, while it significantly reduces the complexity of fine-grain interference analysis (if the corresponding WCET margins are to be computed) and improves its accuracy. The point is that when tasks do not shift their execution earlier upon earlier completion of previous tasks the number of task pairs that can potentially run in parallel (and hence interfere) is significantly reduced, which effectively cuts the number of analysis cases to be covered.

2.5 Mixed-Criticality Aspects

In adaptive autonomous systems one has to provide for unexpected situations. In terms of scheduling this means allocating worst-case amount of resources with a significant extra margin. To damp the high costs that such margins incur, the allocated extra resources are given, 'on an interim basis', to less-critical and less important functions in the system which can be stopped at any time to free up the resources in the case when highly-critical and highly-important functions need them. This reasoning leads to a generic resource management approach commonly referred to as mixed-criticality, see Figure 3.

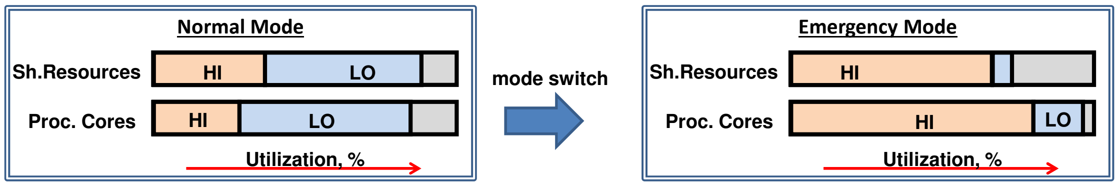


Figure 3: Mixed-criticality Resource Management

We currently consider a common case of having just two levels of criticality. Less-critical functions are given low criticality level, commonly denoted 'LO'. Highly-critical functions are given high criticality level, commonly denoted 'HI'. For example, in a UAV system LO can correspond to mission critical and HI to flight-critical functions.

As shown in Figure 3, in case of emergency the HI tasks get high resource utilization margins. However in normal mode of operation these margins are never used and are given to LO tasks. Only when emergency situation occurs where HI tasks need more resources a 'mode switch' from normal to emergency mode is performed by the resource manager whereby the extra margins are 'claimed' by HI tasks. In our approach, the respective resource management policy can be implemented in concurrency language as part of the 'online scheduler' automaton component [13].

There are two distinct approaches to free up the resources from LO tasks in the case of mode switch. The first approach is *dropping* the LO tasks (*i.e.,* instantaneous aborting them with possibility to resume their execution later on). The second approach is putting the LO tasks in *degraded mode*, *i.e.,* signalling or enforcing them to do less computations and accesses to shared resources at the cost of the lower output quality or missed deadlines. A major challenge in mixed criticality scheduling is that the mode switch may occur at any time not known in advance and that it is required to guarantee schedulability no matter whether and when the switch occurs [17].

As explained in the previous section, to better handle interference we use the time-triggered scheduling, to be more specific, we use STTM (static time triggered per mode) online policy [17, 18], which is a generalization to mixed-criticality scheduling. In this policy, the normal and the emergency modes both have a time-triggered table. A switch from normal to emergency table can occur at any time instant, while it should be guaranteed that if HI critical tasks need to claim their extended resource budgets reserved for unpredictable situations then they will always get them in full amount. Though this appeared to be by far not trivial, in [18] we have proved theoretically and experimentally that this approach is as optimal in the worst case as the event-triggered approach.

3 Work-in-progress: Design Flow

3.1 Underlying Paradigm

There is neither a single MoC nor a single online scheduling policy that would be recognized universal for all application domains and platforms. This is especially the case for multiprocessor systems and when interference, task-dependency and mixed-criticality challenges are to be considered. The policies and MoCs will continue intensive evolution whereas industrial systems need rapidly adjustable implementations, while the corresponding analysis techniques need a basis to establish formal proofs for them. Therefore our target design flow is customizable, at least conceptually, to different MoCs and policies by compiling the MoC and scheduling policy to the common task-automata based concurrency language, for which, in our design flow, we use BIP. Therefore, we do not create a custom middleware specialized for FPPN MoC and for STTM scheduling policy, but instead we express them in BIP [9, 8]. The BIP implementation of the system on top of BIP runtime environment (RTE) should not leave the underlying platform any significant real-time scheduling decision freedom but should map the user-programmed scheduling policies to basic operating system mechanisms, like threads and dynamic priorities [8, 19].

The main contribution of the present paper is handling coarse-grained interference in the context of mixed-critical systems with precedence constraints between multi-rate tasks. We address the complex problem by practically meaningful simplifications. We assume that the task system is synchronous-periodic or can be over-approximated as such by periodic servers. A synchronous system can be represented by a semantically-equivalent static task graph, [20, 4], conveniently presentable to a list-scheduling heuristic, which, in turn, has reputation of reasonable performance for comparable practical instruction-level scheduling problems [21]. Moreover, we present a practical design flow where applications can be programmed. Existing design flows that integrate programming and scheduling *e.g.,* [10, 2, 6, 8, 22, 23], make some restrictive assumptions but in return offer different features, *e.g.,* distributed-system/network support or expressive power. We compare to [8] in the next section. The scheduling techniques [20, 17, 11, 15, 24, 18, 16, 7] also have some restrictions, while in return offering important theoretical properties and features. We discuss related work further in Section 4.3.

3.2 Flow Structure and Assumptions

Our target design flow is shown in Figure 4. At the input we take the application specified as a MoC instance (*i.e.,* a network of task elements connected to channel elements and annotated by parameters) and functional code for the tasks. From the MoC instance the tools derive a task-graph for offline scheduling. The task graph describes the application hyperperiod in terms of jobs nodes and precedences edges. The 'jobs' are task executions and the precedences are derived from the semantics of the given MoC. The application is translated into concurrency language – BIP. The schedule obtained from the offline scheduler is translated into parameters of the online-scheduler model specified in BIP.

The joint application-scheduler model (with a basic structure as previously outlined in Figure 2) is translated by the BIP compiler into a C++ executable. The executable is linked with BIP RTE (the 'engine') and executes on a platform on top of the real-time operating system.

When executing on the platform, the binary executable encounters interferences, as discussed in Section 2.4. Handling interference requires a feedback loop from the binary executable back to the offline scheduler tool. Next to the worst-case execution times (WCET's) of tasks, the worst-case execution times of (blocks) of accesses to the shared resources should be obtained from WCET analysis and back-annotated at the input of the scheduler tool, and then the flow should be re-iterated (at most once, as the 'pure' WCET should not depend on the schedule).

We put the following requirements on our target design flow. We assume FPPN as application MoC. The offline scheduler should support non-preemption, precedence constraints implied from the FPPN and take into consideration coarse-grained interference. The online scheduler should support task migration and task dropping. The online scheduling should be based on STTM scheduling policy for mixed criticality.

The main reason of assuming non-preemption is lack of support of preemption in the current version of BIP language and RTE engine. Though preemption can be modeled and simulated [13], it cannot yet be executed in real-time mode. This is subject of future work, where we will possibly extend the concept of self-timed (*i.e.,* continuous) automata transitions to retractable (*i.e.,* preemptable) transitions. Nevertheless a justification for considering non-preemption is lack of support of preemption in many multi-core platforms that have a large number (> 8) cores (so-called many-core platforms and graphical accelerators).

In our design flow we reuse certain elements from our previous 'DOL-BIP-Critical' flow [8] which was co-developed in collaboration with partners. The name of the MoC involved in that flow was DOL-Critical. It is closely related to FPPN, and the same specification language, named DOL-C, is currently used to specify instances of both FPPN and DOL-Critical models. FPPN has more general notion of task precedence than DOL-Critical, as it supports precedences between any pair of tasks, and not only between equal-rate periodic tasks.

There were essential differences in the scheduling assumptions taken in the previous flow, where the tasks were executed essentially in as-soon-as-possible (ASAP) fashion *i.e.,* immediately after the previous task mapped to the same partition. Instead we impose time-triggered start of each task, which should

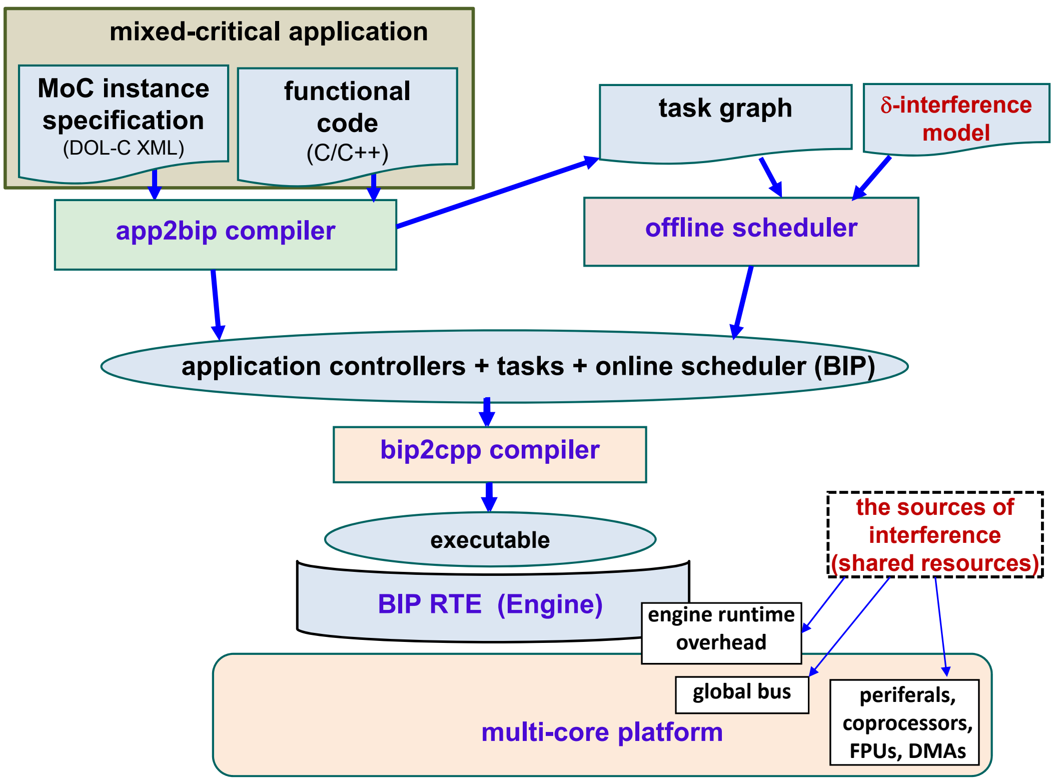

mixed-critical application
MoC instance specification
(DOL-C XML)
functional code
(C/C++)
app2bip compiler
task graph
δ-interference model
offline scheduler
application controllers + tasks + online scheduler (BIP)
bip2cpp compiler
executable
BIP RTE (Engine)
the sources of interference (shared resources)
engine runtime overhead
global bus
periferals, coprocessors, FPUs, DMAs
multi-core platform

Figure 4: Work-in-progress Design Flow

significantly simplify the analysis of bandwidth interference. The offline scheduler of previous flow had the advantage of supporting time partitioning, degraded mode and excluding the interference between HI and LO criticality levels.

Currently in our work-in-progress we have an initial version of an offline scheduler that satisfies the desired criteria, except that the interference models presented at the input of this tool are currently restricted to those for BIP engine interference of implicit-deadline periodic task controllers. Though advanced interference detection methods are known in related work [25], we still miss them in our flow. If such tools were available we could adapt or extend the δ-interference model assumed in the offline scheduler. Next to this, the online scheduler is not yet properly integrated, as it still does not support dropping and task migration, though such features are within reach, *e.g.,* we demonstrate a restrictive form of BIP-component migration in [8] and thread API's offer means for dropping.

In the remainder of the paper we discuss the currently available tools and illustrate their use by concrete examples. For multi-core experiments presented here, we use a LEON4 multiprocessor implemented on FPGA board, using an RTEMS OS with symmetric multiprocessing. For this platform, as measurements show, the worst-case execution time of one BIP interaction step takes: $\delta = 1$ ms.

3.3 An Example Illustrating the Flow

Figure 5 gives a synthetic application example with three tasks. The 'split' task puts two small (a few bytes) data items to the two output channels and sleeps for around 1 ms to imitate some task execution time. Tasks 'A' and 'B' read the data. Task 'A' sleeps alternately for 6 ms and 12 ms, to model 'normal' and 'emergency' workload levels. This task models a high-criticality task. Task 'B' supports two modes of execution: normal and degraded. In normal mode it sleeps for 6 ms, in degraded mode it skips all execution, even reading the input data. This task models a low-criticality task.

All tasks have the same periodic scheduling window, with period and deadline being 25 ms. In a real application, this would correspond to the time during which the two imaginary input data buffers should be read, computations should be done and the output buffers should be written.

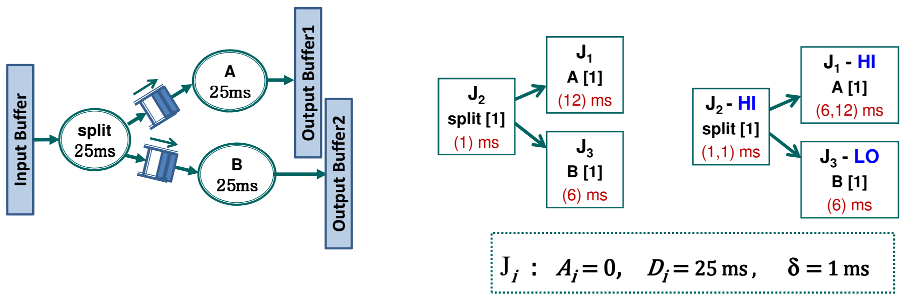


Figure 5: Three-Task Example: MoC (left), Ordinary Task Graph (middle) and Mixed-critical Task Graph

The middle part of the figure gives the 'ordinary' (*i.e.,* non mixed-critical) variant of the task graph. Every task is represented by a job. The jobs are numbered: $J_i = J_1, J_2, J_3$ and annotated by their worst-case execution times. Their individual arrival times A_i and deadlines D_i are the same in this example. The right part of the figure corresponds to the 'mixed-critical' variant of the same graph. The execution times of highly-critical tasks are represented by a two-valued vector: normal-mode time and emergency-mode time.

The engine runtime overhead (as it will become clear later) constitutes $4\delta = 4$ ms per task (in total 12 ms). Therefore, when assuming ordinary execution times this example cannot run on a single core, as the total execution time amounts to $12+1+12+6$=31 ms, which is larger than the 25 ms deadline. The offline scheduler evaluates the load (*i.e.,* maximal demand-to-capacity ratio) of this example to $31/25$=124 %. Therefore it predicts that at least two processors are necessary.

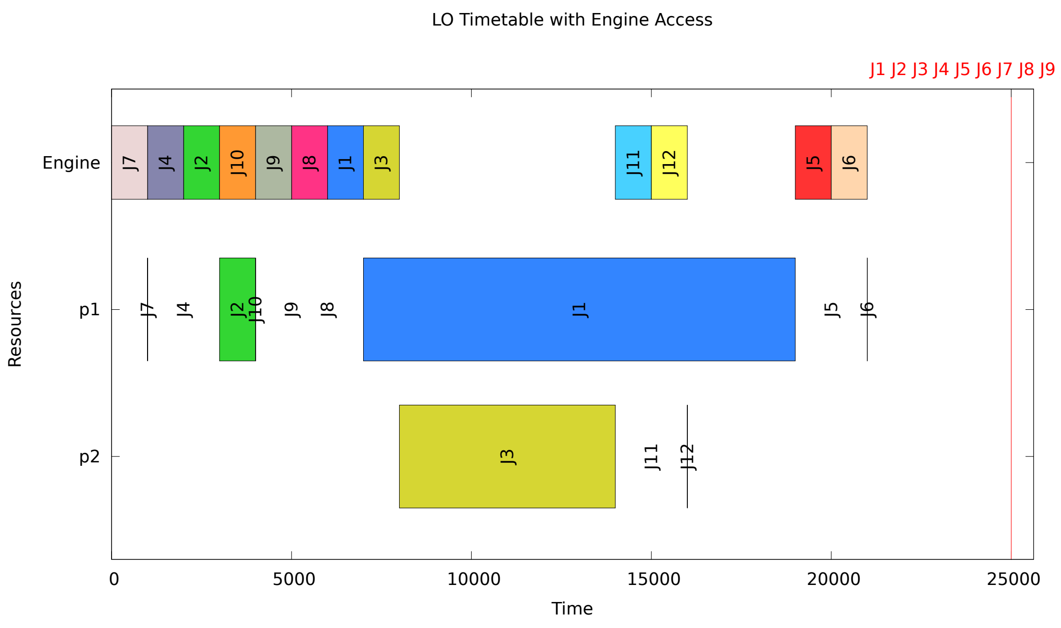
On the other hand, in the mixed-criticality case we consider the two execution modes – normal and emergency – separately. In the normal mode Task 'A' has execution time 6 ms, which is 6 ms less, and we have a load $25/25 = 100$ %, for which a single-core may be sufficient. In the emergency mode the execution time of Task 'A' is again 12 ms, but we drop Task 'B', which saves us $6 + 4$=10 ms and leads to the load of $21/25$=84 %, which again may be doable on a single core, as evaluated by the tool. Thus mixed criticality can be used to use the cores more economically.

The tool generates the schedules for the ordinary graph and for the mixed-critical one, as shown in Figure 6. Figure 7 shows the Gantt charts of executing the two variants of the schedule on the LEON4 board.

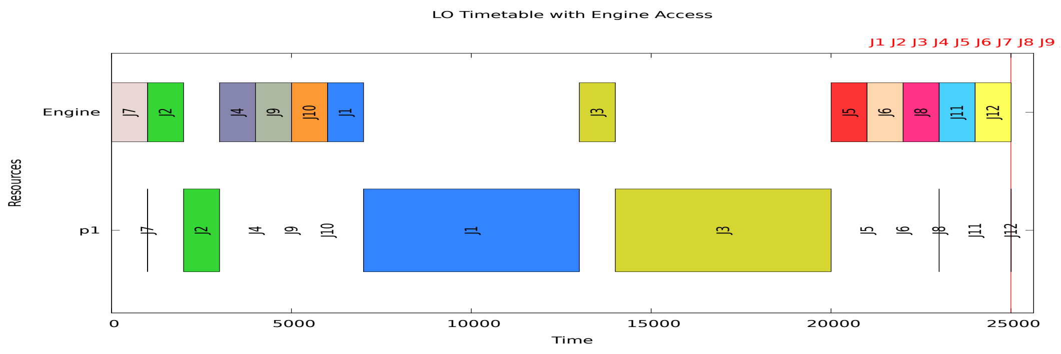
In every Gantt chart the first line shows the execution of the BIP Engine on 'Core 0'. One may wonder why a whole core would have to be reserved to a runtime environment. This is due to lack of support of preemption in current BIP RTE. Moreover, it should be noted that in many-core systems (or graphical accelerators), this is justifiable, as in practice there are plenty of cores available – *e.g.,* 16 per shared-memory cluster in [26] – and no preemption is allowed. On the contrary, for a multi-core system such as LEON4, which supports preemption, in future work we intend to interleave high-priority engine control operations with lower-priority execution of the actual task schedule. Note that the engine thread executes also the BIP components responsible for control operations, such as the task controllers, the MoC controller and the online scheduler.

Recall that the shared resource on which interference-modeling is currently supported by the tools is the engine. As we see in the Gantt chart, every task execution is prefixed and suffixed by two δ-accesses to Core 0 (in fact, two δ's at the prefix and two δ's at the suffix). In the ordinary schedule, Task 'split' and Task 'A' are mapped to Core 1 and Task 'B' to Core 2.

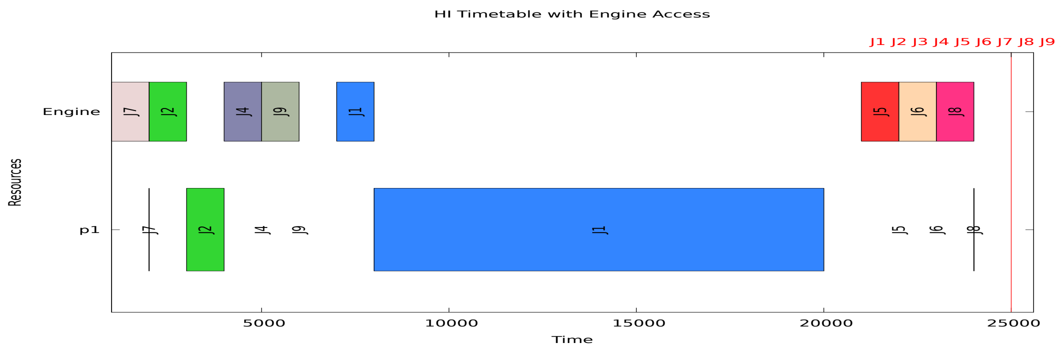
The platform-measurement charts show two periods, one in normal and one in emergency mode. The offline scheduler 'ordinary' solution assumes the overall worst-case, whereas the mixed critical (MC) solu-



(a) Ordinary Schedule

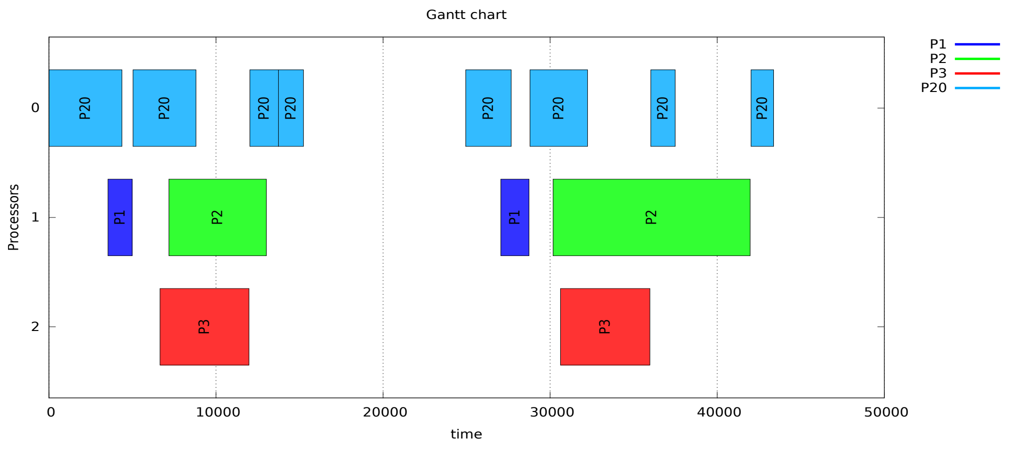


(b) MC Schedule: Normal Mode

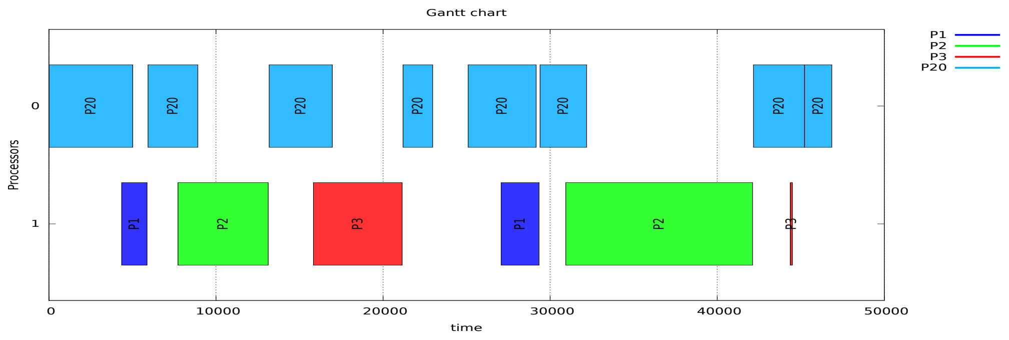


(c) MC Schedule: Emergency Mode

Figure 6: Three-Task Example: Offline-Scheduler Solutions



(a) Ordinary Execution Traces (No mode switch in the second period)



(b) Mixed-critical Execution Traces (Dropping J_3 in the second period)

Figure 7: Three-Task Example: Platform Execution Traces

tion distinguishes two modes. Comparing the corresponding segments of Gantt charts of the solutions and measurements we see a match, though not a perfect one. This is because the offline scheduler output is not yet supported as input to the online scheduler in general. We see that in the emergency mode MC case the offline scheduler drops task 'B' altogether, whereas the online scheduler still makes a short execution of Task 'B' in degraded mode.

Because of current temporary lack of tool integration we had to do manual modifications in the concurrency model that was automatically generated from FPPN, in order to ensure that the online behavior matches the offline solution. Note that a possibility for the user to refine the behavioral model by such modifications is itself an attractive design-flow property. We made modifications in the mixed-criticality variant of the design, in order to introduce the switch from normal to emergency mode. We ensure that if Task 'A' executes beyond its normal-mode execution time then Task 'B' is executed in degraded mode. These modifications are shown in Figure 8.

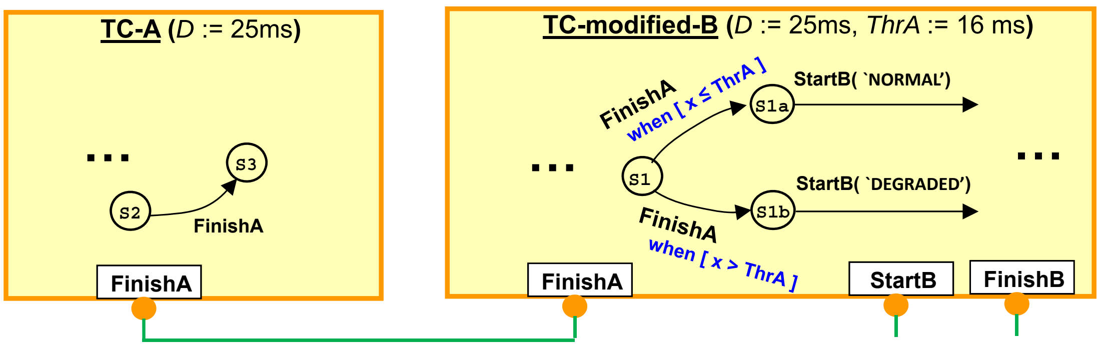


Figure 8: Three-Task Example: Manual Modification Introducing a Mode Switch

We have modified the structure of the TC for Task 'B', which originally was as shown in Figure 2, by introducing a new transition between the 'Arrive' and 'Start' for Task 'B'. This transition is synchronized with 'FinishA' transition in the TC of Task 'A'. We check the value of clock 'x' which measures the time since the begin of the current period. If this value is larger than a certain threshold $ThrA$ then 'B' is executed in degraded mode.

4 Offline Scheduling Algorithm

4.1 Algorithm Description

In this section we zoom into a particular tool in our design flow – the offline scheduler. We give some basic idea on the scheduling problem, the δ-interference model and the scheduling algorithm. Finally we show schedulability-evaluation experiments with random benchmarks.

A scheduling problem instance consists of a DAG task graph obtained automatically from a MoC; we have seen examples in Figure 5. The nodes, J_i are obtained from tasks and are annotated by parameters (A_i, D_i, χ_i, C_i), where $[A_i, D_i]$ give the job scheduling window (between arrival and deadline relative to the hyperperiod), χ_i gives the job criticality level ('LO' or 'HI') and C_i is a vector that gives the execution time in the normal and emergency modes. The problem instance also includes the selected number of cores (not counting the engine core) and some information on interference, currently we only take the value of δ, whose meaning is interference at the start of each job. The δ-interference model is shown at the left side of Figure 9.

This model can be described by a hypothesis that we have a global system controller *i.e.,* the automaton obtained from a combination of all concurrency-model automata present in the system. Lets call it by abuse of terminology the 'engine'. The engine controller makes discrete transitions (control steps), each step costing execution time δ at the control core. At certain steps the engine spawns a job on a compute core taken from a pool of cores. For this, an idle core is selected and reserved at the beginning of the step.

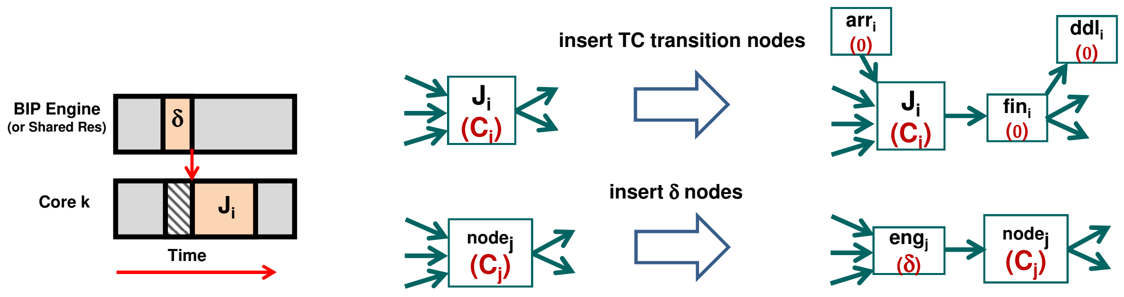


Figure 9: Engine (‘Delta’) Interference and its Modeling in the Task Graph

The steps that do not spawn any computations are modeled as steps that spawn a job with zero execution time. The engine does not make execution steps all the time, for some time intervals it may decide to do idle-waiting.

As we have seen in Figure 2, a periodic controller can be modeled as a system component that, for a given task lets the engine make four subsequent steps corresponding to the following *transitions*: arrival, start, finish and deadline check. The real computation job is, in fact, triggered by the ‘start’ step, the other steps do not trigger any computations. Therefore, as shown in Figure 9, to model periodic jobs we insert three corresponding zero-execution time ‘satellite’ jobs. The arrival job becomes an extra predecessor of the original job, the finish job becomes the new successor after which all the original successors follow, where we also introduce a new successor – the deadline-check job. Now it should become clear why in our example in the previous section every job execution is prefixed and suffixed by two δ-steps. To model the part of the job that is executed on the engine Figure 9 shows the second graph transformation, which inserts a δ-predecessor at every job. Except for the execution time, the newly inserted ‘satellites’ get the same characteristics (*i.e.,* scheduling window and criticality) as the original job.

The scheduling algorithm is applied in our design flow to a graph obtained from the original MoC after it has been post-processed by the two graph transformations defined above. The algorithm generates the two schedules for the two execution modes. These schedules act online as tables for time-triggered execution, see *e.g.,* Figures 6(b) and 6(c).

First the normal-mode table is generated. This is done using global fixed-priority simulation that takes precedences into account. This algorithm is also known as *list-scheduling*. As mentioned before, we assume non-preemptive scheduling. The algorithm has been adapted to take into account two types of resources: a single control core and a pool of compute cores. In order to execute, every job first needs one instance of both resource types for time duration δ to execute its δ-predecessor and then during its own time duration C_i continues running on the compute core only, whereas the control core is available to spawn another job. The algorithm maintains a *list of ready jobs* (and hence its name). As soon as the control core and a compute core become available to start another job the algorithm picks the *highest-priority* ready job and starts its simulated execution. A job is considered *ready* to execute if two conditions hold. Firstly, the job scheduling window $[A_i, D_i]$ must be already begun, *i.e.,* for the current simulated time t we have $t \geq A_i$. Secondly, all DAG predecessors of the job (if any) must be finished.

The *priorities* for selecting the next job to be scheduled are obtained from an earliest-ALAP-first (ALAP means ‘as late as possible’) fixed priority table. Job’s ALAP time gives the latest time when it may complete its execution such that neither that job nor any of its transitive DAG successors will miss the deadlines. ALAP times are computed recursively, from the sinks to the sources, taking into account the execution times. Before ALAP times are calculated, the deadlines of the HI jobs are reduced by the value of their execution time uncertainty, *i.e.,* the difference between their execution times in the emergency and normal modes. Those are the effective deadlines that should be met to avoid missing the deadlines if a switch to the emergency mode occurs. These ‘effective’ deadlines give a HI job higher priority with respect to a LO job whose nominal deadline is the same. It is due to this reason that in our Three-Task example, in its mixed-criticality variant, (see Fig. 5, 6(a)), the HI Task ‘A’ is scheduled before the LO Task ‘B’.

The emergency mode table is calculated such that at any moment of time a switch from normal to emergency mode may take place such that the HI jobs may continue without being preempted or migrated in the middle of execution to another core. To this end, the schedule start times in the normal mode are regarded as job arrival times in the emergency mode. Further, in this mode, we simulate only the HI jobs (while the LO jobs are dropped) taking into consideration only HI-to-HI job precedences while keeping the same job-to-core mapping and the same relative order of HI-job execution as in the normal mode. When a job deadline miss is detected in any of the two modes the algorithm fails.

Since our variant of list scheduling algorithm does not use dynamic priority tables and the static table can be obtained by simple topological sort algorithm, the complexity of our algorithm is the same as the one of list scheduling. Our implementation of this algorithm according to [18] has complexity:

$$O(V(\log V + M) + E)$$

where V, E is the number of nodes, edges respectively and M is the number of processors.

4.2 Experiments

We have performed experiments of measuring the success rate of the algorithm for random generated ordinary and mixed-critical benchmarks that have different level of *normalized stress*, which is a peak resource utilization metric – see [24, 18] – ranging from 0 to 100 %. For mixed-criticality experiments, the stress for both modes of execution was maintained equal. We assumed instances with 10 jobs and no precedence constraints.

Experiments for three different values of ρ were made: 0.1, 0.5 and 0.8, where ρ is the ratio between the stress due to δ-jobs only and the stress due to all jobs. As expected, the mixed-criticality instances are much harder to solve than ordinary ones by the same algorithm. In future work it will be interesting to implement an exact algorithm, *e.g.,* using SMT solvers, to evaluate the optimality of our algorithm experimentally.

We noticed that, counter-intuitively, no significant sensitivity to the value of ρ was detected. A possible reason is that ρ appears to have only a weak connection to the ratio between δ and average task execution time. In future work a better load-related metric for the proportion of interference in the total workload will be investigated.

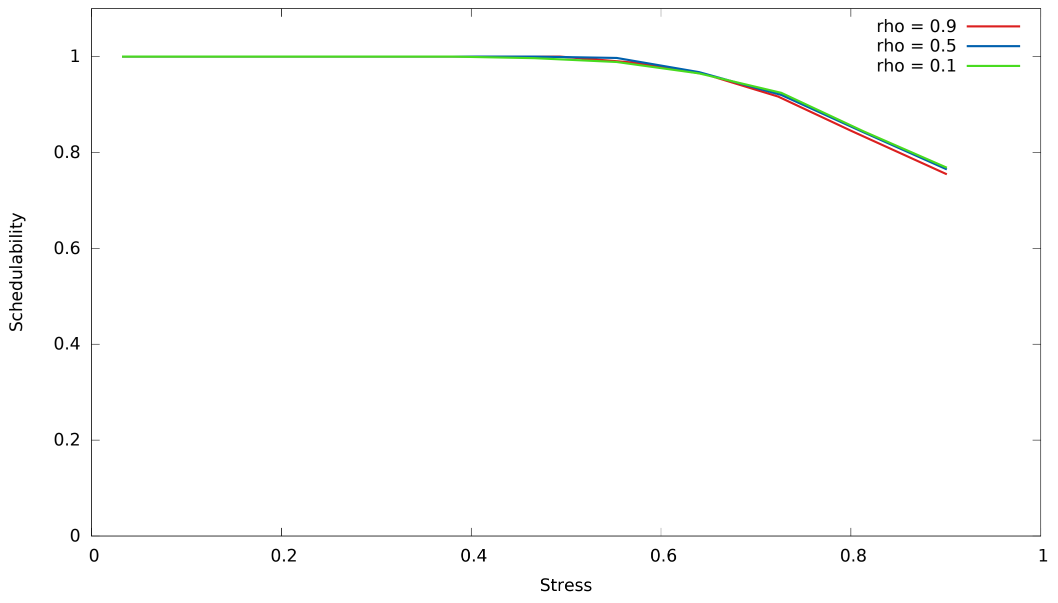
4.3 Related Work

Different previous works address related problems, some of them are discussed in this subsection. Reference [20] is an extension of [17] which calculates STTM tables for multi-rate synchronous mixed-critical systems. This work is restricted to uniprocessor platforms. Task automata verification [11] has unprecedented expressive power, but may be subject to scalability issues for industrial-scale systems unless it is applied with some approximations and abstractions. The superblock approach [15] is a well-recognized technique to address even fine-grain interference and it has been further developed in related work. However a problem remains still open, see [3], concerning calibration of fine-grain analysis techniques to typical processor and bus architectures that are deeply pipelined and have other performance optimization features.

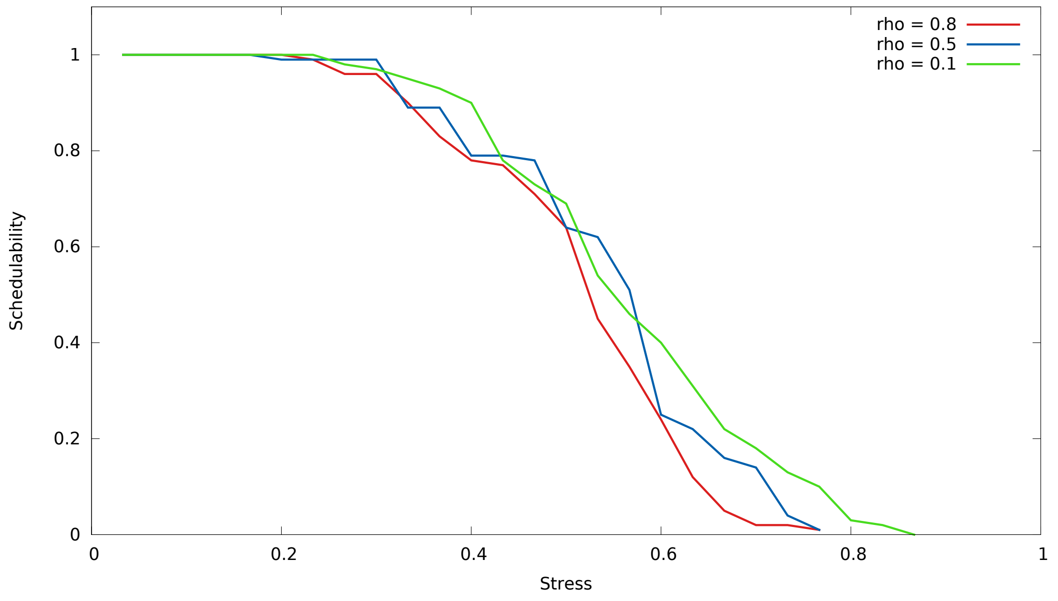
The semantics of synchronous systems is relaxed even further compared to [20] in the direction of functionally equivalent asynchronous pipelined execution with self-timed synchronization, following the philosophy of Kahn process networks (KPN). The goal was to support embedded signal processing and multimedia stream-processing in general. Rich liveness, memory boundedness, code generation and real-time throughput/latency analysis theories have been developed for these models, termed *data-flow MoCs*. An interesting survey for expressive real-time analyses is given in [7]. Paper [16] studies optimal handling of coarse-grained interference in simple variants of such MoCs.

In data-flow MoCs, *classical* concepts such as release times, periods and deadlines are replaced by self-timed iterations, long-run throughput and multi-iteration latency bounds. The 'FPPN' MoC, adopted in our flow, can be seen a data-flow MoC which, in a way, 'attempts' to reconcile itself to classical concepts. Also our work extends the data-flow-related MoCs to mixed criticality.

Only a few scheduling techniques mentioned above are integrated in software engineering toolchains that have both real-time scheduling and code generation. First of all, ADA programming language and its Ravenscar profile are de facto standards for mono-core hard real-time systems. They integrate the most



(a) Ordinary Benchmarks



(b) Mixed-critical Benchmarks

Figure 10: Schedulability Results for Random Benchmarks

trusted and safe multi-task programming and scheduling techniques for tasks that have no explicit precedence constraints. For the case of distributed systems that work was extended to multiple mono-core platforms or partitions communicating via bus and network protocols in the context of TASTE design flow [23]. This work requires extension in order to treat multi-core system nodes and precedence-constrained task models such as FPPN.

Prelude design flow [6] represents an ongoing work on scheduling and deployment of multi-rate synchronous systems defined with more expressive ('synchronous-language') semantics than the ones assumed in our flow and in [20]. It should be noted that the price to be paid for higher expressive power is that it becomes much less obvious how to generate a semantics-preserving task-graph or data-flow MoC model in this case that could provide an input to a precedence-constrained scheduling tool.

A variant of superblock approach was implemented in DOL-BIP-Critical design flow [8]. Task-automata verification is integrated into Times and UPPAAL tools [10]. CompSoC design flow [22] deploys applications based on scheduling algorithms for data-flow MoCs.

Our previous works [24] and [18] present more elaborate techniques than those presented here for optimizing priority and time-triggered tables for mixed criticality, but they assume preemption and do not yet consider interference. In future work we intend to inherit more elements of those techniques into presented scheduling tool. Integrating these techniques directly into our design flow will be considered after we extend our BIP preemption modeling techniques from simulation [13] to real-time deployment.

5 Conclusions and Future Work

In this paper we have proposed a scheduling algorithm and a work-in-progress design flow for timing-critical multi-core applications, taking into account coarse-grained interference, using the interference from the controlling run-time environment as an example. In our design flow we demonstrate the concept of using task automata as concurrency language, which can be used to program the custom resource managers, such as mixed-criticality ones. In future work we plan to introduce the missing features into our design flow (especially, the runtime environment to support task migration and dropping). We also plan to extend our interference models to other resources (*e.g.,* buses and peripherals) and to more general task controllers and models of computation. We also have started a research activity to investigate proper integration of our design approach with a distributed-system design flow TASTE.

References

[1] M. Wirsing, M. M. Hölzl, M. Tribastone, and F. Zambonelli, "ASCENS: engineering autonomic service-component ensembles," in *FMCO'11*, pp. 1–24, 2011. 1, 2.2

[2] S. Chaki and D. Kyle, "DMPL: Programming and verifying distributed mixed-synchrony and mixed-critical software," tech. rep., Carnegie Mellon University, 2016. 1, 2.2, 3.1

[3] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm, "Impact of resource sharing on performance and performance prediction: A survey," in *CONCUR*, vol. 8052 of *Lecture Notes in Computer Science*, pp. 25–43, Springer, 2013. 1, 4.3

[4] P. Poplavko, D. Socci, P. Bourgos, S. Bensalem, and M. Bozga, "Models for deterministic execution of real-time multiprocessor applications," in *DATE'15*, 2015. 2.1, 3.1

[5] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. 2.1

[6] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti, "Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset," in *RTNS*, 2011. 2.1, 3.1, 4.3

[7] S. Stuijk, M. Geilen, B. D. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *SAMOS'11*, IEEE, 2011. 2.1, 3.1, 4.3

[8] G. Giannopoulou, P. Poplavko, D. Socci, P. Huang, N. Stoimenov, P. Bourgos, L. Thiele, M. Bozga, S. Bensalem, S. Girbal, M. Faugere, R. Soulat, and B. D. de Dinechin, "DOL-BIP-critical: A tool chain for rigorous design and implementation of mixed-criticality multi-core systems," Tech. Rep. 363, ETH Zurich, Laboratory TIK, Apr 2016. 2.1, 2.2, 2.4, 3.1, 3.2, 4.3

[9] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "A timed-automata based middleware for time-critical multicore applications," in *Proc. SEUS'15*, IEEE, 2015. 2.2, 3.1

[10] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: A tool for modelling and implementation of embedded systems," in *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pp. 460–464, Springer, 2002. 2.2, 3.1, 4.3

[11] E. Fersman, P. Krcl, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability.," *Information and Computation*, vol. 205, no. 8, pp. 1149–1172, 2007. 2.2, 3.1, 4.3

[12] T. Abdellatif, J. Combaz, and J. Sifakis, "Model-based implementation of real-time applications," in *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, ACM, 2010. 2.2

[13] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Modeling mixed-critical systems in real-time BIP," in *ReTiMiCs'2013*, 2013. 2.2, 2.5, 3.2, 4.3

[14] S. Bensalem, M. Bozga, J. Combaz, and A. Triki, "Rigorous system design flow for autonomous systems," in *ISoLA'14*, pp. 184–198, 2014. 2.2

[15] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha, "Coscheduling of CPU and I/O transactions in cots-based embedded systems," in *RTSS'08*, pp. 221–231, 2008. 2.4, 3.1, 4.3

[16] S. Sriram and E. A. Lee, "Determining the order of processor transactions in statically scheduled multiprocessors," *VLSI Signal Processing*, vol. 15, no. 3, pp. 207–220, 1997. 2.4, 3.1, 4.3

[17] S. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *RTSS '11*, pp. 3–12, IEEE, 2011. 2.5, 3.1, 4.3

[18] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Time-triggered mixed-critical scheduler on single- and multi-processor platforms (revised version)," technical report TR-2015-8, Verimag, 2015. 2.5, 3.1, 4.1, 4.2, 4.3

[19] A. Zerzelidis and A. J. Wellings, "A framework for flexible scheduling in the RTSJ," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 1, 2010. 3.1

[20] S. Baruah, "Semantics-preserving implementation of multirate mixed-criticality synchronous programs," in *RTNS'12*, pp. 11–19, ACM, 2012. 3.1, 4.3

[21] G. De Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994. 3.1

[22] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "Compsoc: A template for composable and predictable multi-processor system on chips," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 1, p. 2, 2009. 3.1, 4.3

[23] M. Perrotin, E. Conquet, P. Dissaux, T. Tsiodras, and J. Hugues, "The TASTE toolset: turning human designed heterogeneous systems into computer built homogeneous software.," in *ERTSS'10*, 2010. 3.1, 4.3

[24] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Multiprocessor scheduling of precedence-constrained mixed-critical jobs," in *ISORC'15*, pp. 198–207, IEEE, 2015. 3.1, 4.2, 4.3

[25] H. Shah, A. Coombes, A. Raabe, K. Huang, and A. Knoll, "Measurement based wcet analysis for multi-core architectures," in *RTNS '14*, ACM, 2014. 3.2

[26] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, “Time-critical computing on a single-chip massively parallel processor,” in *DATE’14*, EDAA, 2014. 3.3