HAL open science

Mitigating Mode-Switch through Run-time Computation of Response Time

Angeliki Kritikakou, Stefanos Skalistis

▶ **To cite this version:**

Angeliki Kritikakou, Stefanos Skalistis. Mitigating Mode-Switch through Run-time Computation of Response Time. ACM Transactions on Design Automation of Electronic Systems, 2023, 28 (5), pp.1-26. 10.1145/3597432 . hal-04397350

HAL Id: hal-04397350

https://hal.science/hal-04397350v1

Submitted on 16 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Mitigating Mode-Switch through Run-time Computation of Response Time

ANGELIKI KRITIKAKOU, Univ Rennes, Inria, IRISA, France
STEFANOS SKALISTIS, Collins Aerospace, Ireland

Mixed-critical systems consist of applications with different criticality. In these systems, different confidence levels of Worst-Case Execution Time (WCET) estimations are used. Dual criticality systems use a less pessimistic, but with lower level of assurance, WCET estimation, and a safe, but pessimistic, WCET estimation. Initially, both high and low criticality tasks are executed. When a high criticality task exceeds its less pessimistic WCET, the system switches mode and low criticality tasks are usually dropped, reducing the overall system Quality of Service (QoS). To postpone mode switch, and thus, improve QoS, existing approaches explore the slack, created dynamically, when the actual execution of a task is faster than its WCET. However, existing approaches observe this slack only after the task has finished execution. To enhance dynamic slack exploitation, we propose a fine-grained approach that is able to expose the slack during the progress of a task, and safely uses it to postpone mode switch. The evaluation results show that the proposed approach has lower cost and achieves significant improvements in avoiding mode-switch, compared to existing approaches.

CCS Concepts: • **Computer systems organization** → **Embedded software**; **Real-time systems**.

Additional Key Words and Phrases: Worst-Case Execution Time, Interference-sensitive, Run-time Adaptation, Time-triggered, Response Time Analysis, Multi-cores

ACM Reference Format:
Angeliki Kritikakou and Stefanos Skalistis. 2023. Mitigating Mode-Switch through Run-time Computation of Response Time. *ACM Trans. Des. Autom. Electron. Syst.* 37, 4, Article 111 (August 2023), 26 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Mixed-critical systems [31] consist of applications with different levels of criticality. The application's criticality level partially depends on the consequences on the system, when the application fails to meet its timing constraints [26]. As a result, applications with different criticality levels have different properties and requirements. A high criticality application requires strict timing guarantees, i.e., ending before its deadline. To ensure timing guarantees, the Worst-Case Execution Time (WCET) of the application has to be considered. Nonetheless, the WCET estimation depends on the application's criticality level [11]; the same code has a higher WCET, if it requires a higher level of assurance, than it would, if it was considered as a non-critical application [11]. When computing WCET estimations, pessimism is introduced due to application and processor complexity. Applications have several execution paths, and thus, the worst-case path is used during WCET computation. Processor components, that take decisions dynamically, have a difficult-to-predict timing behavior, e.g., cache memories and branch predictors. When pessimistic WCET estimations

Authors' addresses: Angeliki Kritikakou, angeliki.kritikakou@irisa.fr, Univ Rennes, Inria, IRISA, Campus Beaulieu, Rennes, France, 35042; Stefanos Skalistis, stefanos.skalistis@collins.com, Collins Aerospace, xxxx, Cork, Ireland, xxxx.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
1084-4309/2023/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

are used, resources are over-allocated to high criticality applications. However, to provide an overall high Quality of Service (QoS), low criticality tasks should be executed as long as possible [11].

To improve the execution of low criticality tasks, while still guaranteeing safe execution of high criticality tasks, different WCET estimations are used for high criticality tasks; a pessimistic, with high assurance, upper bound (C^H) and a less pessimistic, with lower assurance, bound (C^L). Low criticality tasks are usually bounded by less pessimistic WCET estimations (C^L). A dual-criticality system has two executions modes: low criticality mode (LO-mode), executing low and high criticality tasks considering C^L, and high criticality mode (HI-mode), executing only high criticality tasks considering C^H. The system starts in LO-mode. Following the common approach, called BaseLine (BL), the system checks at a specific time instance, given by the C^L, if the task has finished execution. If not, the task is dropped, if it is a low criticality task, or the system switches from LO-mode to HI-mode, if it is a high criticality task. In HI-mode, low criticality tasks are usually dropped, e.g.,[3, 5, 10]. Section 2 provides an illustration example of a mixed-criticality system and the BL approach.

However, in BL approach the mode-switch can occur at specific time instances, which are defined upfront before execution and are equal to the C^L of each high criticality task. To improve the execution of low criticality tasks, existing approaches work on two directions: i) explore other strategies, than dropping low criticality tasks in HI-mode, and ii) explore static or dynamic ways to postpone the mode-switch. In this work, we focus on the second category. Note that, the proposed approach can be combined with approaches of the first category. Regarding the second category, existing static approaches determine the largest value, to be added to the C^L of the high criticality tasks, while the system still remains schedulable; then, this value is used to extend the mode-switch further than C^L. Such methods are inspired by sensitivity analysis [27] and zero-slack [13, 14]. However, the static approaches are applied before execution, thus exploring only the existing slack due to system under-utilisation. On the contrary, dynamic approaches (DYN) are able to exploit the slack created during execution. When the actual execution time of a task is lower than its C^L, slack is created, since the task finished earlier than expected in LO-mode. This slack can be used by the next high criticality tasks and potentially postpone the mode-switch, e.g., through single budget [17], bailout protocol [7] and feedback control mechanisms [23]. However, existing DYN approaches are able to observe and use the dynamic slack, only after a task has terminated. This limitation is highlighted in the illustration example of Section 2.

To address the aforementioned limitation, this work extends the state-of-the-art with a safe and lightweight approach that dynamically computes, not only the slack created due to the early termination of tasks, but also the slack created due to the actual progress of active tasks, during execution, and safely uses it in order to postpone or even avoid mode-switch. Following this approach, there is no need to apply at run-time approaches that have high overhead and are typically applied offline, such as schedulability test or response time analysis, as in [28]. To achieve that, we propose a run-time controller, which is regularly evoked at a set of points inserted to the high criticality job (named instrumentation points) during execution, when the system is in LO-mode. The controller computes the available dynamic slack based on a safe run-time computation of the worst-case response-time bound of the running job. In this way, the actual execution progress of both finished jobs and currently active jobs is exposed at a given point. The controller computes any dynamic slack created between two instrumentation points of the job, based on the difference of the worst-case response-time bound computed at the current and previous points. More precisely, the worst-case response-time bound of a job of a high criticality task at an instrumentation point is computed by taking into account the actual time when the controller is evoked, the remaining worst case delay, that can still occur for this job from point until the job ends, due to preemption by higher priority (low and high criticality) tasks, and the remaining WCET, which corresponds to the

WCET of the code that remains to be executed from the point until the job ends. After the dynamic slack computation, a simple, and safe, condition decides whether a mode-switch can be postponed for later. From our extensive experiments, RRT was able to avoid mode-switch in 50.10% of the experiments, on average. As a result, on average, 60.34% low criticality tasks finished execution.

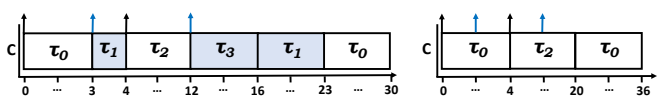
The paper is organized as follows: Section 2 describes the limitations of the state-of-the-art and the add-on of the proposed approach through and illustration example. Section 3 presents the system model. Section 4 presents the proposed approach and its Response Time Analysis (RTA). Section 5 presents the evaluation results. Section 6 presents the state-of-the-art, whereas Section 7 concludes this study.

2 ILLUSTRATION EXAMPLE

We use a simple example based on a dual-criticality mixed-critical system to illustrate the limitations of the state-of-the-art and show how the proposed approach remedies them. Table 1 shows the characteristics of the dual-criticality mixed-critical system, where tasks τ_0 and τ_2 are high criticality tasks, and tasks τ_1 and τ_3 are low criticality tasks. LO-mode mode is depicted in Fig. 1a) and HI-mode mode in Fig. 1b.

Task	C^L	C^H	Priority	Arrival	Criticality
τ_0	10	20	3	0	HI
τ_1	8	-	2	3	LO
τ_2	8	16	0	4	HI
τ_3	4	-	1	12	LO

Table 1. Illustration example (Period: 40 time units).

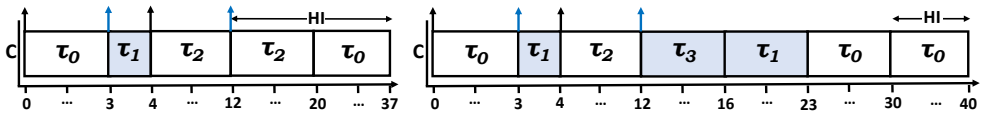


(a) LO-mode execution. (b) HI-mode execution.

Fig. 1. Illustration example a) LO-mode and b) HI-mode.

The behavior of the BL approach is illustrated in Fig. 2a, when τ_2 exceeds its $C^L_{\tau_2}$ at time $t = 12$, and in Fig. 2b, when τ_0 exceeds its $C^L_{\tau_0}$ at $t = 30$. The DYN limitation is depicted in Fig. 3. During execution, at $t = 3$, let's assume that τ_0 has already executed a large part of its code, but, still, the actual execution time is less than its $C^L_{\tau_0}$ $(3 < 10)$. Therefore, when τ_0 is preempted, it has not reached its $C^L_{\tau_0}$, in order to decide mode-switch. At $t = 12$, when τ_2 reaches its $C^L_{\tau_2}$ without terminating, the system has to switch to HI-mode, since no slack is available at that moment. The slack created due to the progress of τ_0 can be seen at this moment, since τ_0 has not finished execution. Only after task termination $(t = 20)$, the slack created by τ_0 can be observed and used.

The proposed approach is based on the Run-time worst-case Response Time (RRT). To illustrate the proposed RRT approach, and keep the complexity of the illustration example low, we use a simple instrumentation for the high criticality tasks of Table 1, i.e., insertion of instrumentation points in a uniform way. As schematically depicted in Fig. 4a, we uniformly inserted five instrumentation points to τ_0 and four instrumentation points to τ_2, and the partial WCET, required to reach from one point the next one, is $C^H_{ptp} = 4$ in HI-mode and $C^L_{ptp} = 2$ in LO-mode. In section 4 we describe how the instrumentation points are placed in high criticality jobs. Let's now assume that i) at



(a) Mode-switch at $t = 12$, where τ_2 exceeds its $C^L_{\tau_2}$.

(b) Mode-switch at $t = 30$, where τ_0 exceeds its $C^L_{\tau_0}$.

Fig. 2. BL approach.

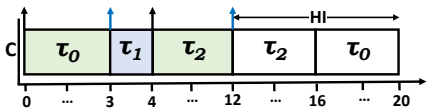


Fig. 3. DYN approach. Mode-switch at $t = 12$, τ_2 exceeds its $C^L_{\tau_2}$ and no slack is available.

$t = 3$ (Fig. 4b), due to a faster execution of τ_0, compared to its WCET, the fourth instrumentation point is reached, and ii) the RRT computed in the previous points of τ_0 was equal to the initial response-time bound, computed offline, i.e., $R^L_{\tau_0} = 30$. The controller computes a new value for the RRT in LO-mode (RR^L) of τ_0, based on the actual time (t) at which the controller is evoked, the remaining worst-case delay (RD^L_{hp}) and the remaining WCET (RC^L) of τ_0 from the fourth point until the task ends, i.e., $RR^L_{\tau_0} = t + RD^L_{hp(\tau_0)} + RC^L_{\tau_0} = 3 + (12 + 8) + 2 = 25$. The difference between the RRT of the current and previous points provides the dynamic slack in LO-mode, i.e., $DS^L = 5$. Mode-switch is decided based on whether the computed dynamic slack is higher than or equal to the additional time required to reach the next instrumentation point in the worst-case, denoted as C_{ptp}. This worst-case additional time is given when a high criticality task takes WCET in HI-mode, instead of WCET in LO-mode, to reach the next instrumentation point, i.e., $C_{ptp} = C^H_{ptp} - C^L_{ptp}$, e.g., $C_{ptp} = 2$ in the illustration example. At $t = 12$, the third instrumentation point of τ_2 is reached and the RRT, computed at the previous points of τ_2, is equal to the response-time bound computed offline, i.e., $R^L_{\tau_2} = 12$. A new value for the response-time bound in LO-mode of τ_2 is computed, i.e., $RR^L_{\tau_2} = t + RD^L_{hp(\tau_2)} + RC^L_{\tau_2} = 12 + (0 + 0) + 2 = 14$. Comparing with the previously computed RRT, the slack between the two points is negative, i.e., -2. The overall dynamic slack is updated, $DS^L = 5 + (-2) = 3$. As the overall slack is higher than or equal to C_{ptp}, there is no need to perform mode-switch at $t = 12$; there is enough slack to reach the next instrumentation point and take the decision later, even if the task execution takes WCET in HI-mode. Thus, the proposed approach is able to avoid mode-switch, compared to DYN approach.

3 SYSTEM MODEL

3.1 System model

We consider a uni-processor system with a set of tasks $\mathcal{T}$ to be executed preemptively. A dual-criticality system is assumed, where each task has a level of criticality equal to either high (H) or low (L), with $H > L$. We use the basic mixed-criticality model, where the system has two modes of execution: i) LO-mode, where both high criticality tasks and low criticality tasks are executed on the processor, and ii) HI-mode, where only the high criticality tasks are executed on the processor. When a job of a low criticality task exceeds its WCET, the job is dropped, whereas when a high criticality job exceeds its WCET, a mode-switch occurs. The proposed approach has as goal to postpone or avoid mode-switch, even if a high criticality task exceeds its WCET, as long as it does

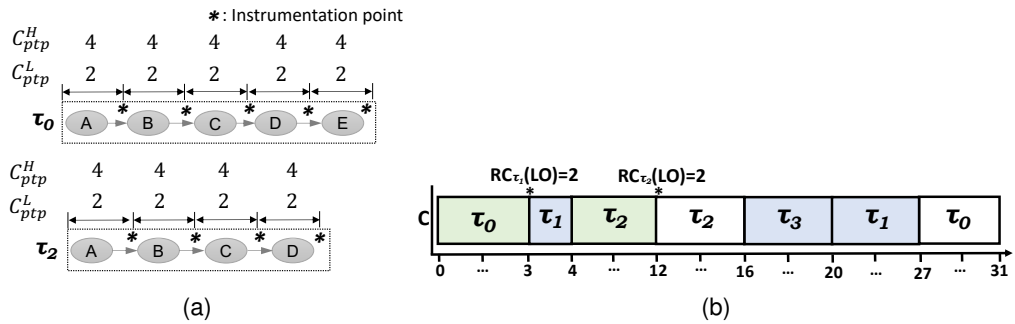


Fig. 4. RRT approach. a) Instrumentation of high criticality tasks and C_{ptp} in LO-mode and in HI-mode, and b) No mode-switch, τ_2 exceeds its $C^L_{\tau_2}$ at $t = 12$, but enough slack exist due to τ_0 progress.

Table 2. Summary of main notation

System model	
$\mathcal{T}$, τ	Task τ belonging to task-set $\mathcal{T}$
CL_τ	Criticality level CL of task τ, $CL \in \{H, L\}$
r_τ, D_τ	Arrival time and deadline of task τ
C^{CL}_τ	WCET C^{CL}_τ of task τ at criticality level CL
P	Common period P
Remaining WCET	
$\mathcal{S}$	Set of functions $\mathcal{S} = \{F_0, F_1, ..., F_l\}$, with F_0 being the main function
G	The ECFG $G = (V, E)$ of a function F
V	The set of nodes $V = \mathcal{N} \cup \mathcal{C} \cup \mathcal{F} \cup \{IN\} \cup \{OUT\}$ of an ECFG G
E	The sets of edges $E \subseteq V \times V$ of ECFG G
$N \in \mathcal{N}$	Block of one or more binary instructions
$C \in \mathcal{C}$	Block of binary instructions of a condition statement
$F_i \in \mathcal{F}$	Binary instructions of the function caller of a function F_i
IN, OUT	The input/output node of ECFG G
p_τ	A point of task τ
$level[p_\tau]$, $head[p_\tau]$	The depth of point p_τ and its ancestor point $head[p_\tau]$
$C_\tau[p'_\tau\text{-}p_\tau]$	Partial WCET from point p'_τ to p_τ of task τ
STI_τ	Structure & Timing Information (*level*, *head* & partial WCET of τ points)
RC_τ	Remaining WCET of task τ
Remaining Response Time	
C_{ptp}	Overhead to reach next point, with WCET in HI-mode, instead of LO-mode
$PD^L_{hp(\tau)}$	Preemption delay of τ in LO-mode, due to higher priority tasks
$RD^L_{hp(\tau)}$	Remaining preemption delay of τ in LO-mode, due to higher priority tasks
RR^L_τ	LO-mode remaining response time of τ in LO-mode
DS^L	Dynamic slack
$\Delta^p_{p+1}x$	Difference of x at point p and at point $p + 1$

not exceed its response time. We define the response time of a task τ_i per criticality-level, $R^{CL}_{\tau_i}$, and the response time when a mode-switch occurs, $R^*_{\tau_i}$. Each task, $\tau_i \in \mathcal{T}$, is characterized by its arrival

time r_{τ_i}, deadline D_{τ_i}, criticality level CL_{τ_i} and the WCET $C^{CL}_{\tau_i}$ for each criticality level, i.e., $C^{H}_{\tau_i}$ and $C^{L}_{\tau_i}$. $C^{CL}_{\tau_i}$ is assumed to be monotonically non-decreasing, with increasing criticality level CL [4]. Tasks are periodically executed with a common period P, i.e., at the k-th period a task releases a job at time $k * P + r_{\tau_i}$. Jobs can be either dependent or independent. We focus on constrained deadlines, i.e., $D_{\tau_i} \leq P$. The task-set $\mathcal{T}$ is considered to be executed preemptively with a scheduling policy S following a Fixed Priority (FP) scheme using a unique priority assignment algorithm (e.g. Audsley's Algorithm [2]). Notice that, a low criticality task could have higher priority than a high criticality task. Our task model depicts asynchronous mono-periodic task-sets. Multi-periodic task-sets could be unrolled to the hyper-period of the system, i.e., the least common multiple of the task periods, and the approach applied inside the hyper-period. Note that, in such a case, the unrolled jobs are executed once in the hyper-period and thus, they do not require additional memory to store information, as they can reuse the memory space of the previous job. Between hyper-periods, the dynamic slack is set to zero. More efficient ways to deal with multi-periodic task sets are future work. A summary of the main notation is presented in Table 2.

4 PROPOSED APPROACH

Our goal is to postpone, or even avoid, mode-switch, even if a high criticality job exceeds its WCET. To achieve that, the proposed approach regularly computes in a fine-grained and safe way the dynamic slack in LO-mode and uses it to safely postpone mode-switch. It takes into account not only the actual progress of the running high criticality job (that invoked the controller), but also the actual execution progress of the remaining active, but preempted, jobs and the actual execution time of already finished jobs. Fig. 5 overviews the proposed approach. Section 4.1 describes the design-time analysis, Section 4.2 presents the run-time control and Section 4.3 presents the approach safety. Table 3 summarises the acronyms.

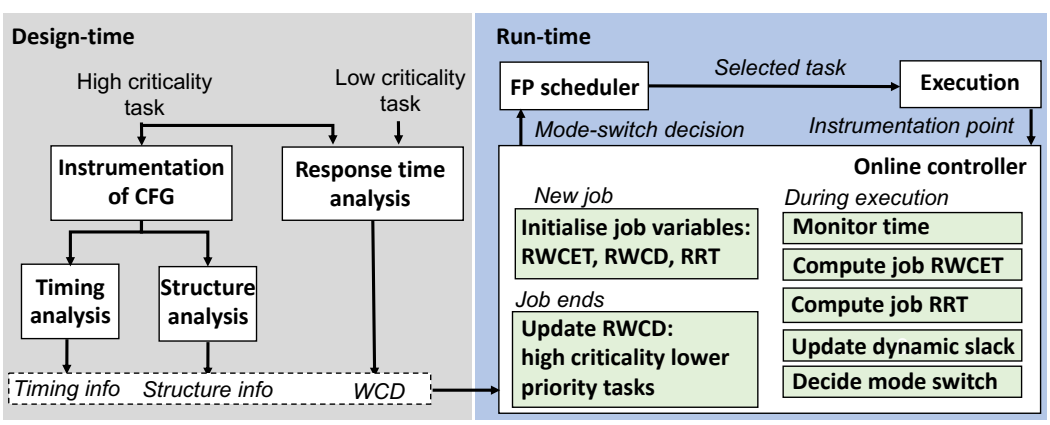


Fig. 5. Overview of the proposed approach.

4.1 Design-time system analysis

Design-time analysis performs the task instrumentation and extracts the information required for the run-time computation of the dynamic slack. We use the approach of [20] as a starting block, and leverage it in order to be applicable for the system under study. Note that, the approach of [20] is designed for only a single task with a unique WCET estimation running on one processor

Table 3. Summary of acronyms

QoS	Quality of Service	WCET	Worst-Case Execution Time
BL	BaseLine	RWCET	Remaining Worst-Case Execution Time
DYN	DYNamic	RRT	Run-time worst-case Response Time
LO-mode	LOw criticality mode	HI-mode	HIgh criticality mode
RTA	Response Time Analysis	FP	Fixed Priority
WCD	Worst-Case Delay	RWCD	Remaining Worst-Case Delay
CL	Criticality Level	STI	Structure and Timing Information
CFG	Control Flow Graph	ECFG	Extended Control Flow Graph

without preemption. As a result, it cannot be applied in systems where multiple periodic tasks with different criticalities are preemptively executed on the same processor. Our approach addresses this limitation by computing at run-time the worst-case response-time bound. To achieve this, not only information regarding the RWCET per high criticality task in LO-mode is needed, but also information regarding the overall system execution, i.e., the Worst-Case Delay (WCD) in LO-mode per high criticality task, due to the preemption from higher priority (low and high criticality) tasks in any period.

High criticality task: A high criticality task is described by a set of Control Flow Graph (CFGs), constructed by the assembly code. Each CFG corresponds to a function F of the high criticality task. Therefore, the high criticality task τ_i is a set of functions $\mathcal{S} = \{F_0, F_1, ..., F_l\}$, with F_0 the main function.

Definition 1. *The CFG of a function F is a directed graph $G = (V, E)$, consisting of:*

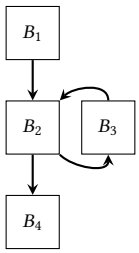
- *A finite set of nodes V composed of 5 disjoint sub-sets $V = \mathcal{N} \cup \mathcal{C} \cup \mathcal{F} \cup \{IN\} \cup \{OUT\}$:*
  - *i) $N \in \mathcal{N}$ represents a block of one or more binary instructions,*
  - *ii) $C \in \mathcal{C}$ represents the block of binary instructions of a condition statement,*
  - *iii) $F \in \mathcal{F}$ represents the binary instructions of the function caller of a function F and links the node of the current function with the CFG of the function F,*
  - *iv) IN is the input node,*
  - *v) OUT is the output node,*
- *a finite set of edges $E \subseteq V \times V$ representing the control flow between nodes.*

Therefore, a CFG can include the following components: i) a single node N, ii) an if-then-else component, i.e., the concatenation of a C conditional node with two mutually executed paths that have the same end node, iii) a loop component, i.e., the concatenation of a loop condition C with two mutually executed paths, one with the exit path and one with the loop repetition, and iv) a function call node F. F Fig. 6 illustrates how to obtain CFG through an example. L.1 to L.9 (Fig. 6c) handle the stack and initialise the local variables and correspond to B_1, which is a component of type N, (Fig. 6b), L.10 to L.14 describe the exit condition of the loop and correspond to B_2 (which is a component of type C), L.15 to L.26 describe the loop kernel and the increase of i and correspond to B_3 (component of type N), and L.27 to L.32 manage the stack and performs the return from the function call correspond to B_4 (component of type N). A special point *start* exists before task execution, which denotes the function call to the main function.

Instrumentation points p_{τ_i} are inserted in the high criticality task τ_i, in order to evoke the controller that will compute the remaining WCET at run-time. Instrumentation points can be inserted before the execution of the first binary instruction of each node of CFG. Representing instrumentation points by a lower-case symbol, five disjoint sub-sets of instrumentation points

```c
int main(void){
  int i;
  int A[10];
  for (i=0;i<10;i++){
    A[i]=i;
  }
  return 0;
}
```

(a) C code.



(b) Basic blocks

```asm
main:
        addi    sp, sp, -64
        sw      ra, 60(sp)
        sw      s0, 56(sp)
        addi    s0, sp, 64
        li      a0, 0
        sw      a0, -12(s0)
        sw      a0, -16(s0)
        j       .LBB0_1
.LBB0_1:
        lw      a1, -16(s0)
        li      a0, 9
        blt     a0, a1, .LBB0_4
        j       .LBB0_2
.LBB0_2:
        lw      a0, -16(s0)
        slli    a2, a0, 2
        addi    a1, s0, -56
        add     a1, a1, a2
        sw      a0, 0(a1)
        j       .LBB0_3
.LBB0_3:
        lw      a0, -16(s0)
        addi    a0, a0, 1
        sw      a0, -16(s0)
        j       .LBB0_1
.LBB0_4:
        li      a0, 0
        lw      ra, 60(sp)
        lw      s0, 56(sp)
        addi    sp, sp, 64
        ret
```

(c) Assembly code (RISC-V).

Fig. 6. Example: CFG is obtained from complied C code.

can exist, based on the node type: $\{n\}$, $\{c\}$, $\{f_i\}$, *in*, *out*. Note that, the point *start* refers to the point before execution, i.e., point *in* of function F_0. Such instrumentation points can be implemented in several ways depending on which level of abstraction the high criticality task code is given and whether we can modify the hardware of the processor. When the source code is available, a function call can be inserted in the source code to call the controller code. When only the assembly code is available, we can modify the assembly code in order to add the call to the controller. When the binary code is available, we can obtain the assembly code from the binary code, for instance using a disassembler. If the hardware is modifiable, a hardware block can be inserted to monitor the program counter and execute the controller.

After the insertion of the instrumentation points, we use an ECFG parser in order to extract information regarding the ECFG structure that will allow us to distinguish different visits of the same instrumentation point during execution (e.g., in loops, function calls).

Definition 2. *The structure information of a point p_{τ_i} is:*

- *The nested level of p_{τ_i},* level$[p_{\tau_i}]$*, which is:*
  - *i) set to* 0*, if p_{τ_i} is the* start *point,*
  - *ii) set to* 1*, if p_{τ_i} is a sequential point between the IN and OUT of an ECFG,*
  - *iii) increased by* 1*, for each loop where p_{τ_i} resides in.*
- *The ancestor point of p_{τ_i},* head$[p_{\tau_i}]$*, that indicates the point where a loop entry or a function call occurred. The* head$[p_{\tau_i}]$ *of a point p_{τ_i} is:*
  - *i) the* start *point, if p_{τ_i} is a point with* level *1 in the main function F_0,*

ii) the function caller, if p_{τ_i} is a point with level *1 in the called function,*
iii) the condition of the loop, if p_{τ_i} is inside a loop.

- *The function call behavior,* type$[p_{\tau_i}]$*, which is:*
  i) F_ENTRY, if p_{τ_i} is a function entry (function caller),
  ii) F_EXIT, if p_{τ_i} is a function exit, i.e. the node where a function returns to,
  iii) F_ENEX, if p_{τ_i} is both a function entry and a function exit, i.e. the point p_{τ_i} where the function returns is also a function caller,
  iv) −, if p_{τ_i} is not related to function calls.

Let's illustrate the structure information with the example of Fig. 6b. With b_1, b_2, b_3 and b_4 being the points inserted in the beginning of each CFG block, we obtain: *level*$[b_1]$=1, *level*$[b_2]$=1, *level*$[b_4]$=1 and *level*$[b_3]$=2, *head*$[b_1]$=*start*, *head*$[b_2]$=*start*, *head*$[b_4]$=*start*, and *head*$[b_3]$=b_2, *type*$[b_1]$=−, *type*$[b_2]$=−, *type*$[b_4]$=−, and *type*$[b_3]$=−.

Last, we extract the timing information by extending the approach of [20] to be applied per criticality level.

Definition 3. *Let x_{τ_i} and p_{τ_i} be two instrumentation points; the partial WCET between these points in a given criticality level CL is $C^{CL}_{\tau_i}[x_{\tau_i}\text{-}p_{\tau_i}] = C^{CL}_{\tau_i}[x_{\tau_i}] - C^{CL}_{\tau_i}[p_{\tau_i}]$, where $C^{CL}_{\tau_i}[x_{\tau_i}]$ denotes the WCET from point x_{τ_i} until the end of code execution and $C^{CL}_{\tau_i}[p_{\tau_i}]$ denotes the WCET from point p_{τ_i} until the end of execution.*

Two types of partial WCET are computed: (1) For all points, we compute $C^{CL}_{\tau_i}[head[p_{\tau_i}]\text{-}p_{\tau_i}]$, and (2) For points placed in the entry of a loop, we compute the $C^{CL}_{\tau_i}$ between any two consecutive loop iterations ($j-1$ and j), i.e., $C^{CL}_{\tau_i}[p^{j-1}_{\tau_i}\text{-}p^{j}_{\tau_i}]$. If multiple paths exist between these points (e.g., branches of if-then-else components, function calls from different entry points), the minimum difference is maintained. Note that, the minimum value is required in order to be safe, since this value will be subtracted from the overall WCET, during RWCET computation at run-time [20].

The timing information extracted for Fig. 6 is: $C^{L}_{\tau_i}[start\text{-}b_1]$ and $C^{H}_{\tau_i}[start\text{-}b_1]$, $C^{L}_{\tau_i}[start\text{-}b_2]$ and $C^{H}_{\tau_i}[start\text{-}b_2]$, $C^{L}_{\tau_i}[start\text{-}b_4]$ and $C^{H}_{\tau_i}[start\text{-}b_4]$, $C^{L}_{\tau_i}[b_2\text{-}b_3]$ and $C^{H}_{\tau_i}[b_2\text{-}b_3]$, $C^{L}_{\tau_i}[b^{j-1}_2\text{-}b^{j}_2]$ with j=0 . . . 9 and $C^{H}_{\tau_i}[b^{j-1}_2\text{-}b^{j}_2]$ with j=0 . . . 9 .

The above partial WCETs are computed in LO-mode and in HI-mode. Note that, the partial WCETs for LO-mode will be used by the run-time controller for RWCET computation in LO-mode. The partial WCETs for HI-mode are used only at design-time to compute the additional time required to reach the next instrumentation point, in the worst-case, denoted as point-to-point overhead C_{ptp}, used to safely decide mode-switch. The worst-case is when the high criticality job takes $C^{HI}_{\tau_i}[x_{\tau_i}\text{-}p_{\tau_i}]$, instead of $C^{LO}_{\tau_i}[x_{\tau_i}\text{-}p_{\tau_i}]$, to reach point p_{τ_i} from point x_{τ_i}. It is computed as C_{ptp}=$max(C_{ptp,F}, C_{ptp,B})$, where $C_{ptp,F}$=$max(C^{H}_{\tau_i}[head[p_{\tau_i}]\text{-}p_{\tau_i}]$ - $C^{L}_{\tau_i}[head[p_{\tau_i}\text{-}p_{\tau_i}])$, ∀ i and p_{τ_i}, and $C_{ptp,B}$=$max(C^{H}_{\tau_i}[p^{j-1}_{\tau_i}\text{-}p^{j}_{\tau_i}]$ - $C^{L}_{\tau_i}[p^{j-1}_{\tau_i}\text{-}p^{j}_{\tau_i}])$, ∀ i, p_{τ_i} ∈ c, and j. The maximum difference provides the worst-case overhead among points.

System Analysis: Initially, we verify whether the task-set $\mathcal{T}$ is schedulable in LO-mode, HI-mode, and mode-switch. The task-set $\mathcal{T}$ at each mode is considered schedulable if the worst-case response-time bound R_{τ_i} of each task τ_i is not greater than its corresponding deadline d_{τ_i}, for all periods. The worst-case response-time $R^{CL}_{\tau_i}$ of a task τ_i, at each criticality level CL, is the greatest response-time of its jobs, where response-time of a job is the time interval from the job release to the job completion. To acquire the worst-case response-time of all tasks, the corresponding RTA for the selected scheduling policy is applied, as required by our system model. The worst-case delay of a high criticality task in LO-mode, denoted as $D^{L}_{hp(\tau_i)}$, due to the execution of higher priority tasks,

considering all periods, is given by $D^L_{hp(\tau_i)} = R^L_{\tau_i} - C^L_{\tau_i}$. The $D^L_{hp(\tau_i)}$ and $C^L_{\tau_i}$ are used to initialise the run-time controller variables at the beginning of each time period T_{τ_i}.

Algorithm 1: Run-time control mechanism.

1 **Function** *RTcontrol_start*($C^L_{\tau_i}$, $D^L_{hp(\tau_i)}$, k_{τ_i}, T_{τ_i})
2 **if** *(LO-mode is active)* **then** /* in low mode */
3 $RC^L_{\tau_i} = C^L_{\tau_i}$;
4 $RD^L_{hp(\tau_i)} = D^L_{hp(\tau_i)}$;
5 $RR^L_{\tau_i} = k_{\tau_i} * T_{\tau_i} + D^L_{hp(\tau_i)} + C^L_{\tau_i}$;
6 **Function** *RTcontrol_exec*(p_{τ_i}, STI_{τ_i}, $RR^L_{\tau_i}$, $RD^L_{hp(\tau_i)}$, $RC^L_{\tau_i}$)
7 **if** *(LO-mode is active)* **then** /* in low mode */
8 t = get_current_time();
9 $RC^L_{\tau_i}{}'$ = Compute_RWCET(p_{τ_i}, STI_{τ_i});
10 $RR^L_{\tau_i}{}' = t + RD^L_{hp(\tau_i)} + RC^L_{\tau_i}{}'$;
11 $DS^L = DS^L + (RR^L_{\tau_i} - RR^L_{\tau_i}{}')$;
12 $progress_{\tau_i}$ = get_total_execution(τ_i) + (t–get_latest_start_time(τ_i));
13 **if** ($progress_{\tau_i} >= C^L_{\tau_i}$) **then**
14 **if** ($DS^L < C_{ptp}$) **then**
15 signal(HI-mode);
16 **Function** *RTcontrol_end*($C^L_{\tau_i}$, $lphc(\tau_i)$)
17 **if** *(LO-mode is active)* **then** /* in low mode */
18 **for** $\tau_j \in lphc(\tau_i)$ **do**
19 $RD^L_{\tau_j} = RD^L_{\tau_j} - C^L_{\tau_i}$;

4.2 Run-time control for dynamic slack computation

During execution, the scheduler selects the job with the highest priority to be executed, until a higher priority job arrives. At an instrumentation point of the current active job, when the system is in LO-mode, the controller is executed, with the highest priority. Algorithm 1 depicts the run-time controller that implements our approach. It has three functionalities: i) at the beginning of the job execution, it performs initialization of a set of variables, ii) during job execution, it updates the slack based on its actual progress and verifies whether execution in LO-mode is still safe, otherwise it informs the scheduler for the mode-switch, and iii) when the job ends, the job contribution to the overall delay of lower priority jobs is removed.

Beginning of period (L. 1-5): As soon as the job is released at the beginning of a new period k_{τ_i}, the remaining WCET in LO-mode is initialized with the overall WCET in LO-mode, i.e., $RC^L_{\tau_i} = C^L_{\tau_i}$. The remaining worst-case delay in LO-mode is initialized with the value computed offline, $RD^L_{hp(\tau_i)} = D^L_{hp(\tau_i)}$. Then, the worst-case response-time bound for period k is computed as $RR^L_{\tau_i} = k_{\tau_i} * T_{\tau_i} + D^L_{hp(\tau_i)} + C^L_{\tau_i}$.

During execution (L. 6-15): The approach computes the new remaining WCET of the job of task τ_i in LO-mode at the current point, $RC^L_{\tau_i}{}'$. This corresponds to the WCET in LO-mode only of the code that remains to be executed, from point p_{τ_i} until the end of the job. For the computation of $RC^L_{\tau_i}{}'$ of

the running high criticality job, the proposed approach applies the approach of [20], leveraged for LO-mode. Algorithm 2 summarises the computation of $RC^L_{\tau_i}{}'$ at a point p_{τ_i}. The algorithm takes as input the instrumentation point p_{τ_i} along with its Structure and Timing Information (STI_{τ_i}), which includes the *type*, *level*, *head* and partial WCETs of the point, pre-computed during the design-time analysis of the high criticality task. To be able to compute the *RC*, without unrolling the code of the high criticality task, the computation is performed per level, with the help of the array RL_{τ_i}. A local level ll_{τ_i} is used to depict the current nested level of point p_{τ_i}, taking into account function calls and loops. The local level is computed by adding the $offset_{\tau_i}$ and the *level* of the point p_{τ_i} (L. 5). Note that, the $level[p_{\tau_i}]$ depicts the level of nested loops inside the ECFG of a function, by definition. The $offset_{\tau_i}$ provides the level that must be added, because of any occurred function call. Therefore, when a function entry point is observed (*C5* is true, L. 14), i.e., a function call occurs, we increase the offset with the level of the entry point (L. 15). When an exit point is observed (*C1* is true, L. 2), i.e., a function returns, we decrease the offset by the level of the entry point (L. 3). Then, the observation level $o_level_{\tau_i}$ is used to decide if we are traversing ECFG in a forward (*C2* or *C4* is true) or backward direction (*C3* is true). When the ECFG is traversed in a forward direction, the remaining WCET in local level ll_{τ_i}, $RL_{\tau_i}[ll_{\tau_i}]$, is computed by subtracting the partial WCET of the point p_{τ_i} from the remaining WCET computed on the previous local level (L. 7 and L. 11). By definition, the point in the previous local level is the head point of p_{τ_i}. When the ECFG is traversed backwardss, we are in a loop. Thus, we have reached the point that corresponds the condition statement of the loop and we subtract the partial WCET computed between any two iterations, $j-1$ and j (L. 9). In this way, the remaining WCET of the head point at local level $ll_{\tau_i}-1$ is updated accordingly, before entering the loop, where points have a local level equal to ll_{τ_i}. Note that, before execution, the initialisation is as follows: $RL_{\tau_i}[0] = C_{\tau_i}$ (the overall WCET of τ_i), the remaining elements of the array RL_{τ_i} to zero, $offset_{\tau_i} = 0$, $o_level_{\tau_i} = 0$, and $last_point_{\tau_i}[0] = start$.

Let's illustrate how the RWCET is computed for the example of Figure 6. At the first invocation of the controller at point c, $ll_{\tau_i} = 1$. Since $o_level_{\tau_i} = 0$, the graph is traversed in forward direction and the RWCET is given by $RL_{\tau_i}[1] = RL_{\tau_i}[0] - C_{\tau_i}[start\text{-}c]$. The rest of the variables are updated, i.e., $last_point_{\tau_i}[1] = c$ and $o_level_{\tau_i} = 1$. For the first invocation at point n_2, $ll_{\tau_i} = 2$. The graph is still traversed in forward direction and the RWCET is given by $RL_{\tau_i}[2] = RL_{\tau_i}[1] - C_{\tau_i}[c\text{-}n_2]$, $last_point_{\tau_i}[2] = n_2$ and $o_level_{\tau_i} = 2$. When c is invoked in the second iteration, $ll_{\tau_i} = 1$. Since $o_level_{\tau_i} < ll_{\tau_i}$ and the last point in this level was c, the graph is now traversed in backward direction. The RWCET is updated by $RL_{\tau_i}[1] = RL_{\tau_i}[1] - C_{\tau_i}[c^{j-1}\text{-}c^j)]$, $last_point_{\tau_i}[1] = c$ and $o_level_{\tau_i} = 1$. With this update, the RWCET will be correctly computed for the points inside the loop. The RWCET in the other points is computed in a similar way.

Then, the approach monitors the current time t using low level functions of the platform that allow us to read the processor clock. The new value of the worst-case response-time bound of job τ_i at point p_{τ_i} ($RR^L_{\tau_i}{}'$), is computed at run-time by adding to the current time t, the updated remaining WCET of task τ_i in LO-mode ($RC^L_{\tau_i}{}'$) and the remaining delay for task τ_i due to higher priority (both low and high criticality) tasks in LO-mode. The difference between the updated value, $RR^L_{\tau_i}{}'$, and the previously computed value, $RR^L_{\tau_i}$, provides in a safe way any new slack (positive or negative) created due to the progress made between two instrumentation points of the job. The overall slack DS^L is updated accordingly. The actual progress of the job is computed based on the total execution up this point, using information provided by the scheduler. More precisely, the $get_total_execution(\tau_i)$ denotes the sum of all previous execution fragments, where τ_i was preempted, and $get_latest_start_time(\tau_i)$ is the time instance that the current fragment of τ_i started execution. Note that, this value is used only to check whether the $C^L_{\tau_i}$ has been reached, and thus, to enable the safe condition that checks for mode-switch. The safety condition verifies

Algorithm 2: Run-time computation of remaining RC_{τ_i} at point p_{τ_i}.

```
1  Function Compute_RWCET(p_τi, STI_τi)
2      if (type[p_τi] ==F_EXIT||F_ENEX) then                    /* C1 */
3          offset_τi = offset_τi − level[p_τi];
4          o_level_τi = o_level_τi − 1;
5      ll_τi = offset_τi + level[p_τi];
6      if (o_level_τi < ll_τi) then                             /* C2 */
7          RL_τi[ll_τi] = RL_τi[ll_τi] − C_τi[head[p_τi]-p_τi];
8      else if (last_point_τi[ll_τi] == p_τi) then              /* C3 */
9          RL_τi[ll_τi] = RL_τi[ll_τi − 1] − C_τi[p_τi^(j−1), p_τi^j];
10     else                                                     /* C4 */
11         RL_τi[ll_τi] = RL_τi[ll_τi] − C_τi[head[p_τi]-p_τi];
12     last_point_τi[ll_τi] = p_τi;
13     o_level_τi = ll_τi;
14     if (type[p_τi] ==F_ENTRY||F_ENEX) then                   /* C5 */
15         offset_τi = offset_τi + level[p_τi]
16     return RL_τi[ll_τi];
```

if it is still safe to continue execution in LO-mode, i.e., whether enough slack exists to reach the next instrumentation point in the worst-case. Otherwise, we switch to HI-mode by suspending low criticality tasks.

End of execution (L. 16-19): When a task τ_i finishes, it informs the lower priority, high criticality, tasks $lphc(\tau_i)$, that have been preempted from the task τ_i. This is achieved by updating their worst-case delay, through subtraction of the overall WCET of τ_i in LO-mode, i.e., $C^L_{\tau_i}$.

Due to the instrumentation points inserted to the the high criticality task τ_i, a controller is invoked during execution and re-computes in a safe way the remaining WCET (RWCET), RC_{τ_i} at each point p_{τ_i}, based on the task progress. RC_{τ_i} is the C_{τ_i} of the rest of the code that remains to be executed from point p_{τ_i} until the high criticality task τ_i ends.

4.3 Safety

In this section, we prove that the proposed run-time approach is safe, i.e., tasks respect their deadlines and system execution is correct. We will follow a two-step approach to prove the safety of the proposed method, as the former will allow us to build up the safety proof of the latter. At step one, we consider the extreme case where each instruction of a high criticality task is also an instrumentation point, and thus, any preemption has to occur at an instrumentation point. Considering the maximum number of points is, of course, a mathematical artifact that would enable us to prove that the approach is safe in all cases and is not envisioned for application in real-life. At step two, having proven that the proposed method is safe for the extreme case, we will prove that by removing an instrumentation point, it only impacts the slack that is exposed between two consecutive points. This means that by removing *points*, less slack is computed, while the proposed approach remains safe for an arbitrary number of points.

To ease notation, for a given a point p_{τ_i}, we will refer to the previous visited point as $p_{\tau_i} - 1$ and to the next point as $p_{\tau_i} + 1$, even if they are the same instrumentation point, e.g. in a loop. This suffices, since the sequence of visited points, during execution, constitutes a linear execution

path, as if all loops were unrolled and all function calls were in-lined [19]. We will also refer to any execution between two points of a job as a segment.

4.3.1 Step 1: Each instruction corresponds to an instrumentation point.

Lemma 1. *For any job of a high-criticality task τ_o that is executed non-preemptively in LO-mode, if the previous jobs (of the same or other tasks) executed with precisely their worst-case behavior in LO-mode, then the proposed run-time control correctly signals a mode switch, when an overrun occurs, and τ_o always respects its deadline.*

Proof. According to L. 13-14 (Alg. 1), a mode switch will occur during the execution of job j of task τ_o, iff the progress of τ_o is greater or equal to its $C^L_{\tau_o}$ ($progress_{\tau_o} \geq C^L_{\tau_o}$) and the slack DS^L is less than the point-to-point overhead C_{ptp}. Assuming that the overrun occurs at some point $p^o_{\tau_o}$ of the unique points P_{τ_o} of task τ_o; at that point, the progress would be, based on L. 12, equal to the actual execution of all previously executed segments due to preemption (first term) plus the execution time, since the job was last scheduled (second term). Since the job overruns at point $p^o_{\tau_o}$, its actual execution time is greater than $C^L_{\tau_o}$, thus the condition $progress_{\tau_o} \geq C^L_{\tau_o}$ holds. Given two consecutive points $p, p+1$ the computed dynamic slack is (L. 10-11):

$$DS^{L\prime} = DS^L + (RR^L_{\tau_o} - RR^L_{\tau_o}{}') \Rightarrow \tag{1}$$

$$\Delta^{p+1}_p DS^L = \Delta^p_{p+1} t + \Delta^p_{p+1} RD^L_{hp(\tau_o)} + \Delta^p_{p+1} RC^L_{\tau_o} \tag{2}$$

where $\Delta^p_{p+1} x$ is difference operator between the value of the associated variable x at point p and its value at point $p+1$, e.g. $\Delta^p_{p+1} t = t_p - t_{p+1} = -\Delta^{p+1}_p t$. Applying Equation 2 for all pairs of consecutive points up to $p^o_{\tau_o}$ we establish the total slack as:

$$\sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^{p+1}_p DS^L = \sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \left(\Delta^p_{p+1} t + \Delta^p_{p+1} RD^L_{hp(\tau_o)} + \Delta^p_{p+1} RC^L_{\tau_o} \right) \Leftrightarrow \tag{3}$$

$$DS^L_{p^o_{\tau_o}} = DS^L_{p^s_{\tau_o}} + \sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^p_{p+1} t + \sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^p_{p+1} RD^L_{hp(\tau_o)} + \sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^p_{p+1} RC^L_{\tau_o} \tag{4}$$

where $DS^L_{p^s_{\tau_o}}$ is the available slack at the first point $p^s_{\tau_o}$ of the running task. Since all previous tasks executed with precisely their worst-case behavior in LO-mode, $DS^L_{p^s_{\tau_o}}$ is equal to zero. In addition, since the task is executed non-preemptively (assumption of this Lemma), the second sum, containing the remaining preemptions, equals zero. The first sum is equal to the progress of task τ_o. Note that, the value is negative, since in $\Delta^p_{p+1} t$, later time instances (with larger values) are subtracted from earlier time instances (with smaller values). The last sum equals to the partial WCET of task τ_o from the first point up to $p^o_{\tau_o}$, since $C^L_{\tau_o} = \sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^p_{p+1} RC^L_{\tau_o} + \sum_{p \in P_{\tau_o}}^{p \geq p^o_{\tau_o}} \Delta^p_{p+1} RC^L_{\tau_o}$. Thus, the last sum is clearly less than $C^L_{\tau_o}$ (assuming that the computation of remaining WCET (L. 9) is safe, which was proven in [19]). We therefore establish:

$$DS^L_{p^o_{\tau_o}} = -progress_{\tau_o} + \left(C^L_{\tau_o} - \sum_{p \in P_{\tau_o}}^{p \geq p^o_{\tau_o}} \left(\Delta^p_{p+1} RC^L_{\tau_o} \right) \right) \tag{5}$$

Since task τ_o has overrun, i.e. $progress_{\tau_o} \geq C^L_{\tau_o}$ it follows that:

$$DS^L_{p^o_{\tau_o}} \leq -C^L_{\tau_o} + C^L_{\tau_o} - \sum_{p \in P_{\tau_o}}^{p \geq p^o_{\tau_o}} \left(\Delta^p_{p+1} RC^L_{\tau_o}\right) \Rightarrow DS^L_{p^o_{\tau_o}} \leq 0 \tag{6}$$

which satisfies the condition $DS^L < C_{ptp}$ and, therefore, concludes the proof. □

Corollary 1. *Given two consecutive points $p, p+1$ of any job of task τ_i, that are executed non-preemptively, the approach will correctly signal an overrun if it occurs at point $p+1$.*

Having this fundamental guarantee, i.e., for two consecutive points, executed non-preemptively, the run-time control is safe, allows us to focus on what happens when preemptions and under-runs/overruns occur, collectively. As in this extreme case we assumed the maximum number of instrumentation points, some of these points will coincide with the preemption points. Thus, two consecutive instrumentation points of the same task are always executed non-preemptively or a pre-emeption occurred at the former point. In the following lemmas we prove each separate case with respect to the amount of slack available and whether a mode-switch has already occurred. All lemmas assume that a schedulable task-set $\mathcal{T}$ is given and that $p^o_{\tau_o}$ is the first point where some job of a high-criticality task overruns, with the response-time bound of that point $p^o_{\tau_o} \in P_{\tau_o}$ is $R^{CL}_{p^o_{\tau_o}}$ in CL-mode, with $CL \in \{H, L\}$ and the actual time at that point is $t_{p^o_{\tau_o}}$.

Lemma 2. *For any job of a high-criticality task τ_u that is executed non-preemptively in LO-mode and underruns by U, the computed slack is never greater than the underrun.*

Proof. Since the task τ_u is executed non-preemptively it suffices that the difference between the slack at the beginning of execution, i.e. $DS^L_{p^s_{\tau_u}}$, and at the end of execution, i.e. $DS^L_{p^e_{\tau_u}}$, is not greater than the underrun U, where $p^s_{\tau_u}, p^e_{\tau_u}$, are the instrumentation points at the beginning and end of the task, respectively. Applying Equation 4 for the end of execution we acquire:

$$DS^L_{p^e_{\tau_u}} = DS^L_{p^s_{\tau_e}} + \sum_{p \in P_{\tau_u}}^{p < p^e_{\tau_u}} \Delta^p_{p+1} t + \sum_{p \in P_{\tau_u}}^{p < p^e_{\tau_u}} \Delta^p_{p+1} RD^L_{hp(\tau_u)} + \sum_{p \in P_{\tau_u}}^{p < p^e_{\tau_u}} \Delta^p_{p+1} RC^L_{\tau_o} \tag{7}$$

The first sum is equal to the progress of task τ_u, which is equal to its WCET $C^L_{\tau_u}$ minus the amount of underrun U. Note that, the value is negative, since in $\Delta^p_{p+1} t$, later time instances (with larger values) are subtracted from earlier time instances (with smaller values). In addition, since the task is executed non-preemptively (assumption of this Lemma), the second sum, containing the remaining preemptions, equals zero. The last sum equals to the partial WCET of task τ_u from the first point $p^s_{\tau_u}$ up to $p^e_{\tau_u}$, i.e. its WCET $C^L_{\tau_u}$. Therefore, we establish:

$$DS^L_{p^e_{\tau_u}} - DS^L_{p^s_{\tau_e}} = -(C^L_{\tau_u} - U) + (0) + (C^L_{\tau_u}) \Rightarrow DS^L_{p^e_{\tau_u}} - DS^L_{p^s_{\tau_e}} = U \tag{8}$$

Thus, the computed slack is no greater than the amount of underrun U, which concludes this proof. □

Lemma 3. *Given the first point $p^o_{\tau_o}$, where some job of a high-criticality task overruns and there is not enough slack to reach the next point, a mode-switch will occur and the next point $p^o_{\tau_o}+1$ will respect its response-time bound in mode HI-mode, i.e. $t_{p^o_{\tau_o}+1} \leq R^H_{p^o_{\tau_o}+1_\tau}$, when the mode-switch is being controlled by the proposed approach.*

Proof. Since we assume the maximum number of points, the segment of the current job of task τ_o from $p^o_{\tau_o} - 1$ to $p^o_{\tau_o}$, is executed non-preemptively; if a preemption existed, it would be either at

$p^o_{\tau_o} - 1$ or $p^o_{\tau_o}$. Thus, according to Corollary 1, if there is not enough slack, the overrun would be correctly detected at point $p^o_{\tau_o}$ and since there is not enough slack a mode-switch will occur.

During the mode-switch, by definition of RTAs for mixed-critical systems, all points p that have a larger response-time bound at mode switch, i.e., $R^*_p \geq R^*_{p_{\tau_o}}$, will respect their response-time bound $t_p \leq R^*_p$, and thus, high-criticality tasks will meet their deadlines. Similarly, when the system has fully transitioned to HI-mode, high-criticality tasks will meet their deadlines, since the task-set has been deemed schedulable. □

Having established that the approach behaves correctly, when there is not enough slack, and high-criticallity tasks meet their deadlines, we now establish correctness in case there is enough slack, i.e. $DS^L_{p_{\tau_o}} \geq C_{ptp}$. An amount of slack existing at an overruning point $p^o_{\tau_o}$ can only be due two orthogonal reasons. The first reason is that the slack existed at the start of execution of the current job of τ_o which is addressed by Lemma 4. The other reason is due to some jobs of other tasks τ_u, that preempted the current job of τ_o, have completed their execution and underrun in LO-mode, prior to $t_{p^o_{\tau_o}}$, which is addressed by Lemma 5. Note that, since $p^o_{\tau_o}$ is the first point to overrun, its actual execution ($progress_{\tau_o}$) is no less than its WCET in LO-mode, $C^L_{\tau_o}$. Thus, any slack at point $p^o_{\tau_o}$ cannot occur because of any underrun of the current job of τ_o at some previous points.

Lemma 4. *Given the first point $p^o_{\tau_o}$, where some job of a high-criticality task overruns and there is enough slack to reach the next point, i.e. $DS^L_{p_{\tau_o}} \geq C_{ptp}$, and that slack existed at the start of execution of the current job of τ_o (denoted as point $p^s_{\tau_o}$), then no mode-switch will occur and all tasks will meet their deadlines if they do not overrun, when the mode-switch is being controlled by the proposed approach.*

Proof. Since all subsequent tasks are assumed not to overrun, it suffices to show that the computed slack at $p^o_{\tau_o}$ is not greater than the underrun of previous jobs (of the same or different tasks) at that point. This is sufficient as from the response time perspective, the overrun execution is equivalent to some other execution, where the underrun jobs of tasks τ_u executed for (at most) their WCET in LO-mode, and thus, the overrun job of task τ_o executes for less than its $C^L_{\tau_o}$ at the point of overrun $p^o_{\tau_o}$. The fact that the computed slack at $p^o_{\tau_o}$ is not greater than the underrun of previous jobs is nevertheless given by Lemma 2. □

Lemma 5. *Given the first point $p^o_{\tau_o}$, where some job of a high-criticality task overruns and there is enough slack to reach the next point, i.e. $DS^L_{p_{\tau_o}} \geq C_{ptp}$, and that slack was created by underruning jobs that preempted τ_o, then no mode-switch will occur and all tasks will meet their deadlines if they do not overrun, when the mode-switch is being controlled by the proposed approach.*

Proof. Since all subsequent tasks are assumed not to overrun, it suffices to show that the computed slack at $p^o_{\tau_o}$ is not greater than the underrun at that point. This is sufficient as from the response time perspective, the overrun execution is equivalent to some other execution, where the underrun jobs of tasks τ_u executed for (at most) their WCET in LO-mode, and thus, the overrun job of task τ_o executes for less than its $C^L_{\tau_o}$ at the point of overrun $p^o_{\tau_o}$. Following Equation 4, considering preemptions, the sum $\sum_{p \in P_{\tau_o}}^{p < p^o_{\tau_o}} \Delta^p_{p+1} t$ contains both the progress of the job of task τ_o and the actual progress of the job of tasks τ_u that preempted τ_o. Since the job of task τ_o overrun at point $p^o_{\tau_o}$, its progress is equal to its WCET in LO-mode, i.e. $C^L_{\tau_o}$. Let $hpjf(\tau_o)$, be the set of jobs that preempted

the job of τ_o and finished, i.e., $hpjf(\tau_o) \subseteq hp(\tau_o)$. We establish:

$$\sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^p_{p+1} t = -\sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^{p+1}_p t = -\left(C^L_{\tau_o} + \sum_{j_{\tau_u} \in hpjf(\tau_o)} \sum_{p \in P_{\tau_u}} \Delta^p_{p+1} t \right) = \tag{9}$$

$$= -C^L_{\tau_o} - \left(\left(\sum_{j_{\tau_u} \in hpjf(\tau_o)} C^L_{\tau_u} \right) - U_p \right) = -C^L_{\tau_o} - \left(\sum_{j_{\tau_u} \in hpjf(\tau_o)} C^L_{\tau_u} \right) + U_p \tag{10}$$

where $U_p \geq 0$ is the total amount of underrun, if any, of the jobs that preempted the current job of τ_o. Thus, the computed slack is established as (Equation 4):

$$DS^L_{p^o_{\tau_o}} = DS^L_{p^s_{\tau_o}} - C^L_{\tau_o} - \left(\sum_{j_{\tau_u} \in hpjf(\tau_o)} C^L_{\tau_u} \right) + U_p + \sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^p_{p+1} RD^L_{hp(\tau_o)} + \left(C^L_{\tau_o} - \sum_{p \in P_{\tau_o}}^{p \geq p^o_{\tau_o}} \Delta^p_{p+1} RC^L_{\tau_o} \right) \Rightarrow \tag{11}$$

$$DS^L_{p^o_{\tau_o}} = U_p + DS^L_{p^s_{\tau_o}} + \left(- \sum_{j_{\tau_u} \in hpjf(\tau_o)} C^L_{\tau_u} + \sum_{p \in P_{\tau_o}}^{p<p^o_{\tau_o}} \Delta^p_{p+1} RD^L_{hp(\tau_o)} \right) - \sum_{p \in P_{\tau_o}}^{p \geq p^o_{\tau_o}} \Delta^p_{p+1} RC^L_{\tau_o} \tag{12}$$

According to Algorithm 1 (L. 16-19) when a job finishes execution, it subtracts its WCET from the remaining delay of its lower priority, high criticality tasks that it has preempted. Thus, the jobs $j_{\tau_u} \in hpjf(\tau_o)$ that preempted the current job of τ_o, have subtracted their WCET from $RD_{hp(\tau_o)}$, hence the terms in parenthesis cancel each other. The last sum of Equation 12 is greater than zero, since it is the partial WCET of τ_o from the start point up to point $p^o_{\tau_o}$. Thus, the slack $DS^L_{p^o_{\tau_o}}$ at point $p^o_{\tau_o}$ is no greater than any underrun that was created during execution of τ_o plus the slack $DS^L_{p^s_{\tau_o}}$ that τ_o started with. Therefore, we have shown that the execution is equivalent, from a response time perspective, to an execution where the underrun jobs executed for at most their WCET in LO-mode and the overrun job of task τ_o executed for less than its $C^L_{\tau_o}$ at the point of overrun, thus proving that the next point $p^o_{\tau_o} + 1$ will respect its response-time bound. □

Theorem 6 (Time safety). *Given a schedulable task-set $\mathcal{T}$ where the mode switch is being controlled by the proposed approach, the tasks always respect their deadlines.*

Proof. In Lemma 4 and Lemma 5 it was proven that the execution to the first point of overrun is safe, i.e. either a mode switch will occur or there is enough slack to proceed to the next point. In addition, the execution in case of enough slack is equivalent, in terms of time, to an execution where the under-running task(s) executed for at most their WCET in LO-mode and the overrunning task τ_o executed for less than its $C^L_{\tau_o}$ at the point of overrun. As a result, that point can be consider as not having an overrun and some other future point will be the first to overrun. By iterative application of those Lemmata, until a mode switch occurs or the end of execution, to all points, we conclude our proof. □

4.3.2 *Step 2: Impact of removing instrumentation points.*

Following, we prove that removing points preserves safety of the proposed approach and it only impacts the exposed slack.

Theorem 7. *Any task τ_i will meet its deadline, if some instrumentation points are removed.*

Proof. Applying Equation 2 to three consecutive points $p-1, p, p+1$ of a high-criticality task, we establish:

$$\Delta_{p-1}^{p} DS^L = \Delta_{p}^{p-1} t + \Delta_{p}^{p-1} RD_{hp(\tau_i)}^L + \Delta_{p}^{p-1} RC_{\tau_i}^L \tag{13}$$

$$\Delta_{p}^{p+1} DS^L = \Delta_{p+1}^{p} t + \Delta_{p+1}^{p} RD_{hp(\tau_i)}^L + \Delta_{p+1}^{p} RC_{\tau_i}^L \tag{14}$$

By adding the equations together, the computed slack, if point p did not exist, is:

$$\Delta_{p-1}^{p+1} DS^L = \Delta_{p+1}^{p-1} t + \Delta_{p+1}^{p-1} RD_{hp(\tau_i)}^L + \Delta_{p+1}^{p-1} RC_{\tau_i}^L \tag{15}$$

This effectively states that the slack computation remains the same, when a point is removed, and becomes visible only at the next point. Since the control algorithm decides if there is enough slack to reach the next point (L. 14) based on the point-to-point WCET overhead C_{ptp}, Lemma 1 and Theorem 6 still applies, hence the response-time bound cannot increase. Thus, by iteratively removing points, we prove that any assignment of instrumentation points is safe. □

5 EXPERIMENTAL EVALUATION

To evaluate the proposed approach (RRT), we compare the run-time control overhead and the performance, with respect to the mode-switch decisions and the execution of the low criticality tasks, with the: i) BaseLine (BL) approach, which switches mode when a high criticality job exceeds its C^L, such as [3, 5, 10], and ii) Dynamic (DYN) approach, which observes the dynamic slack, but only after a job finishes, such as [7, 17, 23].

5.1 Run-time control overhead

For the timing overhead, we implemented the controllers of RRT, DYN and BL approaches on the on the TMS320C6678 chip (TMS) of Texas Instrument [29] (Table 4). The overhead in cycles, depicted in Table 5, is obtained by applying a measurement-based approach. The measurements have been obtained using the processor's local timer during the controller execution and we have followed the approach of multiple executions, where each controller is executed 50 times and we maintain the largest observed value.

Table 4. TMS platform (8 DSP cores).

Instr./cycle	Freq.	Instr. L1	Data L1	L2	L3	DDR3
8	1 GHz	32 KB	32 KB	512 KB	4 MB	512 MB

To obtain the overall overhead, we compute how many times the actions of each mechanism will be performed, which depends on the task set and the actual execution.

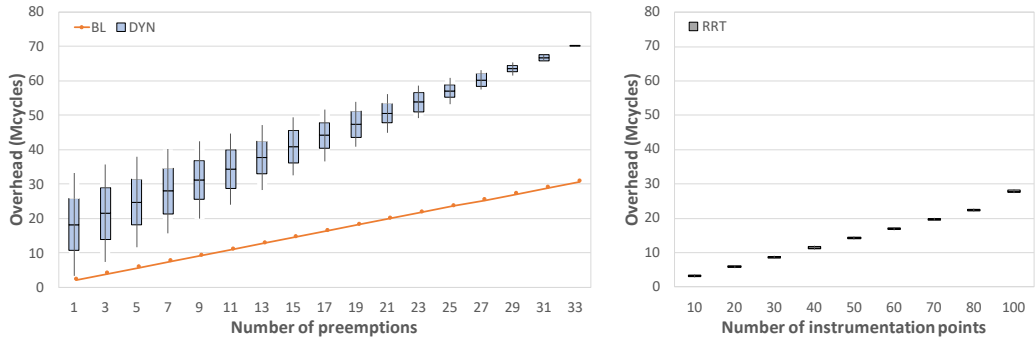
- For the BL approach, a timer must be set, before the task starts execution, in order to trigger an interrupt when C^L has been reached, and check for mode-switch [18]. The timer is set in the beginning of a job execution (895 cycles) and there is a single time moment when the BL controller will be executed (192 cycles), i.e., when the timer is triggered. In this overhead, we need to add the cost due to potential preemptions. Each time the job is preempted, its timer has to be paused, as a new timer will be set with the WCET in low mode of the higher priority job that will be now executed. When the execution returns to the preempted job, the timer needs to be resumed with the WCET in low mode (895 cycles) of the preempted job. Thus, the overhead of the BL mechanism per job is given by $O_{BL} = (895 + 192) + 895 * \#preemptions = 1,087 + 895 * \#preemptions$. The minimum overhead is given when the job is not preempted. For the maximum overhead, the worst-case is when a high-criticality job, that has the lower priority, is preempted sequentially

(i.e., no nested preemption takes place). In the worst-case, the number of preemption is equal to the number of jobs, that have a higher priority. Figure 7a depicts the cost of BL mechanism based on the number of preemptions.

- For the DYN approach, on top of the overhead due to preemptions (similar to BL), the controller may be evoked many times in order to check for the mode-switch. The number of possible evocations depends on how many of the higher priority tasks, that have preempted the task, have created slack, and whether the slack has been used to extend the WCET in low mode, each time. When a higher priority task finishes execution, it updates the dynamic slack (45 cycles). When the timer is triggered, the DYN controller is executed (252 cycles). If slack exists, it is used to extend the WCET in low mode (895 cycles to re-set the timer). The overhead of the DYN mechanism per task is given by $O_{DYN} = (895 + 252) + (895 + 45) * \#preemptions + (252 + 895) * \#extensions = 1,147 + 940 * \#preemptions + 1,147 * \#extensions$. Similar to BL, the minimum overhead is when the task is not preempted. For the maximum overhead, the worst-case is when all tasks with higher priorities, thus they preempted the task, created slack, and when between two preemptions, the task has reached the mode-switch point and extended the WCET in low, but not enough so as the task can finish execution. Figure 7a gives the DYN cost as box plot, considering the minimum and the maximum possible extensions for each number of preemptions.
- On the contrary, the proposed approach does not require the use of such a timer, that requires to be paused and resume during job preemptions, inserting high overhead. The RRT overhead depends on how many times the controller is evoked and it is exactly bounded by the number of instrumentation points. When the job finishes execution, the preemption delay of the high criticality low priority jobs ($lphc$), that has preempted, is updated. The RRT overhead is given by $O_{RRT} = 42 + 274 * \#points + 28 * \#lphc$. The maximum cost is given when all the points evoke the controller, i.e., no mode-switch took place, and only one low criticality task exist in the task set, and thus, the high priority high criticality task will have to update, at the end of execution, all the low priority high criticality tasks. Figure 7b depicts in box plots (considering 1 up to 31 low priority high criticality tasks that need to be updated) for 10 up to 100 instrumentation points.

Table 5. Overhead of run-time control (cycles).

	Start		Execution		End	
App.	**Action**	**Cyc.**	**Action**	**Cyc.**	**Action**	**Cyc.**
BL	Set timer	895	Interrupt	162	N/A	-
			Decide mode-switch	30		
			Total	192		
DYN	Set timer	895	Interrupt	162	Update DS^L	48
			Use DS^L to extend $C^L_{\tau_i}$	90		
			Re-set timer	895		
			Total	1147		
RRT	Initialise	42	$RC^L_{\tau_i}$ and $progress_{\tau_i}$	152	Update RD_{τ_j}	28 per τ_j
			Monitor t	32		
			$RR^L_{\tau_i}$ and DS^L	45		
			Decide mode-switch	45		
			Total	274		



(a) BL and DYN (b) RRT

Fig. 7. Overhead comparison

5.2 Performance evaluation

For the performance evaluation, we developed a simulation framework where we can tune different parameters of the task set in order to perform extensive evaluation experiments. We compare the decisions taken by RRT and DYN with the BL decisions, by computing how many times RRT and DYN decided to i) Not Switch (NS), same as BL, ii) Switch at the Same (SS) job as BL did, iii) Switch at another job, executed Later (SL) compared to BL, and iv) Switch Avoided (SA), compared to BL. Furthermore, we compare the percentage of the low criticality jobs (among all experiments) that i) did not start execution, and ii) finished.

Benchmark characterisation: To obtain realistic values for the simulation set-up in order to perform the above comparison, e.g., regarding the C^L, C^H, number of instrumentation points, partial WCETs, and actual execution time of jobs, we execute benchmarks with different characteristics on the TMS under different platform configurations. We used the Discrete Cosine Transformation (DCT), Mergesort (MERGE), and Fast Fourier Transformation (FFT) from StreamIT benchmarks [30], as they are typical kernels for real-time systems, exhibiting different characteristics in terms of memory accesses and execution paths. The source code of each benchmark has been modified in order to implement the instrumentation points, following the approach presented in Section 4.1, and executed on the TMS platform. The instrumentation points have been inserted uniformly in the benchmark codes. The number of instrumentation points per benchmark has been selected in order to have the total controller cost around 1% of the benchmark's WCET, in the worst-case, based on controller overhead measured on the TMS platform (Table 5). The measurements have been obtained using the processor's local timer during the bechmark execution.

Our first set of experiments obtains estimations of the C^L and the partial WCETs between instrumentation points. To achieve that, the sources that variate the execution time have to be eliminated [15], by disabling data-caches, removing interferences (i.e., disabling all but one processor) and providing, as much as possible, values as input data that potentially enforce the worst-case path. We have followed the approach of multiple executions, where each benchmark has been executed 50 times, for each different input. We observed a deviation of $\sim$ 1% between measurements among the executions of one input. We maintain the largest observed value among all inputs as the C^L value. The C^H is obtained by applying a common practise of inserting a margin of 30%.

Our second set of experiments characterizes the behavior of the actual execution time of the benchmarks under different software and hardware scenarios, and compute the observed timing variability. We studied two parameters that typically affect the execution time, i.e., the different

execution paths due to the inputs and the benchmark and the use of caches of the hardware platform. We tune each parameter independently in order to characterize its impact to the execution time. Table 6 shows the observed variability, computed by comparing the execution time of the best observed value and the worst observed value (which is given by C^L in Table 6). On the one hand, we observe that the impact of caches in execution time is similar for all benchmarks, with 71.14% on average. On the other hand, we observe that the impact of different execution paths is benchmark depended; it is higher for applications with several execution paths, e.g., DCT, and, smaller for single-path applications, e.g., FFT. The obtained values depicted in Table 6 will be used to drive the simulation set-up with realistic values.

Table 6. Benchmark characterisation

Instr. points (Number)

DCT	MERGE	FFT
25	17	10

WCET (cycles)

	DCT	MERGE	FFT
C^L	981,120	669,026	275,891
C^H	1,177,344	802,832	331,070

Cache variability

Path	DCT	MERGE	FFT
Best-Path	73.83%	69.03%	69.40%
Worst-Path	76.57%	68.60%	69.38%

Path variability

Caches	DCT	MERGE	FFT
Disabled	46.65%	12.84%	0.15%
Enabled	40.51%	14.69%	0.46%

Experimental set-up: The experimental simulation framework consists of i) a task-set generator, ii) an offline analysis, iii) an online scheduler, and iv) an online controller. The task-set generator constructs schedulable task-sets with realistic parameters based on the benchmark characterization, i.e., C^L, C^H, number of instrumentation points, and partial WCETs. The instrumentation points are generated in a uniform way, following the instrumentation of the benchmark codes. The offline analysis provides the worst-case delay in LO-mode, due to higher priority tasks. The online scheduler dynamically selects the ready task with the highest priority to be executed, taking into account the mode of execution. The online controller implements each approach to be evaluated. If the online controller decides mode-switch, it informs the online scheduler to drop low criticality tasks.

We perform experiments by generating task-set sizes from 2 tasks up to 40 tasks, consisting of an equal number of high and low criticality tasks. For each task set size, we perform a number of experiments. At each experiment, the task-set generator creates a different task-set using the measured information provided in Table 6. For each task, it selects the C^L randomly in the range [275,891-981,120], the number of points randomly in the range [10,25], the period computed considering a system utilisation $U = 70\%$, with $U = \sum_{\tau_i \in \mathcal{T}} \frac{C^L_{\tau_i}}{P_{\tau_i}}$, and a unique task priority given by Rate Monotonic, where the shorter the task period, the higher the job priority [2]. The timing overhead of the controller is included in the C^L. Note that, each task invokes a number of jobs. Among all experiments, the minimum, average and maximum number of simulated jobs is 22, 835, and 1,787.

For each experiment, the actual execution time of low criticality tasks is equal to their C^L. The actual execution time of high criticality tasks is tuned based on the benchmark characterisation. Two configurations are evaluated:

- Cache-related configuration, based on the trend of the observed timing variability regarding caches, i.e., similar variability for all benchmarks. Thus, the actual execution time of high criticality tasks is tuned in a similar way. Per task-set size, we perform 10 experiments. At each experiment,

the actual execution time among two points is varied almost exhaustively, from $C^L - 40\% \times C^L$ up to $C^L + 30\% \times C^L$, with a step of 5%, based on the observed variability.

- Path-related configuration, based on the trend of the observed timing variability regarding execution paths, i.e., the variability due to execution paths is benchmark-dependent. Thus, the actual execution time of each high criticality tasks is tuned independently. Per task-set size, we performed 1,000 experiments. At each experiment, the actual execution time among two points is randomly given in the range $[C^L - 50\% \times C^L, C^L + 50\% \times C^L]$, based on the observed variability.

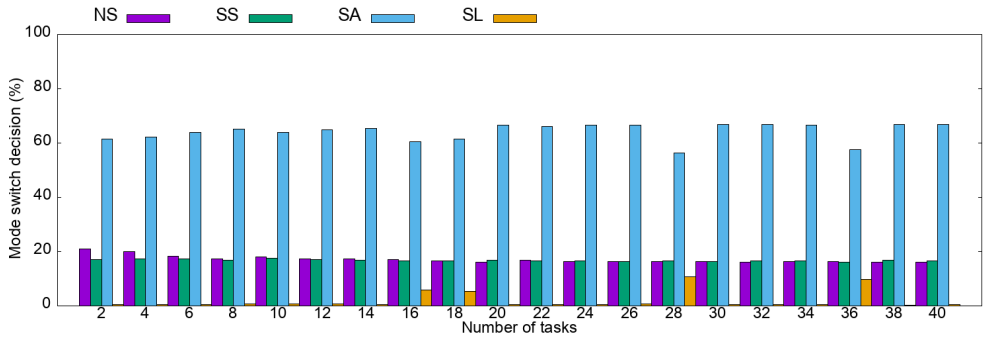
Cache-related configuration: Fig. 8 shows the mode-switch decisions and the number of non started and finished low criticality tasks. We observe that the proposed approach (Fig. 8a) is able to avoid the mode-switch (SA) in a significant number of experiments, outperforming the DYN approach (Fig. 8b). When the actual execution time of all tasks is lower than the C^L (avg. in 17.1% of the experiments), RRT and DYN have the same behavior with BL, since mode-switch does not occur. For the majority of the experiments, DYN, most of the time decides to switch mode at the same time as BL approach, i.e., on average, 78.46% of the experiments. On the contrary, RRT behaves as BL for only 17.78% of the experiments. Compared to BL, RRT is able to avoid mode-switch in 64.13% of the experiments. DYN avoids mode-switch for only 3.46% of the experiments. Hence, RRT managed to increase the mode-switch avoidance by a factor of ×18.54, compared to DYN. In the rest of the experiments, the approaches perform mode-switch later than BL; 1.99% for RRT and 0.98% for DYN. Regarding the execution of low criticality task, DYN behaves quite similarly to BL. DYN manages to start execution 2.67% more low criticality tasks, compared to the number of low criticality tasks that started execution with BL. Moreover, with DYN manages to finish, on average, 20.9% of the total number of low criticality jobs, compared to 17.5% for the BL. This provides a gain of a factor of ×1.19 more low criticality jobs that managed to finish their execution, compared to the number of low criticality jobs that finished in BL. On the contrary, due to the fine-grained slack exploitation, RRT provides significant improvements, as 82.60% of the total low criticality jobs managed to finish execution. This provides a gain factor of ×4.72, compared to the number of low criticality tasks that finished execution with BL.

Path-related configuration: As shows Fig. 9, on average, RRT switches as BL in 50.42% of the experiments, while DYN in 63.26%. For the remaining experiments, RRT avoids mode-switch (SA) in 36.07% of the experiments (Fig. 9a), while DYN in 11.03% (Fig. 9b). As a result, RRT increases mode-switch avoidance by ×3.27. Furthermore, we observe that DYN decides to switch mode later than BL in 25.7% of the experiments, while RRT in 13.5%. This difference comes from the fact in 12.20% of these experiments RRT totally avoided mode-switch, whereas DYN performed mode-switch, but later that BL. We also explore the impact of these decisions in the execution of the low criticality tasks. On average, for DYN, 13.50% low criticality jobs finished their execution, which is 12.28% more jobs than BL. However, RRT managed to finish 38.09% low criticality jobs, corresponding to 36.82% more jobs than BL.

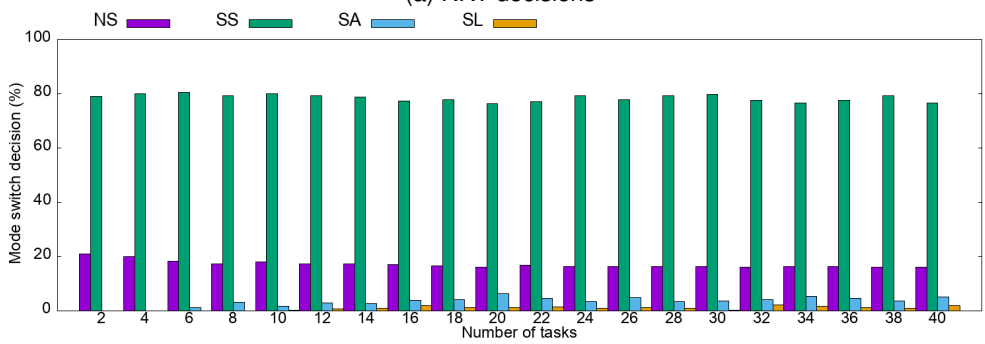
6 RELATED WORK

We briefly describe the representative mixed-criticality approaches, relevant to our work, which are summarised in Table 7. A detailed survey is available in [12].

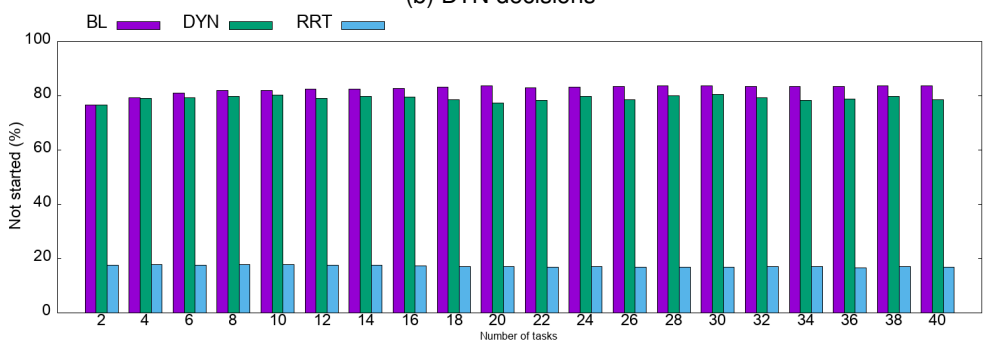
The majority of existing works use static decisions regarding when the mode switch occurs, typically given by the value of the low criticality WCET of high criticality tasks. Upon mode switch, low-criticality jobs are dropped, e.g., in time-triggered scheduling [10], EDF with Virtual Deadlines scheduling [3] and priority based on Adaptive Mixed Criticality [5] on uniprocessors, global [21] and partitioned [6] Earliest Deadline First (EDF)-based scheduling and global fixed-priority scheduling [25] on multiprocessors. However, as the mode switch decision is defined



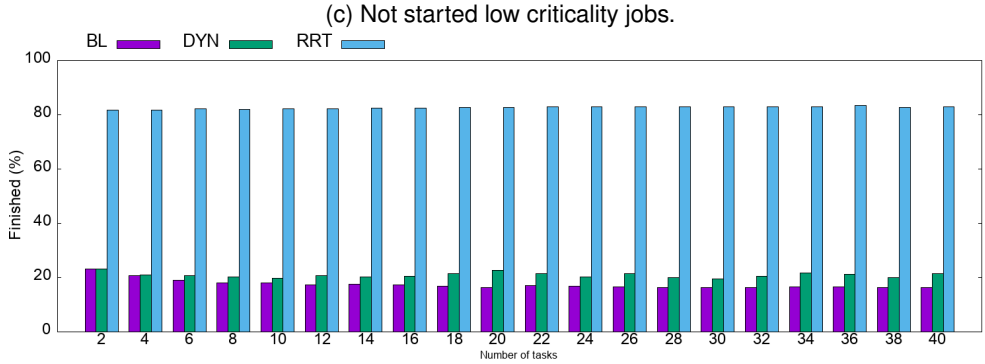
(a) RRT decisions



(b) DYN decisions

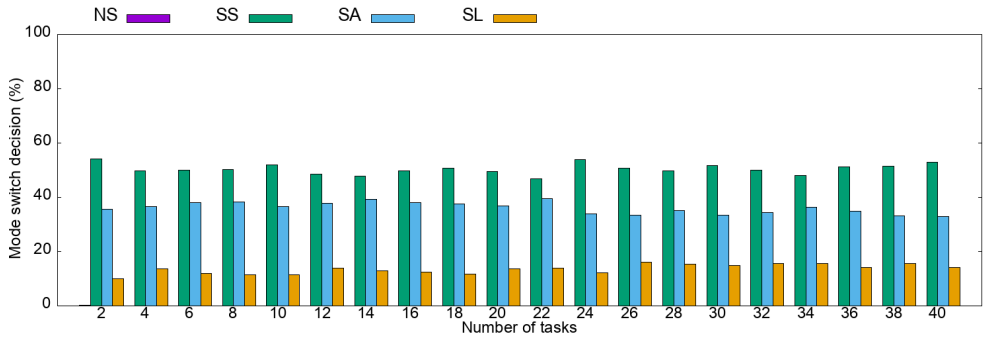


(c) Not started low criticality jobs.

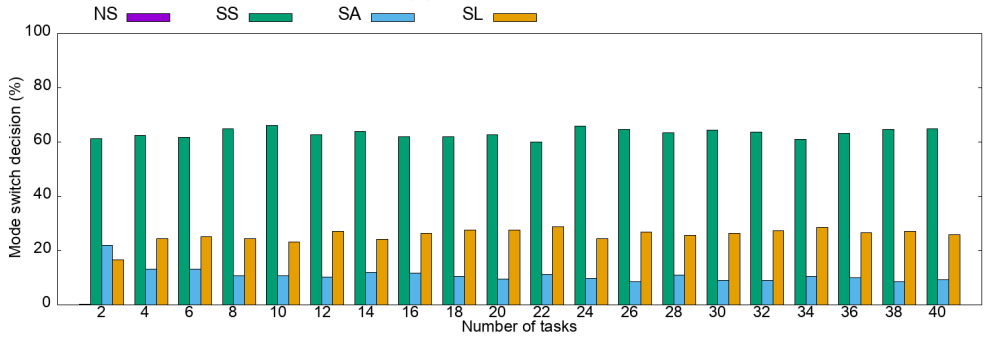


(d) Finished low criticality jobs.

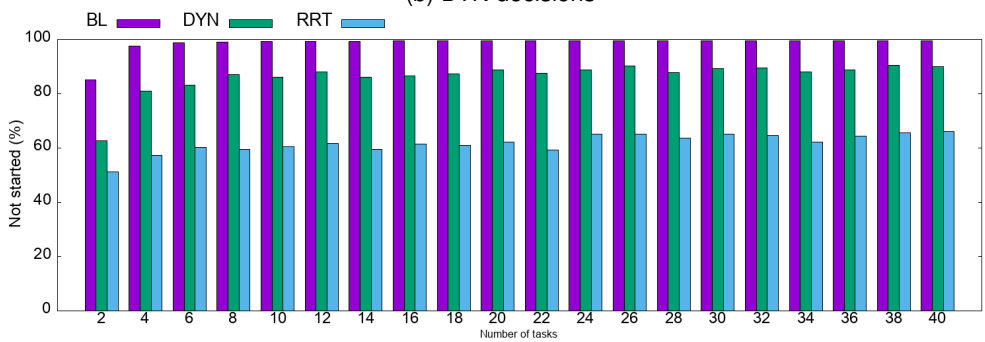
Fig. 8. Cache-related configuration.



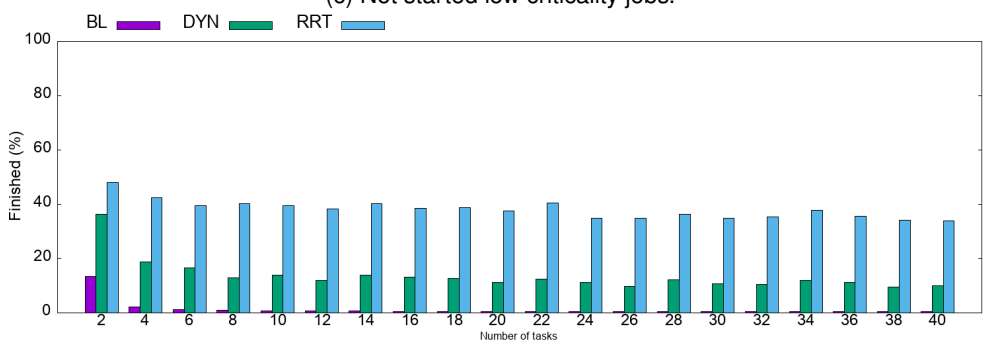
(a) RRT decisions



(b) DYN decisions



(c) Not started low criticality jobs.



(d) Finished low criticality jobs.

Fig. 9. Path-related configuration.

Table 7. Comparison with representative State-of-the-Art

Ref.	When to switch				In HI-mode			
	Low crit. WCET	Static Slack	Dynamic Slack	Dynamic Delay	Drop low	Reduce priority	Timing budget	Extend Periods
[3, 5, 10] [6, 21, 25]	√				√			
[1, 16, 22]	√				√			
[13, 14, 27]		√			√			
[17, 24]		√	√		√			
[7]		√	√		√			
[23]		√	√				√	
[28]				√	√			
[9]	√		√			√	√	√
Proposed			√		√			

statically, while the low criticality tasks are dropped in HI-mode, the system performance is degraded. To increase the execution time of the low criticality tasks, existing works i) explore other strategies, than dropping low criticality tasks, in HI-mode, and ii) explore, statically or dynamically, ways to postpone the mode switch or switch back to LO-mode. In summary, first category approaches i) set the priority of low criticality tasks below the priority of any high criticality task, ii) reduce the execution time requirements of low criticality tasks in high criticality mode, and iii) extend the periods of low criticality tasks [9]. We will focus on the second category, where the proposed approach belongs to.

Static approaches determine the largest value, that could be added to a task's low criticality WCET, postponing the mode switch, such that the whole task set remains schedulable. Methods inspired by sensitivity analysis [8] compute offline the margins that low criticality tasks are allowed to overrun before being suspended, on uniprocessor systems [27]. Other approaches use zero-slack scheduling, where low-critical tasks are allowed to run until the zero-slack of higher-criticality jobs is finished. The zero-slack of a task is offline computed based on the low criticality and high criticality WCET and the interference from higher-priority and higher-criticality tasks [13, 14]. However, as those techniques exploit only static slacks, usually due to the system being under-loaded, they use the run-time information to further postpone the mode switch.

Dynamic approaches mainly exploit the slack, created during execution due to earlier-than-the-WCET execution of the tasks. The most straightforward way to use the dynamic slack is to enable switching back to LO-mode, after the execution of high criticality tasks [1, 16, 22]. The remaining approaches use the dynamic slack to postpone the mode switch. For instance, a single overrun budget is used for high and low criticality tasks for EDF scheduling. It is based on the feasible task procrastination timelength, updated using the run-time information about the completion times of tasks [17]. In [24], idle slacks are inserted during scheduling for low criticality tasks, and an online approach explores the slack created due to early termination of tasks. Through the bailout protocol [7], high criticality tasks, that overrun their low criticality WCET, ask for funds, putting the system in bailout mode. Tasks, finished early, donate their slack and low criticality task, released in bailout mode, are abandoned, donating their low criticality WCET. When the bailout fund is zero, the system enters recovery mode. The adaptMC [23] exploits this dynamic slack through a control feedback mechanism that run-time updates the task budgets. Our work extends the state-of-the-art by exploring, not only the slack created due to the early termination of tasks, but also the slack

due to the current execution progress of active tasks, by safely computing, during execution, the response time of the running job.

Few approaches exploit, the dynamic slack based on the progress of a task, by computing the i) observed delay [28], and ii) remaining WCET [19, 20]. The first approach inserts a checkpoint to a high criticality task and profiles the average execution time up to that point. The actual time is compared to the profiled value to compute the observed delay. A schedulability analysis is applied, during execution, to check whether the system remains schedulable, in case the low criticality WCET of the task is extended by the observed delay [28]. Our approach is based on safe remaining WCET, instead of average values, without the need of applying a schedulability test or a response time analysis during execution. The second approaches insert several checkpoints in a high criticality task. At each checkpoint, it safely computes the remaining WCET, i.e., the WCET of the code, that remains to be executed, from the observation point until the end. The WCET is used to decide to postpone mode switch, considering interference from low criticality tasks executed in parallel on a multicore [19, 20]. However, this approach considers that each core runs only a single task non-preemptively and high and low criticality tasks are executed in different processors. Our work extends the approach of [19, 20] for multi-periodic high and low criticality tasks executed pre-emptively on the same processor.

7 CONCLUSION

This work proposes a safe and lightweight approach that at run-time exposes the dynamic slack, created as the execution of jobs progresses, and exploits it in order to postpone and avoid mode switch in mixed-criticality systems. The proposed approach operates upon the novel concepts of run-time computation of the worst case response time, removing the requirement for performing a schedulability test or a response time analysis, at run-time. We have shown formally that the proposed approach is safe. From the obtained results, our approach is able, on average, to avoid mode switch in 50.10% of the experiments, which allows to 60.34% low criticality tasks to finish execution.

REFERENCES

[1] James H. Anderson, Sanjoy K. Baruah, and Björn B. Brandenburg. 2009. Multicore Operating-System Support for Mixed Criticality. In *Int'l Workshop on Mixed Criticality Systems (WMC)*.

[2] Neil C Audsley. 1991. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Citeseer.

[3] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. 2011. Mixed-Criticality Scheduling of Sporadic Task Systems. In *European Symposium on Algorithms (ESA)*, Vol. 6942. 555–566.

[4] S.K. Baruah, L. Haohan, and L. Stougie. 2010. Towards the Design of Certifiable Mixed-criticality Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, USA, 13–22.

[5] S. K. Baruah, A. Burns, and R. I. Davis. 2011. Response-Time Analysis for Mixed Criticality Systems. In *Real-Time Systems Symposium (RTSS)*. 34–43.

[6] Sanjoy K. Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. 2013. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems* (2013), 1–36.

[7] I. Bate, A. Burns, and R. I. Davis. 2015. A Bailout Protocol for Mixed Criticality Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 259–268.

[8] E. Bini, M. Di Natale, and G. Buttazzo. 2006. Sensitivity analysis for fixed-priority real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 10 pp.–22.

[9] Alan Burns and B. Baruah. 2013. Towards A More Practical Model for Mixed Criticality Systems. In *Real-Time Systems Symposium (RTSS)*.

[10] Alan Burns and Sanjoy K. Baruah. 2011. Timing Faults and Mixed Criticality Systems. In *Dependable and Historic Computing (Lecture Notes in Computer Science, Vol. 6875)*, CliffB. Jones and JohnL. Lloyd (Eds.). Springer Berlin Heidelberg, 147–166.

[11] Alan Burns and Robert Davis. 2020. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep* (2020).
[12] Alan Burns and Robert I. Davis. 2018. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv. (CSUR)* 50, 6 (2018), 82:1–82:37.
[13] D. d. Niz, K. Lakshmanan, and R. Rajkumar. 2009. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Real-Time Systems Symposium (RTSS)*. 291–300.
[14] D. de Niz and L. T. X. Phan. 2014. Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 111–122.
[15] Jean-François Deverge and Isabelle Puaut. 2007. Safe measurement-based WCET estimation. In *Int'l Workshop on Worst-Case Execution Time Analysis (WCET)*, Reinhard Wilhelm (Ed.), Vol. 1.
[16] Tom Fleming and Alan Burns. 2013. Extending Mixed Criticality Scheduling. In *Real-Time Systems Symposium (RTSS)*.
[17] Biao Hu, Kai Huang, Pengcheng Huang, Lothar Thiele, and Alois Knoll. 2016. On-the-fly fast overrun budgeting for mixed-criticality systems. In *Int'l Conf. Embedded Software (EMSOFT)*. 1–10.
[18] Zhe Jiang, Kecheng Yang, Nathan Fisher, N. Audsley, and Zheng Dong. 2020. Pythia-MCS: Enabling Quarter-Clairvoyance in I/O-Driven Mixed-Criticality Systems. In *RTSS*.
[19] Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, and Matthieu Roy. 2014. Run-time Control to Increase Task Parallelism in Mixed-Critical Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*.
[20] Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. 2014. Distributed Run-time WCET Controller for Concurrent Critical Tasks in Mixed-critical Systems. In *Int'l Conf. Real Time and Networks Systems (RTNS)*. Article 139, 10 pages.
[21] Haohan Li and S. Baruah. 2012. Global Mixed-Criticality Scheduling on Multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 166–175.
[22] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. 2010. Mixed-Criticality Real-Time Scheduling for Multicore Systems.. In *International Conference on Computer and Information Technology (CIT)*. 1864–1871.
[23] Alessandro Vittorio Papadopoulos, Enrico Bini, Sanjoy Baruah, and Alan Burns. 2018. AdaptMC: A Control-Theoretic Approach for Achieving Resilience in Mixed-Criticality Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, Vol. 106. 14:1–14:22.
[24] T. Park and S. Kim. 2011. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *ACM Int'l Conf. Embedded Software (EMSOFT)*. 253–262.
[25] Risat M. Pathan. 2012. Schedulability Analysis of Mixed-Criticality Systems on Multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 309–320.
[26] SAE. 2010. Aerospace Recommended Practices 4754a - Development of Civil Aircraft and Systems. SAE.
[27] F. Santy, L. George, P. Thierry, and J. Goossens. 2012. Relaxing Mixed-Criticality Scheduling Strictness for Task Sets Scheduled with FP. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 155–165.
[28] Soham Sinha and Richard West. 2020. PAStime: Progress-aware Scheduling for Time-critical Computing. In *Euromicro Conference on Real-Time Systems (ECRTS)*.
[29] Texas Instruments. 2013. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor*. Technical Report SPRS691D. TI.
[30] William Thies et al. 2010. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *PACT*. 12.
[31] S. Vestal. 2007. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Real-Time Systems Symposium (RTSS)*. 239–243.