Ensuring Schedulability for Embedded Multi-cores

Technical Note

Peter Poplavko, Verimag

Rany Kahil, Verimag

Dario Socci, Verimag

Saddek Bensalem, Verimag

Marius Bozga, Verimag

December 2017

Abstract

To manage the complexity of concurrent-system design, the applications running in the nodes of distributed systems have to be designed in an appropriate high-level model of computation (MoC). In addition, for systems that are timing-critical and compute-intensive, it may be required to introduce so-called mixed-critical resource managers (dynamic scheduling policies) that adapt system resource usage to critical run-time situations (eg overheating, overload, hardware errors) by giving the highly critical subset of system functions priority over low-critical ones in emergency situations. However, especially for modern platforms -- multi- and many- cores -- it is highly non-trivial to manage resources not only because of their inherent parallelism but also because of ``parasitic'' interference between the cores due to shared hardware resources (buses, FPU's, DMA's, etc). To close the semantical gap between MoCs on one side and resource managers on the other, we compile the MoCs into expressive automata-based language, used to validate and implement a given MoC/resource manager combination. In this context, we present our current work-in-progress on scheduling tools for handling the multi-core interference in mixed-critical applications.

Contents

1 Introduction

In this report 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 *timing-critical*, *i.e.,* subject to real-time constraints. An example of such a system is a "fleet of UAV's (unmanned air vehicles) [1]" 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' [2], *i.e.,* blocking due to conflicts in simultaneous accesses to shared hardware resources, such as buses, FPU's, DMA channels, IO peripherals. A known strategy in embedded multiprocessors is to join as many shared-resource accesses as possible into coarse-grain blocks and to schedule these blocks explicitly as sub-tasks to improve predictability [3, 4]. Coarse-grain interference is a major assumption of our scheduling tool, whereby for simplicity we assume a single shared resource and constant block duration δ. Our scheduling tool support mixed-criticality resource management, an important feature for adaptive embedded systems.

Our work-in-progress design flow offers not only scheduling but also deployment, which should rigorously follow online the schedule calculated offline. The deployment is ensured by a compilation tool-chain starting from a high-level model of computation where the application is specified. Our design flow can be adapted to various application models and online schedulers by exploiting an expressive intermediate language to define the software concurrency.

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 report 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) [5].

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 name just a few: MRDF (multi-rate dataflow, often named SDF – Synchronous Dataflow) [6], Prelude [7], SADF (scenario-aware dataflow) [8] and DOL-Critical [9].

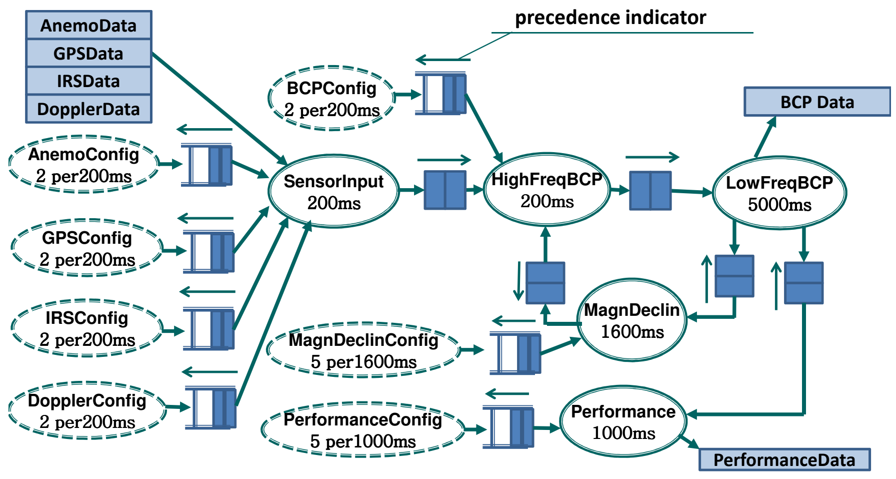


Figure 1: Application modeled in a MoC: Flight Management System in FPPN

2.2 Resource Managers and Concurrency Language

An important property of modern embedded systems is ability to execute autonomously and to adapt itself to unexpected phenomena. 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 scaling, *etc.*. Especially the mixed-criticality approaches are gaining more an more interest and have a high relevance for collective adaptive systems [1]. 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) [10].

We believe that for 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* [11, 12]. Timed-automata languages in general are known to be convenient means to specify resource managers, such as QoS [13] and mixed criticality [14]. In our design flow the concurrency language is BIP. Under 'BIP' we mean in fact its 'real-time dialect' [13], designed to express networks of connected timed automata components. In [9], we extended it 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. In ideal case, this would be ensured by letting the user create a set of grammar rules for automatic translation of his preferred MoC into automata.

2.3 Concurrency-language based Design 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.

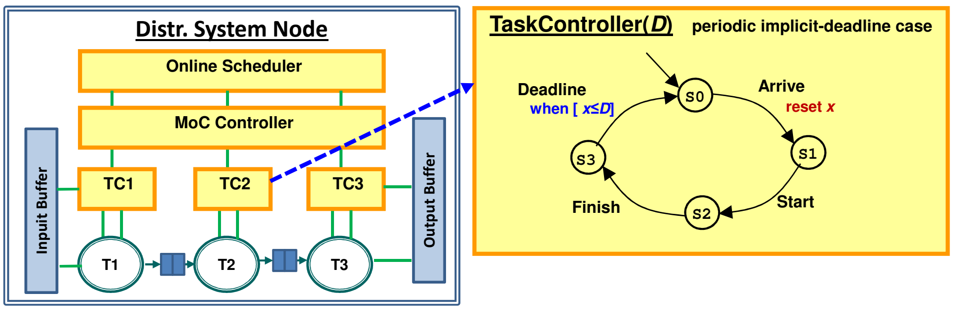


Figure 2: Concurrency-language Representation of an Timing-critical Application

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 'SO' 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 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 Distributed System Aspects

Figure 3 illustrates the schedulability conditions of a timing-critical distributed system.

In the figure we consider a simple single-rate two-node system (sender and receiver) and a timing window of a single system iteration. The iteration window consists of three different sub-windows. The first one is between the arrival (*i.e.,* the release) time and the deadline of the sender tasks. In this window the sender should prepare the output to be sent to the receiver in its buffer. The next sub-window corresponds to the network delay, and the third one is the window for holding the data at the destination node, this window represents the arrival time and the deadline of the tasks at the receiver. Note that subsequent system iteration windows may overlap in time (*i.e.,* pipelined executions, when iteration period is smaller than the iteration window size), and that this model can be generalized to multi-rate system (in which case one iteration would correspond to a hyperperiod) with multiple buffers. Note that in a distributed system different nodes may need to maintain sufficient alignment of their local time models by running a clock synchronization protocol.

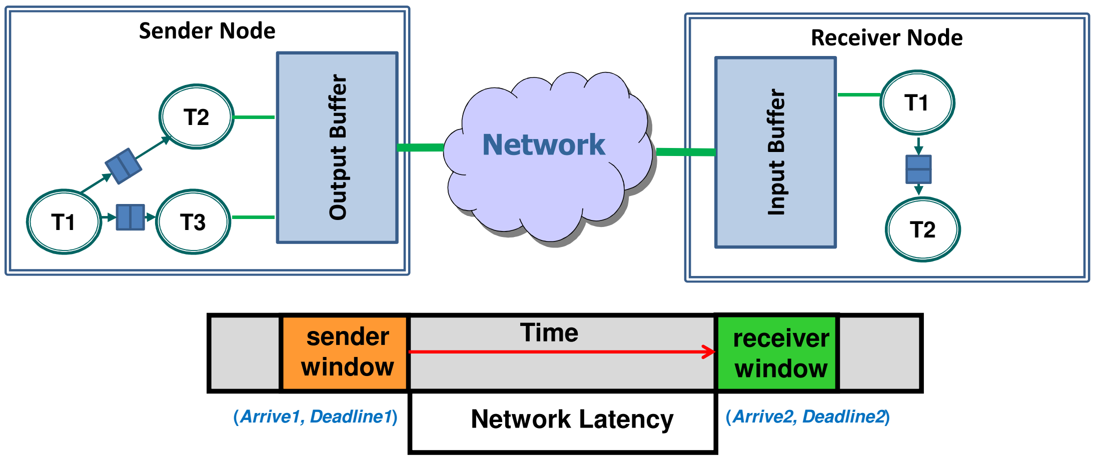


Figure 3: A Simple Distributed System and its Iteration Window

In our current work in progress we still mostly focus on the design of a single system node, with common or distinct scheduling windows (non pipelined – for simplicity) for different tasks of the given application running in the node.

2.5 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 resource 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.

We study a particular case of logical interference, which we call δ-interference or *engine-interference* [9]. 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 [9].

Interference can be *coarse-grain* or *fine-grain*. In the former case the accesses to the shared resource occurs in 'coarse' blocks, called superblocks [4], 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 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 resource conflicts become more predictable or disappear altogether, which for example can be realized in certain hardware platforms by the 'ordered transaction' approach proposed in [3].

The δ-interference is coarse-grained, and in the proposed approach we control it by explicitly scheduling the timing and ordering of engine steps. We believe that our approach can be extended to other cases of coarse-grained interference. Moreover, to manage interference in hard real-time systems we advocate *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 interference analysis and improves its accuracy. The point is that when the tasks do not shift 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.

2.6 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 4.

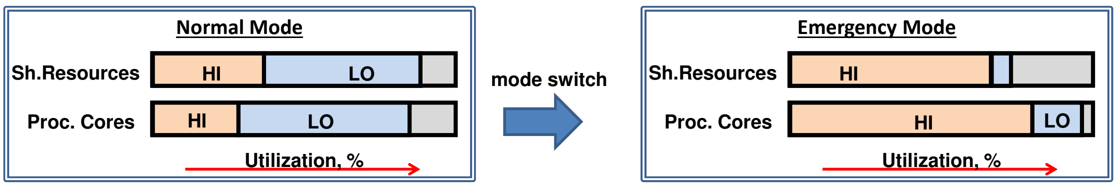


Figure 4: Mixed-criticality Resource Management

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 4, 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 [14].

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.

As explained in the previous section, to better handle interference we use the time-triggered scheduling. This approach is generalized to mixed criticality in STTM (static time triggered per mode) online policy [15, 16]. In this policy, the normal mode and the emergency mode both have a time-triggered table. A switch from normal to emergency table can occur at any time instant, while guaranteeing 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. In [16] we have proved theoretically and experimentally that in many cases this approach is as optimal in the worst case as the event-triggered approach.

3 Work-in-progress: Design Flow

3.1 Flow Structure and Assumptions

Our target design flow is shown in Figure 5. 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 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 2.5. 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).

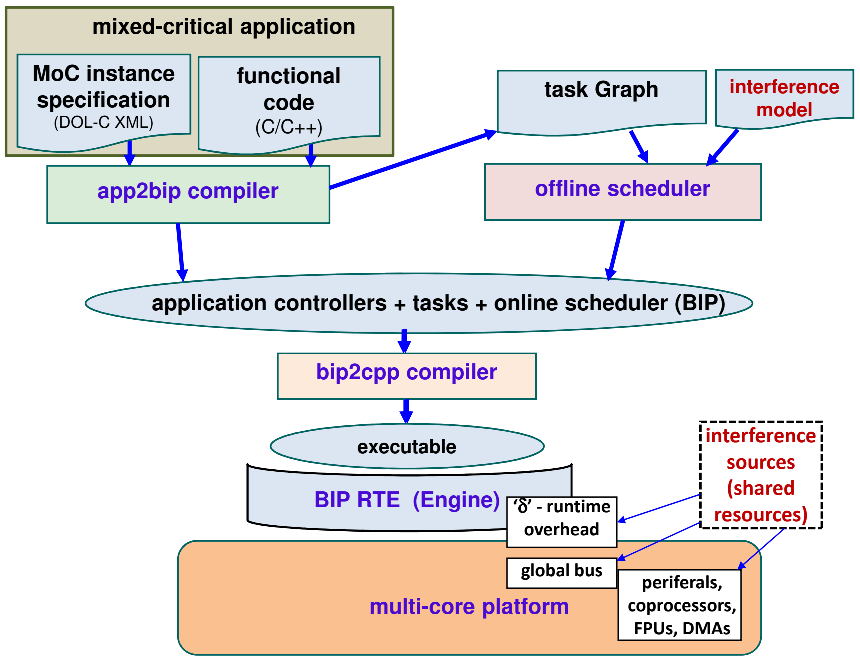

mixed-critical application
MoC instance specification (DOL-C XML)
functional code (C/C++)
task Graph
interference model
app2bip compiler
offline scheduler
application controllers + tasks + online scheduler (BIP)
bip2cpp compiler
executable
BIP RTE (Engine)
'δ' - runtime overhead
interference sources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

Figure 5: Work-in-progress Design Flow

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.

One reason for assuming non-preemption is the lack of support of preemption in many multi-core platforms with a large number of cores (many-core platforms). Another reason is the lack of support of preemption in our current runtime environment.

In our design flow we reuse certain elements from the previously presented DOL-BIP-Critical flow, which was co-developed in collaboration with partners [9]. The name of the MoC involved in the flow was DOL-Critical. It is closely related to FPPN, and the same language, named DOL-C, is currently used to specify instances of both FPPN and DOL-Critical. 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 are essential difference in the scheduling assumptions of the previous flow. The tasks are executed essentially in as soon as possible (ASAP) fashion – *i.e.,* on each core the next task starts immediately after the previous one on the same core has finished (if it is assigned to the same timing partition). Instead we impose time-triggered start of each task, which should significantly simplify the analysis of bandwidth interference. The offline scheduler had the advantage of supporting of degraded mode and of 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 it still lacks a complete support of interference analysis. Currently it offers an ad hoc model for engine δ-interference for the case of implicit-deadline periodic task controllers. The online scheduler is not yet properly integrated, as it still does not support neither dropping nor task migration.

In the remainder of the report we discuss the currently available tools illustrate their use by concrete examples. For multi-core experiments presented here, we use a LEON4 processor 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.

3.2 An Example Illustrating the Flow

Figure 6 gives a synthetic application example with three tasks. Note that this example is available as ‘parallelproc’ example in ‘Taste2BIP’ tool.

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.

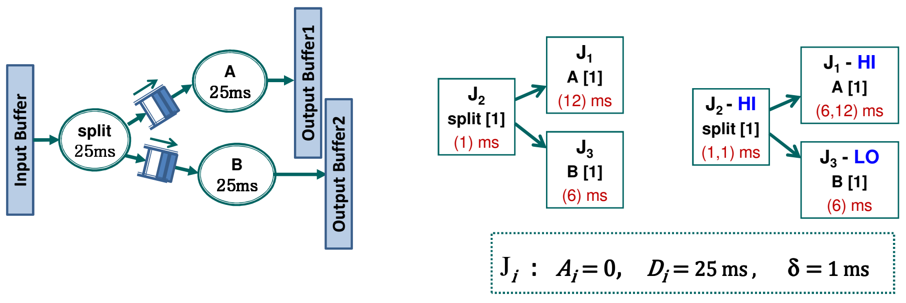
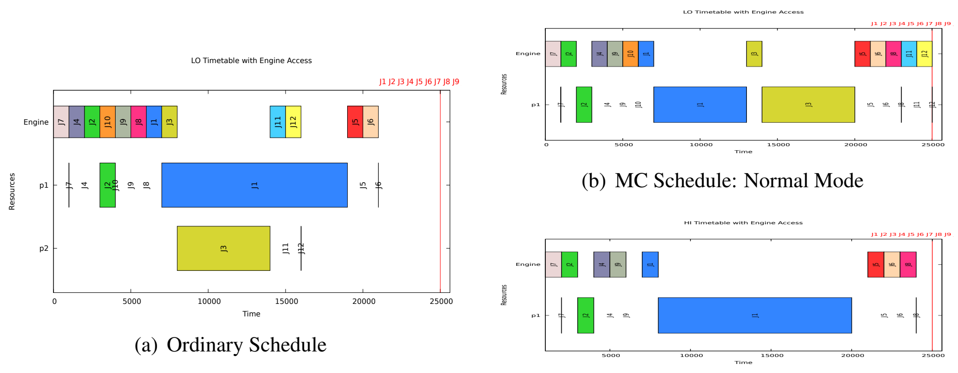


Figure 6: Three-Task Example: MoC (left), Ordinary Task Graph (middle) and Mixed-critical Task Graph



(a) Ordinary Schedule

(b) MC Schedule: Normal Mode

(c) MC Schedule: Emergency Mode

Figure 7: Three-Task Example: Offline-Scheduler Solutions

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 2-value 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 processors are necessary.

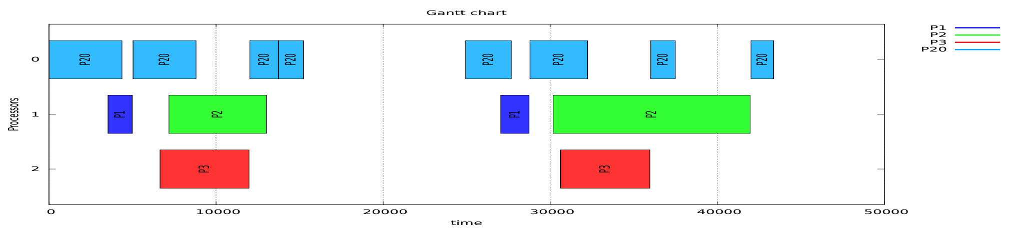
On the other hand, in the mixed-criticality case we consider the two execution modes – normal and emergency – separately. In the normal mode Task ‘A’ has execution time 6 ms, which is 6 ms less, and we have a load $25/25 = 100$ %, for which a single-core may be sufficient. In the emergency mode the execution 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 7. Figure 8 shows the Gantt charts of executing the two variants of the schedule on the LEON4 board.

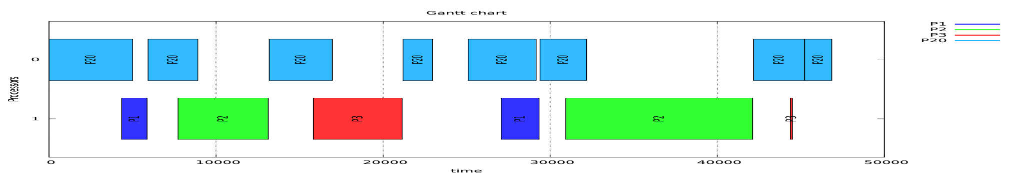
In every Gantt chart the first line shows the execution of the BIP Engine on ‘Core 0’. One may wonder why a whole core would have to be reserved to a runtime environment, but in many-core systems (or graphical accelerators), this is justifiable, as there are plenty of cores available and no preemption is allowed. On the contrary, for a multi-core system such as LEON4. which supports preemption, in future work it would be attractive to interleave high-priority engine control operations with lower-priority data computations. Note that the control thread executes also the the task controllers, the MoC controller and the online scheduler.

Recall that the shared resource on which interference-modeling is currently modeled 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’ solutions 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 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.



(a) Ordinary Execution Traces



(b) Mixed-critical Execution Traces

Figure 8: Three-Task Example: Platform Execution Traces

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 9.

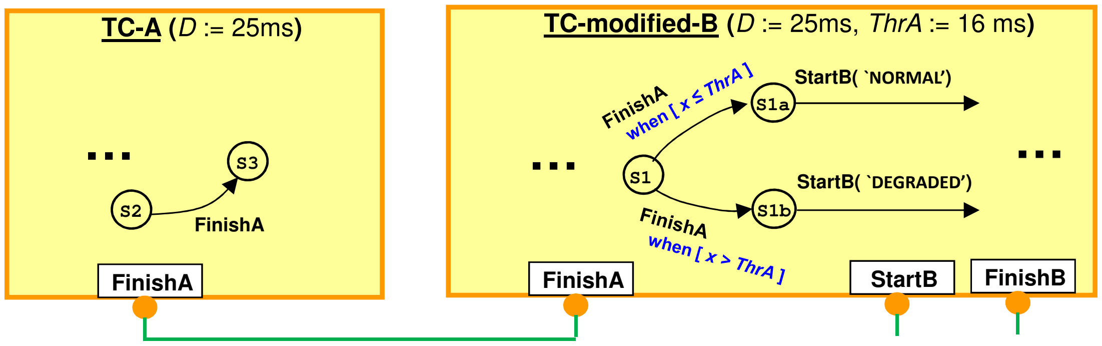


Figure 9: Three-Task Example: Manual Modification Introducing a Mode Switch

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

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 shedulability-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 6. 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 10.

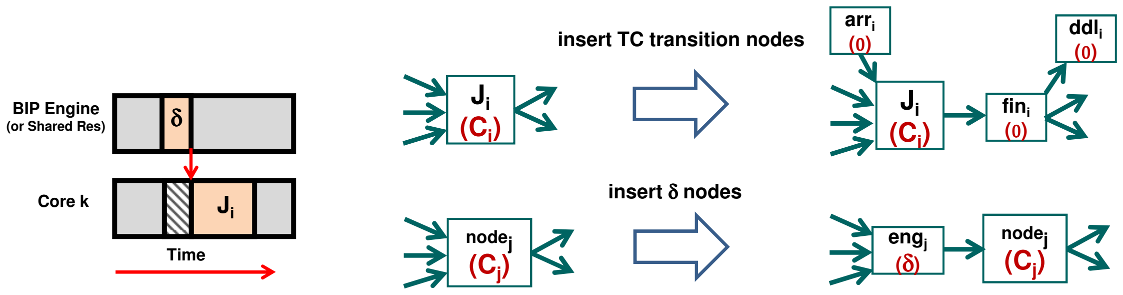


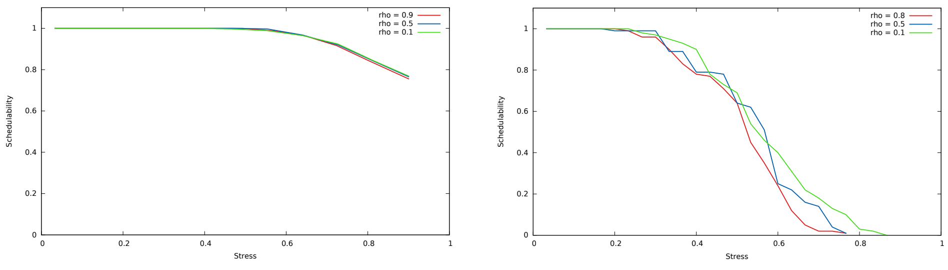
Figure 10: Engine ('Delta') Interference and its Modeling in the Task Graph

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. 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 10, 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 10 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 7(b) and 7(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 the 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



(a) Ordinary Benchmarks (b) Mixed-critical Benchmarks

Figure 11: Schedulability Results for Random Benchmarks

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. 6, 7(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.

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 [17, 16] – 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. Counter-intuitively, no significant sensitivity to the value of ρ was detected.

5 Conclusions and Future Work

In this report 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.

References

[1] 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

[2] 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

[3] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. Signal Processing and Communications, Taylor & Francis, 2009. 1, 2.5

[4] 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. 1, 2.5

[5] P. Poplavko, D. Socci, S. Paraskevas Bourgos, and M. B. Bensalem, “Models for deterministic execution of real-time multiprocessor applications,” in *DATE’15*, 2015. 2.1

[6] E. Lee and D. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, 1987. 2.1

[7] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti, “Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset,” in *19th International Conference on Real-Time and Network Systems*, 2011. 2.1

[8] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk, “A scenario-aware data flow model for combined long-run average and worst-case performance analysis,” in *MEMOCODE’06*, pp. 185–194, IEEE, 2006. 2.1

[9] 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, 2.5, 3.1

[10] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, “A timed-automata based middleware for time-critical multicore applications,” in *SEUS*, 2015. 2.2

[11] 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

[12] E. Fersman, P. Krcl, P. Pettersson, and W. Y. 0001, “Task automata: Schedulability, decidability and undecidability.,” *Inf. Comput.*, vol. 205, no. 8, pp. 1149–1172, 2007. 2.2

[13] 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

[14] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, “Modeling mixed-critical systems in real-time BIP,” in *ReTiMiCs’2013*, 2013. 2.2, 2.6

[15] S. Baruah and G. Fohler, “Certification-cognizant time-triggered scheduling of mixed-criticality systems,” in *RTSS ’11*, pp. 3–12, IEEE, 2011. 2.6

[16] 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. 2.6, 4

[17] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Multiprocessor scheduling of precedence-constrained mixed-critical jobs," in *IEEE 18th International Symposium on Real-Time Distributed Computing, ISORC 2015, Auckland, New Zealand, 13-17 April, 2015*, pp. 198–207, IEEE Computer Society, 2015. 4