Mixed Critical Automotive Embedded Applications on Multicores: A Safe Scheduling Approach for Dependability

Daniel Loche, Michaël Lauer, Matthieu Roy, Jean-Charles Fabre

Mixed Critical Automotive Embedded Applications on Multicores: A Safe Scheduling Approach for Dependability

Daniel Loche†*,
* Technocentre RENAULT
F-78280, Toulouse, France
Email: daniel.loche@renault.com

Michaël Lauer†, Matthieu Roy†, Jean-Charles Fabre†
† LAAS-CNRS
31400, Toulouse, France
Email: first.lastName@laas.fr

CARS - POSITION PAPER

***Abstract*—Memory access durations on multicore architectures are highly variable, since concurrent accesses to memory by different cores induce time interferences. Consequently, critical software tasks may be delayed by noncritical ones, leading to deadline misses and possible catastrophic failures. We present an approach to tackle the implementation of mixed criticality workloads on multicore chips, focusing on task chains, i.e., sequences of tasks with end-to-end deadlines. Our main contribution is a Monitoring & Control System able to stop noncritical software execution in order to prevent memory interference and guarantee that critical tasks deadlines are met. This paper describes our approach, and the associated experimental framework to conduct experiments to analyze attainable real-time guarantees on a multicore platform.**

***Index Terms*—multicore, real-time, deadline, task chains, mixed-criticality**

I. Introduction

Software-intensive embedded systems are getting increasingly resource-demanding. Moreover, system requirements on energy consumption, weight and space of embedded architectures are calling for a drastic reduction of computing units. Combining both trends leads to consider multicore processors as a platform to run mixed criticality workloads on a single system that integrates services inherited from federated resources.

Yet, multicore architectures impose that software has to cope with execution interferences due to memory, I/O & resource sharing, cache overwriting, and tasks synchronization. Such interferences imply that execution times are hardly predictable. In this context, schedulability analysis and WCET (Worst-Case Execution Time) estimations require high computing resources, related to the NP-completeness of the problem. A mixed criticality application runs both critical tasks and noncritical tasks with no real-time constraints. In such case, interferences from noncritical tasks may impair the temporal behavior of critical ones.

Future computers of automotive system are implemented as a set of functions implemented as chains of tasks. To handle real-time constraints, chains of tasks need to comply with end-to-end deadlines. A function has an input (request, sensor...) and generates an output (response, activation...). For instance, manual braking corresponds to a function where a driver braking input must activate the braking system, with some deadline. To fulfill this function, a chain of software tasks is executed (from the sensor to the actuator), passing through computing and decision components. Our primary goal is to ensure that, given a task chain, its end-to-end deadline is the maximum admitted response time— braking is a perfect example for this.

Running chains of tasks on a mixed criticality multicore may lead to deadline misses which may have strong side effects on safety. The main problem is to respect such constraints with the least possible compromises on noncritical software. The focus on tasks in the chain is thus of high interest to get more scheduling flexibility. Our approach takes advantage of mixed criticality software to satisfy highly critical task chains timing constraints in a multicore environment and on the same time give as much computing resources as possible to noncritical tasks.

The main difference with other approaches relies on the assumptions regarding deadlines. We consider end-to-end deadlines for task chains, instead of individual task deadlines. Most of competing solutions do not make this assumption, which matches some real-time industrial applications. In this paper, we first present the concept of our embedded Monitoring and Control Agent. Then we propose an evaluation framework and a tool to execute a task set on a real physical multicore platform and collect measurements. Ongoing experiments are run to analyze the behavior of our system regarding tasks allocation, scheduling policies, task set characteristics and deadline satisfaction.

II. State of the Art

Current solutions are dedicated either to guarantee real-time constrains with Real-Time Operating Systems (RTOS) [1], [2], [3] or to maximize resource use for a wide range of applications with General Purpose OS (GPOS) [4]. Few solutions try to mix both objectives with a more or less satisfactory result [5],[6],[7].

1) Real-time OS: On industrial applications, uncertainty factors are avoided as much as possible otherwise complexity is too high for debugging, to detect problems and unexpected behavior.

Generally, the management of real-time constraints relies on static scheduling with time slot reservation for each software partition (Time Division Multiple Access scheduling) or Worst-Case execution time (WCET) analysis. For multiple tasks, even independent with each other, running concurrently on a multicore processor, a Worst-Case scenario includes maximum contention over memory access with full cache miss, but also maximum contention on the core's utilization and so on. This leads to an unrealistic WCET estimation that will never occur. Also, a formal computation of such value is impossible, on a multicore. Estimating WCET from many experiments and simulations is a possible solution. However, it is admitted that WCET are much easier to estimate on a monocore architecture. Real-time deadline can be ensured in this case. Highly critical systems are mainly based on monocore for this reason. Multicore-based solutions needs much more compromises on tasks execution. For instance, PikeOS hypervisor [1] obtained highest certification level for rail industry on a dual-core platform using temporal partitioning. Execution time is divided into slices and applicative software is separated into multiple resource partition. A critical resource partition can be affected alone to a temporal partition in order to avoid interferences with other resource partitions. This kind of method is effective but necessarily over-reserves execution time resources. In automotive and aeronautics industries, classic AUTOSAR [2] and ARINC 653 [3] respectively applies the same approach: processors are not used at their full potential, in the favor of real-time constraints control.

2) General purpose OS: On the other side, general purpose systems bring the opposite pros and cons. Scheduling policies used are made to run lots of different kind of tasks, from highly interactive ones to background tasks, more or less computing resource demanding. GPOS schedulers reached a complexity level high enough to get inexplicable behavior [8]. These systems are highly versatile in the variety of applications they can run at the expense of predictability. It is possible to run both classic Linux processes but also real-time ones but with no strong guarantees with a vanilla kernel. Linux scheduler evolved a lot, even recently, and offers multiple scheduling strata to use, from the Completely Fair Scheduling [4] (CFS) for common process, to Round-Robin (RR) and Earliest Deadline First (EDF) for real-time tasks as described in [9]. Vanilla Linux has latencies at around hundred ms, but some patches reduce it down to micro-seconds. Comparisons can be found in [10] and [11]. For this reason, we decide to start with Linux, to take benefit from this versatility and add software tools to guarantee real-time constraints.

III. Monitoring & Control Agent

A. Concept Description

Our approach presents software execution monitoring and control with a *Monitoring and Control Agent (MCA)* to guarantee end-to-end deadline constraints. We focus on the respect of end-to-end constraints of tasks chains, not individual tasks constraints. MCA role is to pause best-effort tasks execution to free computation time for more critical tasks only when needed.

We consider critical and noncritical periodic tasks respectively considered as soft real-time and best-effort tasks. Critical task belongs to a task chain with an end-to-end deadline. Critical tasks have the following parameters:

- Priority level
- Periodicity
- Deadline - for simplification purpose, equal to period
- Core allocation - tasks are mapped to specific cores to be executed, to avoid migration

Best-effort tasks have only a periodicity and a static Core allocation. We need to be able to switch to a *backup mode* by disabling all noncritical tasks and let only critical tasks running. The goal is to dispose of guaranteed WCET for the critical task chain when executed in backup mode. Such analysis is to be done offline and we describe in section IV-A - Framework Setup a first method. Finally, we need an isolated entity able to communicate and control the system execution.

B. Monitoring & Control Agent Architecture

The Monitoring and Control Agent is made of two components: a *Core Control Component* and a *Task Wrapper Component* as shown in figure 1.

1) Core Control Component (CCC): The Core Component updates the timing state of task chains execution at a given frequency. For each task chain it stores an execution timing state with the current chain execution time and the list of executed/unexecuted tasks in the chain. It is then possible to compute a Remaining Worst-Case Execution Time (RWCET) in backup mode from the unexecuted critical tasks in the chain. If a potential end-to-end deadline miss is anticipated, best-effort tasks are stopped to switch to backup mode and guarantee said WCET under the deadline.

To sum up, CCC has at disposal:

- end-to-end deadline for every task chain
- individual tasks last execution time
- RWCET in *backup mode*

From this data, it is possible to compute the task chains current run-time WCET. We know **a)** the current task chain execution time, i.e. how much time has elapsed since the activation of the first task of the chain **b)** the remaining tasks to execute and **c)** in backup mode, how long they could take to be executed. We compare this current run-time task chain WCET to the end-to-end deadline. This is made with the following formula adapted from [12], at a given time t:

$$ET(t) + RWCET(t) + W_{max} + t_{SW} \leq D_{cj} \qquad (1)$$

Where $ET(t)$ stands for the current execution time, $RWCET(t)$: the task chain Remaining WCET when executed in isolation, $W_{max}$is the CCC updating period (*seen as a "reaction time"*) and t_{SW} the latency to switch to the monocore backup mode. When equation (1) becomes false, we need to trigger the CCC to switch to backup mode with

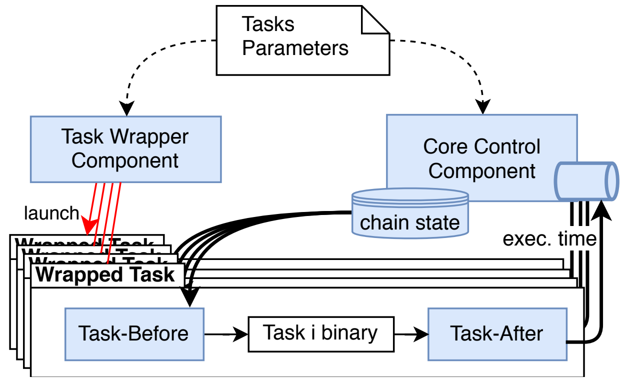

Fig. 1. Monitoring & Control Agent Architecture

only the critical task chain executed to guarantee its deadline. The mode switch is made first by sending a signal to every Best-effort task from the CCC. Upon reception, they pause themselves. In addition, the "Before" wrapper block checks whether the CCC allows task execution.

2) Task Wrapper Component (TWC): To monitor its execution, any task is wrapped with two software blocks: a "Before" and an "After" block. The "Before" bloc is activated before best-effort tasks execution, to prevent their execution if the Core Component detected a potential overloaded state and before critical tasks to inform the CCC about their start. The "after" bloc communicates execution time of every critical task to the Core Component. The Core component can update the task chains execution timing state with that information.

IV. Implementation Framework

We describe in this section how we decided to implement the Monitoring & Control Agent principle.

A. Framework Setup

The Core Control Component is launched with highest priority on an isolated core. It could be implemented on a separated processor or even a dedicated FPGA. Both the Core Component and the Task Wrapper Component get as an input a configuration file. It contains the full information about the tasks to run, their periodicity, task chains data and so on. as described previously. With such inputs the CCC is ready to run, and the TWC can encapsulate all the tasks and run them into the real-time environment. All the tasks are launched according to their core allocation. Moreover, we decided to run every critical task on the same core: the task chain uses only one core. This choice simplifies the task chain WCET analysis and avoid latencies from tasks migration for our experiments. This way offline backup mode WCET analysis is easier to compute with methods presented by [13] for instance, as only one core is activated in such state. We can also choose a semi-partitioned solution with migration allowed only for noncritical tasks. An analysis of the tasks behavior, depending on the scheduling policy is planned later.

B. Operating system and Run-time environment

The run-time environment defines how the MCA is concretely implemented. We describe here our chosen platform.

1) Linux OS: We decided to use Linux (latest Linux Mint xfce distribution) to take benefit of its possibility to run both classic Linux processes and Real-time processes with different scheduling policies (see II-2). Its versatility grants easier compatibility with benchmarking suites.

POSIX enables to force tasks execution to dedicated cores and change scheduling policy. As stated before, vanilla Linux Kernel is not made for hard real-time application. Therefore, we add a Xenomai co-kernel to improve latency down to micro-seconds and run our MCA to respect desired real-time constraints. Please note that from Linux point of view, "threads" and "processes" are equivalent and correspond to "tasks" for us.

2) Xenomai co-kernel patch: Xenomai is a real-time kernel that can be installed as a co-kernel to a classic Linux distribution as presented in deep by [14]. Our framework and experiments are implemented on the real-time APIs proposed by Xenomai 3.0.5.

V. Experimental Platform & Milestones

A. Objectives

We plan on doing experiments with a benchmark suite. The objectives are to **a)** Validate qualitatively our approach **b)** Compare our approach in term of mean CPU use and check solution weight on overhaul execution **c)** Perform a sensitivity analysis to identify how the system behaves according to multiple parameters. We can vary the ratio of critical/noncritical tasks, the time period of tasks and their execution time. Such analysis can open the way to get some conclusions on how to allocate the tasks between the cores and extend the system to more tasks chains and more cores.

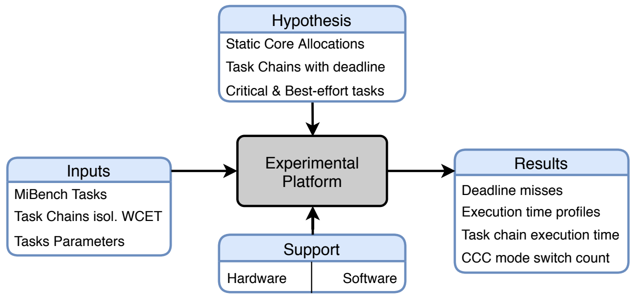

Fig. 2. Experimental Platform

In this section, we describe the different steps according to IV - Implementation Framework in order to get a complete system running with the MCA and monitoring tools. As presented in 2, the experimental has 3 input domains of which 2 are configurable. First input domain gathers the basic hypothesis. The two others are: the tasks inputs (task set, tasks parameters and task chains WCET method) and the execution support (hardware and software presented above) for which choices can be made.

B. Platform Setup

1) Hardware: The platform used for the experimentation is a bare-bone computer equipped with an Intel Core i5-8250U. This processor embeds 4 cores, with possible multi-threading (8 threads, disabled for our tests), from 1.60 GHz to 3.40 GHz. It has 3 caches level, L1, L2 and L3 (shared), with respectively 32 KiB/core, 256 KiB/core and 8 Mib.

2) Software Setup: The MiBench Benchmark suite [15] will be used for our experiments. MiBench consists of a large panel of tasks with different memory needs and execution profiles to mimic existing applications. It is used here to validate the framework and put into practice our experiments.
We selected a set of 15 applications from the benchmark for our experiments: $basicmath$, $bitcount$, $qsort$, $susan$, $jpeg$, $typeset$, $dijkstra$, $patricia$, $stringsearch$, $blowfish$, sha, $adpcm$, $CRC32$, fft and gsm. From these 15 tasks, we form task chains composed of around 5 tasks each, those tasks chains could represent real critical applications. Our Task Wrapped Component only needs to wrap MiBench tasks binaries to work.

C. Off-line characterization

The first step is to characterize the task set on our platform to assign task deadlines and task chains end-to-end deadlines, as MiBench does not include such information. The off-line analysis of backup mode is made by executing only the selected task chain in isolation on one core several times and monitor execution time. Then, we extract for each task τ_i of the chain their maximum $WCET(\tau_i)$ in such condition. This way, we define that at time t, if it remains N tasks in the chain to execute that:

$$RWCET(t) = \sum_{i=1}^{N} WCET(\tau_i) \tag{2}$$

. This way, we have all the needed information to compile them into input files for the framework and launch it.

D. Sensitivity Analysis and Prospective

We described how to generate a task set to work with, made of 1 task chain (with associated offline analysis), and a certain number of best-effort tasks selected from MiBench suite. The amount and the profile of those tasks is the first parameter we will be able to change in order to see its influence on our experiments (see Input domain in figure 2: MiBench tasks). We can select tasks following their memory use profile, change the number of “small” and “large” tasks... Tasks parameters allows to set processor charge, by changing their execution period. Also, the RWCET computation method could be changed for possible enhancements.

On the perspective scope, the hypothesis (see Hypothesis domain in figure 2) can evolve to enhance our first Framework, for example adding criticality layers to avoid stopping every noncritical tasks at once, or changing the tasks core allocation. Our experimental platform is summarized in figure 2. Such setup allows sensitivity analysis based around various parameters from the hardware (any running Linux with Xenomai) to task chain and the best-effort task set choice, through different scheduling policies and CPU load changes.

E. Comparison with other approaches

This task chain-based approach compares with other individual task timing constraints-based approaches. The first comparison point concerns computing resource use for noncritical tasks: the higher the better. The second comparison point is the tasks set coverage regarding the ability to respect end-to-end deadlines compared to the tasks set coverage of other solutions to respect individual tasks deadlines. We expect from our experiments to see how the MCA a gain in CPU use with our MCA Agent still being able to respect end-to-end deadlines. We will check how much it triggered, the execution time profiles and the influence of said parameters at run-time.

VI. Conclusion

We defined a complete process to instrument mixed-critical tasks on a multicore real-time Linux platform with a Monitoring and Control Agent to guarantee end-to-end constraints for critical task chains. Our ongoing work aims to **a)** validate the approach compared to other strategies handling mixed criticality application through on-going experiments and **b)** analyze the effect of different factors on the execution of critical software. Finally, we will apply our approach to a real automotive application case with Renault.

References

[1] S. Fisher and S. AG, “Certifying Applications in a Multi-Core Environment: The Worlds First Multi-Core Certification to SIL 4.” p. 4, 2013.

[2] AUTOSAR, “Timing Analysis,” *Standard Release 4.3.0*, p. 118, 2016.

[3] P. J. Prisaznuk, “ARINC 653 role in Integrated Modular Avionics (IMA),” in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, Oct. 2008, pp. 1.E.5–1–1.E.5–10.

[4] C. S. Wong, I. Tan *et al.*, “Towards Achieving Fairness in the Linux Scheduler,” *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 34–43, 2008.

[5] G. Giannopoulou, N. Stoimenov *et al.*, “Scheduling of mixed-criticality applications on resource-sharing multicore systems,” in *ACM International Conference on Embedded Software*, 2013, p. 17.

[6] B. C. Ward, J. L. Herman *et al.*, “Making Shared Caches More Predictable on Multicore Platforms.” IEEE, Jul. 2013, pp. 157–167.

[7] A. Blin, C. Courtaud *et al.*, “Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System,” p. 11, 2016.

[8] J.-P. Lozi, F. Gaud *et al.*, “The Linux Scheduler: a Decade of Wasted Cores,” p. 16, 2016.

[9] J. Lelli, G. Lipari *et al.*, “An efficient and scalable implementation of global EDF in Linux,” *7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT11)*, 2011.

[10] F. Cerqueira and B. B. Brandenburg, “A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUSRT.” SYSGO AG, 2013, pp. 19–29.

[11] D. J. H. Brown and B. Martin, “How fast is fast enough? Choosing between Xenomai and Linux for real-time applications,” Tech. Rep., 2010.

[12] A. Kritikakou, T. Marty, and M. Roy, “DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 23, no. 2, pp. 1–26, 2017.

[13] R. Wilhelm, T. Mitra *et al.*, “The worst-case execution-time problemoverview of methods and survey of tools,” *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, Apr. 2008.

[14] P. Gerum, “Xenomai - Implementing a RTOS emulation framework on GNU/Linux,” Xenomai, Tech. Rep., 2004.

[15] M. R. Guthaus, J. S. Ringenberg *et al.*, “MiBench: A free, commercially representative embedded benchmark suite.” Austin, TX, USA: IEEE, Dec. 2001, p. 12.