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

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

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

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

VERIMAG

Abstract. Those autonomic concurrent systems which are timing-critical and compute intensive need special resource managers in order to ensure adaptation to unexpected situations in terms of compute resources. 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 a representation in this language and combine the result with a resource manager into a joint software design used to deploy the given system on the target platform. In this context, we discuss our work in progress on a scheduler that aims to handle the interference in mixed-critical applications by controlling it at the task level.

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

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

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

be *autonomic* [26], *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) [7]" that coordinate with the leader UAV within strict time bounds to avoid mutual collision. Such systems should not only be correctly specified but also *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' [2], *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 and illustrates it via a small synthetic application example. Section 4 gives a basic explanation of the scheduling algorithm and discusses the results. 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) [18].

In the figure we see (1) tasks, *e.g.,* 'HighFreqBCP', *etc.*, annotated by periods, (2) inter-task channels, *e.g.,* between 'DopplerConfig' and 'SensorInput',

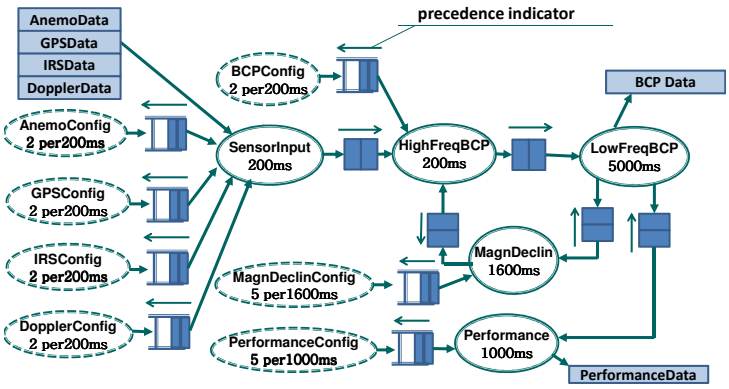


Fig. 1. Application modeled in a MoC: Flight Management System in FPPN

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 keep the slots for input and output data available during the whole interval between the task arrival and the deadline. As a MoC, FPPN should define the partial ordering of execution and interaction 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 name just a few: MRDF (multi-rate data-flow, often named SDF – Synchronous Dataflow) [14], Prelude [8], SADF (scenario-aware data-flow) [25] and DOL-Critical [11].

2.2 Resource Managers and Concurrency Language

An important property of autonomic embedded systems is their ability to adapt themselves to unexpected phenomena [26]. 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 [7]. A resource manager is an integral part of an *online scheduler i.e.,* a middleware that implements a customized online scheduling policy.

Unfortunately, there is a considerable semantical gap between the online schedulers and the middlewares that implement MoCs, even though both define software concurrency behavior. We aim at a common approach that can ensure

consolidation, by representing both types of middleware 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) [23].

We believe that for autonomic timing-critical systems a proper choice of concurrency language is a combination of procedural languages and *task automata*. The latter are timed automata extended with tasks [3, 10]. Timed-automata languages in general are known to be convenient means to specify resource managers, such as QoS [1] and mixed criticality [20].

In our design flow the concurrency language is BIP. Under 'BIP' we mean in fact its 'real-time dialect' [1], designed to express networks of connected timed automata components. In [6] BIP was demonstrated relevant for distributed autonomic systems. In [11] 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, 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 an ideal case, the compilation would be configured by a user-defined set of grammar rules for automatic translation of the user's 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 component.

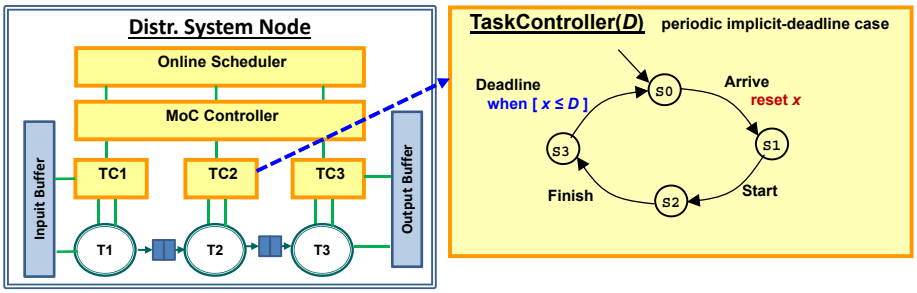


Fig. 2. Concurrency Language Representation of a Timing-critical Application

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 MoC semantics, we refer to that component as MoC controller. One can also introduce an automaton that would further restrict the ordering and the timing of task executions – the online scheduler. This component would impose user-programmed scheduling policy. Note that automata can be hierarchical, *i.e.,* they can represent a composition of 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 as 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 in our design flow the given task controller is both time- and event-driven, as the tasks arrive periodically (in a time-driven way) but start when the MoC controller would enable the 'Start' interaction, thus indicating that the task predecessors have finished (in an event-driven way).

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 cores run in parallel. Interference appears when cores await response from resources that are in use by other cores.

The concerned resources can be either hardware or protected logical (software) resources. Shared hardware resources that can cause interference are 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.

In a design flow for mono-core systems the 'worst-case execution time (WCET) analysis' conveniently precedes 'schedulability analysis', as the task WCETs do not depend on the schedule. On the contrary, in a multi-core system, because of

interference task execution delay may significantly change depending on which tasks are scheduled on the other cores. Therefore part of task WCET analysis may have to be re-done when schedules are analysed, which is a major obstacle in the design of timing-critical systems based on multi-cores [2].

Luckily, coarse-grain interference can be *controlled* by scheduling the superblocks in a way that the resource conflicts are eliminated. To achieve this, in a 'controlled' schedule superblocks are executed sequentially. At the same time, uncontrollable fine-grained interference can be for as much as possible transformed into coarse-grained one by 'concentrating' the resource-access intensive parts of source code together into coarse-grained superblocks, which can be controlled. The controlled interference approach is well-known in the literature. For example, in [24], coarse-grained blocks of accesses to global bus are considered as special sub-tasks which are scheduled in an optimal static order.

In our scheduling algorithm we assume 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 superblocks. In addition, though different resources (*e.g.,* different FPU's and different memory banks) can be accessed independently and though different superblocks can have different timing costs, we make a simplifying assumption that there is only one shared resource and the duration of all superblocks is the same, we denote it δ. In a way, we consider superblocks as instances of a special task whose WCET is δ.

A particular form of such interference that manifests itself in our design approach is called *engine interference* [11]. In our concurrency model, governed by automata, one can distinguish task-concurrency control operations which correspond to discrete transitions of the automata components that constitute the system. All discrete transitions are coordinated via a single control thread called the *engine*. Suppose that δ is the worst-case time to handle one discrete transition. Then the runtime overhead of task concurrency control operations can be conveniently modeled as interference between superblocks of size δ. In addition to the necessary accesses to the engine needed to coordinate task concurrency, each coarse-grained block of accesses to any resource can be, in principle, delegated to the engine as well. For this, the compiler would have to represent each superblock as a discrete transition or, if it is large, as a sequence of transitions. Therefore, the engine interference can be generalized to subsume other forms of coarse-grained interference.

In the present work, engine interference is the only form of interference that is automatically modelled by our tools. Compared to [11], the novelty is that in the present work we control this form of interference in the scheduling. Our scheduling algorithm assumes that there is one shared resource, and we model the engine as such. Further, it assumes that all superblocks are explicitly represented by special tasks with equal WCET δ, and we model the task-controller transitions as such.

To manage the remaining fine-grained interference we advocate the *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 a fine-grain interference analysis (which would compute the WCET margins) 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.

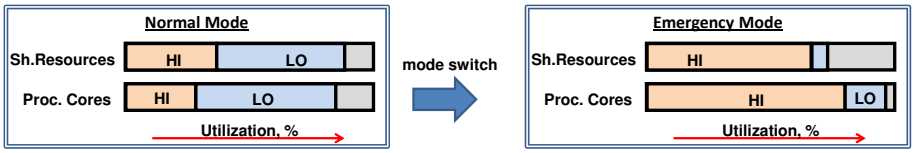


Fig. 3. Mixed-criticality Resource Management

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 is implemented in concurrency language as part of the 'online scheduler' automaton component [20].

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

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 [5].

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 [5, 22], which is a generalization to mixed-criticality scheduling. In this policy, the normal and the emergency modes each 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 [22] 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 timing-critical systems. This is especially the case for multiprocessor and distributed 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 representing the scheduling policy in a 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 [23, 11]. 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 [11, 27].

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, [4, 18], conveniently presentable to a list-scheduling heuristic, which, in turn, has reputation of reasonable performance for comparable instruction-level scheduling problems [13]. Moreover, we present a design flow where applications can be both programmed and scheduled. Other design flows that have this property, *e.g.,* [3,

7, 8, 11, 12, 16], do not take into consideration all the aspects we do but in return offer other features, *e.g.,* distributed-system/network support or expressive power. We compare to [11] in the next section. Related scheduling techniques [4, 5, 10, 15, 21, 22, 24, 25] also have some restrictions, while in return offering important theoretical properties and features. We discuss related work further in extended version of this paper [17].

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 job nodes and precedence 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 running on the platform, the binary executable encounters interferences, as discussed in Section 2.4. Handling interference requires a feedback loop from the binary executable to the offline scheduler tool. Next to the worst-case execution times (WCET's) of tasks, the worst-case execution time δ of coarse-grained superblocks should be obtained 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 [20], it cannot yet be executed in real-time mode. This is subject of future work. A justification for considering non-preemption is frequent lack of support of preemption in 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 [11] 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

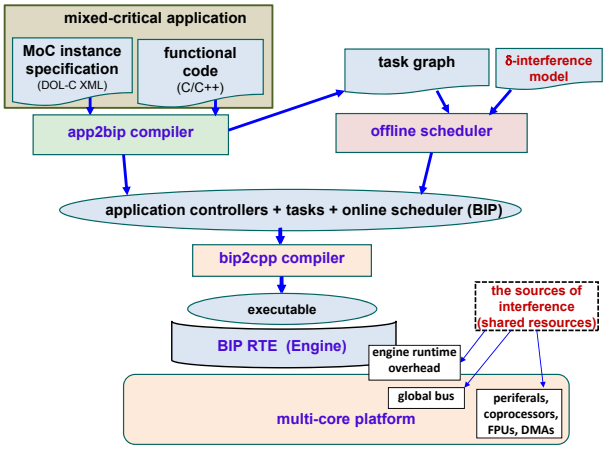


Fig. 4. Work-in-progress Design Flow

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 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 a version of the 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 [19], 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 [11] 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 platform with four cores implemented on FPGA, using 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.

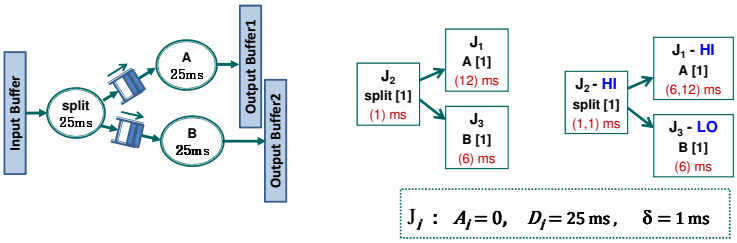


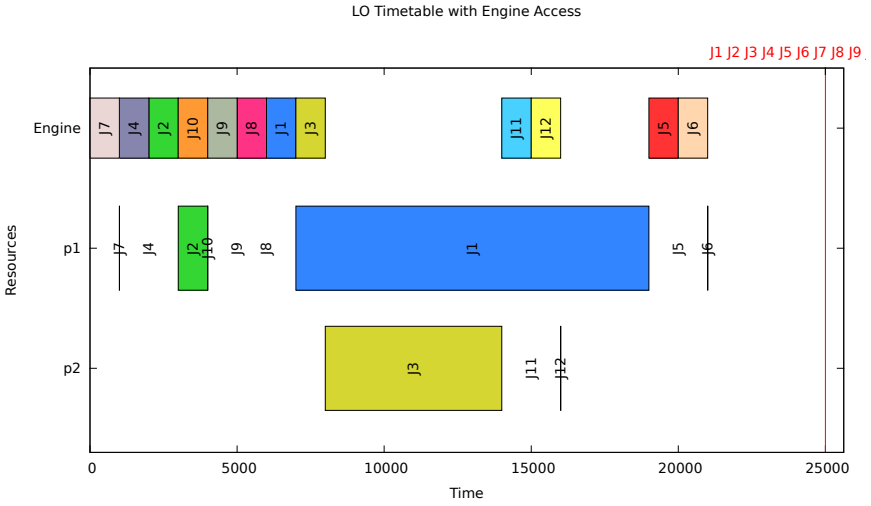
Fig. 5. Three-Task Example: MoC (left), Ordinary Task Graph (middle) and Mixed-critical Task Graph

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.

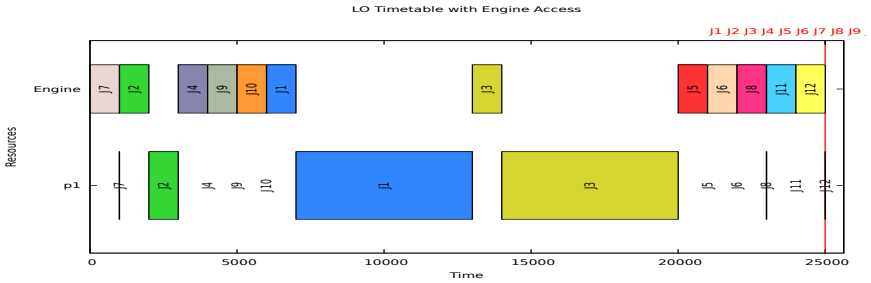
The engine runtime overhead (as it will become clear later) constitutes 4δ = 4 ms per task (in total 12 ms). Therefore, when assuming ordinary execution times this example cannot run on a single core, as the total execution time amounts to $12+1+12+6$=31 ms, which is larger than the 25 ms deadline. The 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 cores 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. Thus, mixed criticality can help to use the cores more economically.

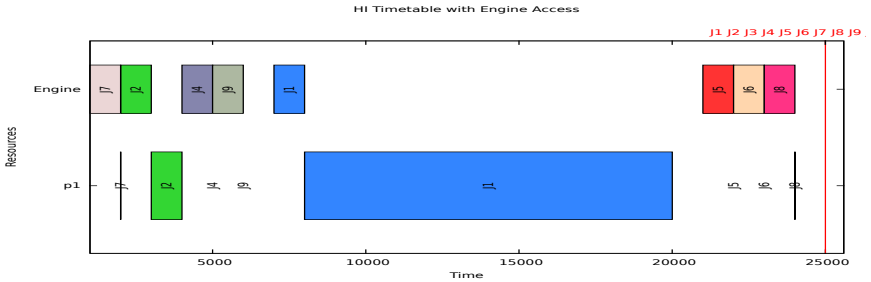
The tool generates the schedules for the ordinary graph and for the mixed-critical one, as shown in Figure 6. Figure 7 shows the Gantt charts of executing the two variants of the schedule on the LEON4 board.



(a) Ordinary Schedule

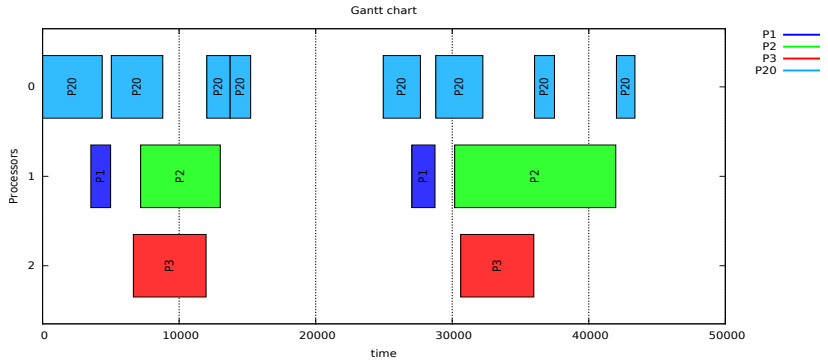


(b) MC Schedule: Normal Mode

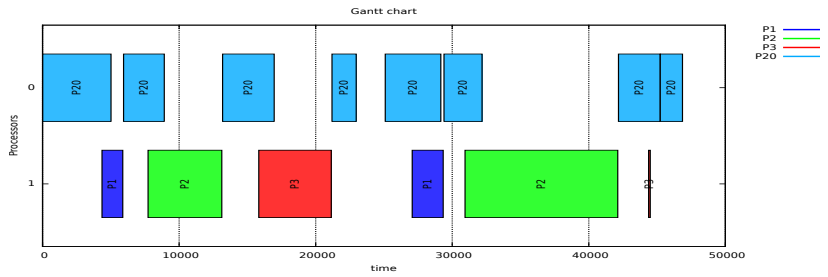


(c) MC Schedule: Emergency Mode

Fig. 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)

Fig. 7. Three-Task Example: Platform Execution Traces

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 [9] – and no preemption is allowed. On the contrary, a platform such as LEON4 supports preemption and does not assume one thread per core. For such platforms in future work we intend to interleave high-priority engine control thread with a lower-priority task-execution thread on Core 0. 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 Figure 6, every task execution is prefixed and suffixed by two δ-accesses to Core 0. In the ordinary schedule, Task 'split' and Task 'A' are mapped to Core 1 and Task 'B' to Core 2.

The platform-measurement charts in Figure 7 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) solution 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. 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.

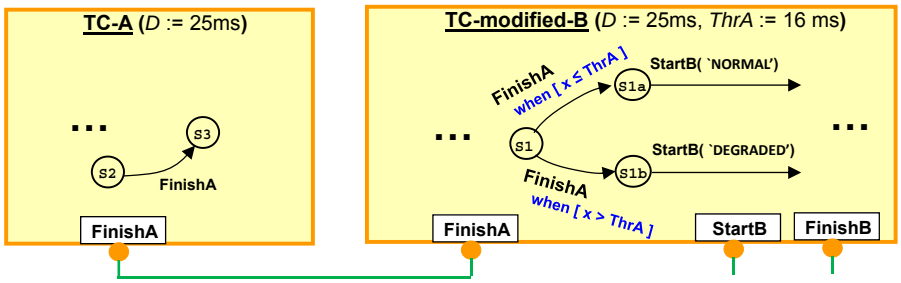


Fig. 8. Three-Task Example: Manual Modification Introducing a Mode Switch

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

For space reasons, here we just summarize the offline scheduling algorithm and its results, more detailed description and related work analysis can be found in extended version of this paper [17].

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 BIP engine discrete-transition execution time δ to model interference.

The goal of the scheduling algorithm is to generate two time triggered scheduling tables: for normal mode and emergency-mode. These schedules act online as tables for time-triggered execution. For example, Figures 6(b) and 6(c) are actually graphical representation of these tables for the given example.

The scheduling tool first transforms the task graph by inserting special 'satellite' jobs that model engine interference due to periodic task controller. Then the normal-mode table is generated. This is done using list 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 needs availability of one instance of both resource types to execute for δ time and immediately it continues to execute only on the compute core for WCET time. In normal mode, the *priorities* for selecting the next job to be scheduled are obtained from fixed priority table that favors jobs that have HI criticality and high difference between execution times in emergency and normal mode. Also we favor jobs that have small deadline themselves or in their successors. The results of list schedule simulation with normal job execution times are stored in normal-mode table.

The emergency mode table is calculated, again by list scheduler, but now with emergency execution times and only for HI jobs and HI-to-HI job precedences. We ensure that at any moment a switch from normal to emergency mode may take place while the HI jobs that are running at the moment of the switch may continue running on the same cores. To this end, the schedule start times in the normal mode are regarded as job arrival times in the emergency mode, whereas we enforce the same core mapping and relative job execution order as in the normal mode.

Our algorithm has the same (almost linear) algorithmic complexity as unmodified list scheduling, since it adds constant amount of additional computation for each job and precedence edge. Random benchmarks [17] confirm that for the same level of computational workload mixed critical problems are significantly harder to solve. At the same time we did not see significant sensitivity to the workload component given by interference, which possibly means that we need to improve the employed interference evaluation metric.

In future work we intend to investigate how to improve non-preemptive scheduler for better support of mixed criticality. For reference we consider to implement exact algorithm with exhaustive search. We intend to replace list scheduling by topological permutation scheduling as it is a more powerful offline global fixed-priority heuristic for the case where there is no preemption and jobs have non-zero arrival times [13]. Also, in our previous works [21] and [22] for preemptive case we realized more elaborate techniques than those in the current algorithm for optimizing for mixed-criticality, we will investigate how to port them to non-preemptive case. Integrating them directly into our design flow will be considered after we extend our BIP framework for support of preemption.

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, dropping and migration). 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.

References

1. Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: Proceedings of the tenth ACM international conference on Embedded software. EMSOFT '10, ACM (2010)
2. Abel, A., Benz, F., Doerfert, J., Dörr, B., Hahn, S., Haupenthal, F., Jacobs, M., Moin, A.H., Reineke, J., Schommer, B., Wilhelm, R.: Impact of resource sharing on performance and performance prediction: A survey. In: CONCUR. Lecture Notes in Computer Science, vol. 8052, pp. 25–43. Springer (2013)
3. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: 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)
4. Baruah, S.: Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In: RTNS'12. pp. 11–19. ACM (2012)
5. Baruah, S., Fohler, G.: Certification-cognizant time-triggered scheduling of mixed-criticality systems. In: RTSS '11. pp. 3–12. IEEE (2011)
6. Bensalem, S., Bozga, M., Combaz, J., Triki, A.: Rigorous system design flow for autonomous systems. In: ISoLA'14. pp. 184–198 (2014)
7. Chaki, S., Kyle, D.: DMPL: Programming and verifying distributed mixed-synchrony and mixed-critical software. Tech. rep., Carnegie Mellon University (2016), http://www.andrew.cmu.edu/user/schaki/misc/dmpl-extended.pdf
8. Cordovilla, M., Boniol, F., Forget, J., Noulard, E., Pagetti, C.: Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In: RTNS (2011)
9. de Dinechin, B.D., van Amstel, D., Poulhiès, M., Lager, G.: Time-critical computing on a single-chip massively parallel processor. In: DATE'14. EDAA (2014)
10. Fersman, E., Krcl, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)
11. Giannopoulou, G., Poplavko, P., Socci, D., Huang, P., Stoimenov, N., Bourgos, P., Thiele, L., Bozga, M., Bensalem, S., Girbal, S., Faugere, M., Soulat, R., de Dinechin, B.D.: 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)
12. Hansson, A., Goossens, K., Bekooij, M., Huisken, J.: Compsoc: A template for composable and predictable multi-processor system on chips. ACM Transactions on Design Automation of Electronic Systems (TODAES) 14(1), 2 (2009)

13. Heijligers, M.: The Application of Genetic Algorithms to High-Level Synthesis. Ph.D. thesis, Univ. of Eindhoven (1996)
14. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
15. Pellizzoni, R., Bui, B.D., Caccamo, M., Sha, L.: Coscheduling of CPU and I/O transactions in cots-based embedded systems. In: RTSS'08. pp. 221–231 (2008)
16. Perrotin, M., Conquet, E., Dissaux, P., Tsiodras, T., Hugues, J.: The TASTE toolset: turning human designed heterogeneous systems into computer built homogeneous software. In: ERTSS'10 (2010)
17. Poplavko, P., Kahil, R., Socci, D., Bensalem, S., Bozga, M.: Mixed-critical systems design with coarse-grained multi-core interference. Technical Report TR-2016-4, Verimag (2016)
18. Poplavko, P., Socci, D., Bourgos, P., Bensalem, S., Bozga, M.: Models for deterministic execution of real-time multiprocessor applications. In: DATE'15 (2015)
19. Shah, H., Coombes, A., Raabe, A., Huang, K., Knoll, A.: Measurement based wcet analysis for multi-core architectures. In: RTNS '14. ACM (2014)
20. Socci, D., Poplavko, P., Bensalem, S., Bozga, M.: Modeling mixed-critical systems in real-time BIP. In: ReTiMiCs'2013 (2013)
21. Socci, D., Poplavko, P., Bensalem, S., Bozga, M.: Multiprocessor scheduling of precedence-constrained mixed-critical jobs. In: ISORC'15. pp. 198–207. IEEE (2015)
22. Socci, D., Poplavko, P., Bensalem, S., Bozga, M.: Time-triggered mixed-critical scheduler on single- and multi-processor platforms (revised version). Technical Report TR-2015-8, Verimag (2015)
23. Socci, D., Poplavko, P., Bensalem, S., Bozga, M.: A timed-automata based middleware for time-critical multicore applications. In: Proc. SEUS'15. IEEE (2015)
24. Sriram, S., Lee, E.A.: Determining the order of processor transactions in statically scheduled multiprocessors. VLSI Signal Processing 15(3), 207–220 (1997)
25. Stuijk, S., Geilen, M., Theelen, B.D., Basten, T.: Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In: SAMOS'11. IEEE (2011)
26. Wirsing, M., Hölzl, M.M., Tribastone, M., Zambonelli, F.: ASCENS: engineering autonomic service-component ensembles. In: FMCO'11. pp. 1–24 (2011)
27. Zerzelidis, A., Wellings, A.J.: A framework for flexible scheduling in the RTSJ. ACM Trans. Embedded Comput. Syst. 10(1) (2010)