HAL archives-ouvertes.fr

Run-time Control to Increase Task Parallelism in Mixed-Critical Systems

Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, Matthieu Roy

To cite this version:

Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, Matthieu Roy. Run-time Control to Increase Task Parallelism in Mixed-Critical Systems. 26th Euromicro Conference on Real-Time Systems (ECRTS14), Jul 2014, Madrid, Spain. 11p. hal-01015476

HAL Id: hal-01015476

https://hal.archives-ouvertes.fr/hal-01015476

Submitted on 26 Jun 2014

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.

Run-time Control to Increase Task Parallelism in Mixed-Critical Systems

Angeliki Kritikakou*, Olivier Baldellon†‡, Claire Pagetti*, Christine Rochange§, Matthieu Roy†
* ONERA-Toulouse, France
† CNRS, LAAS, Toulouse, France
‡ University of Toulouse, IRIT, Toulouse, France

***Abstract*—Although multi/many-core platforms enable the parallel execution of tasks, the sharing of resources may lead to long WCETs that fail to meet the real-time constraints of the system. Then, a safe solution is the execution of the most critical tasks in isolation followed by the execution of the remaining tasks. To improve the system performance, we propose an approach where a critical task can run in parallel with less critical tasks, as long as the real-time constraints are met. When no further interferences can be tolerated, the proposed run-time control suspends the low critical tasks until the termination of the critical task. In this paper, we describe the design and prove the correctness of our approach. To do so, a graph grammar is defined to formally model the critical task as a set of control flow graphs on which a safe partial WCET analysis is applied and used at run-time to control the safe execution of the critical task.**

I. INTRODUCTION

A. Context

The chip market moves towards multi/many-core systems due to increased system requirements and power dissipation issues of single-core systems [1]. As these systems offer massive computing power, a higher integration of applications is performed in the same platform. The integrated applications have diverse characteristics which create *mixed-critical systems* [2]. A *mixed-critical system* consists of applications with different levels of criticality. The criticality level of an application depends on the potential consequences on the system in case the application fails to meet its timing constraints. The Design Assurance Level (DAL) model [3] defines the *hard real-time* applications with high criticality levels A, B or C and the *soft real-time* applications with low criticality levels D or E.

Applications with high criticality level require strict guarantees on their correct execution. To ensure these guarantees, real-time task scheduling techniques should use a safe estimation of the Worst-Case Execution Time (WCET) [4]. Several WCET estimation techniques exist, but static analysis tools [5], such as AIT or OTAWA, are recommended for high criticality applications. Unfortunately, many/multi-core systems have a dynamic difficult-to-predict behavior. More precisely, the concurrent accesses to the shared resources introduce timing variations, e.g. in the communication network and in the memory hierarchy with variable delays under concurrent requests. Therefore, the effects of possible task interferences have to be upper bounded to guarantee real-time response, usually by assuming full contention under concurrent requests. The result is a safe but pessimistic WCET, in the sense that the cases leading to the worst case scenarios are unlikely to occur. This leads to over-allocating resources to high criticality applications and it may even be the cause of the system unschedulability.

B. Motivation

Let us consider $n+1$ independent synchronous tasks $\mathcal{T} = \{\tau_C, \tau_1, \ldots, \tau_n\}$ where τ_C is a periodic task of high criticality level (DAL A, B or C), period T_C and deadline D_C; τ_i are tasks of low criticality level (DAL D or E). A partitioned scheduling is applied where each task is executed on a different core.

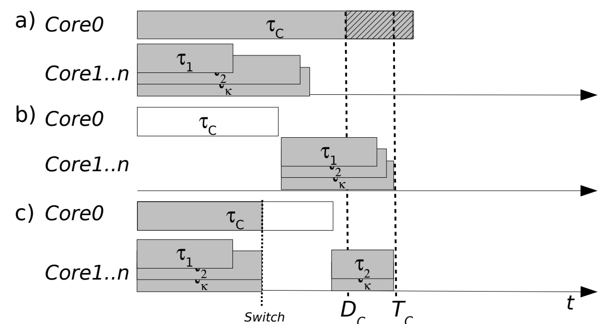

Fig. 1. Mixed-critical schedules scenarios

Two scenarios for the WCET computation of τ_C are considered: 1) maximum load (max): all tasks run in parallel, and 2) isolation (iso): only τ_C runs on the system. In max scenario, we assume that due to the resource sharing, the WCET of τ_C is above the deadline, i.e. $\mathsf{WCET}_{\mathsf{max}} > D_C$, as depicted in Fig. 1(a). In this case, the hard real-time constraints cannot be met. Existing mixed-critical scheduling approaches, such as [6], [7], [8], assume that the task set is schedulable at least in the highest criticality level, and thus are not directly applicable. Then, a safe solution is to execute τ_C in isolation. When the critical task terminates and if time slack exists, the low criticality tasks are executed. In this case, no conflict occurs with the low criticality tasks. Hence, the WCET is significantly lower and the critical task respects its deadline, i.e. $\mathsf{WCET}_{\mathsf{iso}} \leq D_C$, as shown in Fig. 1(b).

Our goal is to increase the task parallelism and to reduce the over-provisioning of resources by combining the benefits and discarding the drawbacks of the previous cases. To achieve that, the low criticality tasks are allowed to run in parallel with the critical task, as long as it is safe. At run-time, if the interferences may lead to a deadline miss of τ_C, the low criticality tasks are suspended until the termination of τ_C. If

time slack exists, the low criticality tasks are resumed. In this way, the critical task is guaranteed to meet its deadline, whereas the low criticality tasks run in parallel improving the resources utilization, as shown in Fig. 1(c).

C. Proposed methodology and contributions

This optimistic mixed-critical schedule can be achieved using an appropriate run-time control mechanism, as proposed by our methodology. We introduce a set of *observation points* to enable the run-time (online) monitoring of the timing behavior of the critical task and the control of task set scenarios. At each observation point, a *safety condition* is applied to check whether it is still safe to continue the execution of τ_C in the maximum load scenario. The safety condition uses the remaining WCET of the critical task in isolation scenario, which is run-time computed by our low-overhead algorithm. If the safety condition evaluates that a risk exists of overloading the system and, thus, the critical task runs too slow, a backup process is applied to guarantee the real-time response of τ_C: the low criticality tasks are suspended and τ_C runs in isolation. When the critical task finishes its execution and if time remains until the next release of τ_C, the low criticality tasks are resumed. We prove the correctness of the proposed safety condition and the low-overhead run-time algorithm for computing the remaining WCET.

As the computation of the remaining WCET is time consuming, it cannot be performed at run-time. Hence, we propose a run-time algorithm based on pre-computed data which reflect the program structure (static analysis). Then, the run-time computation involves only basic arithmetic and reflects the actual progress of the critical task execution. To achieve this goal, during the design-time (offline) analysis, the critical task is represented by a set of Extended Control Flow Graphs (ECFGs) with observation points. We propose a graph grammar to formally describe the set of ECFGs under study and to prove the correctness of our run-time computation algorithm. Based on the obtained ECFGs, a safe WCET analysis is applied for the pre-computation of several partial remaining WCETs used by the run-time control.

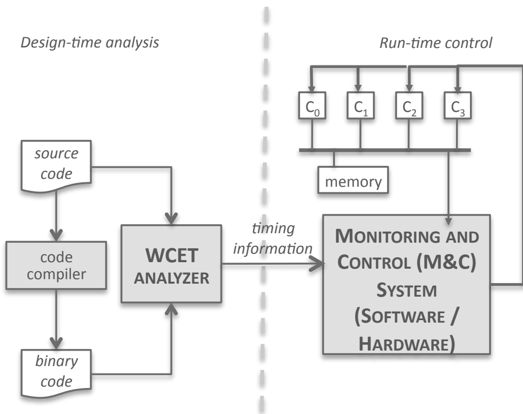

Fig. 2. Overall methodology

The proposed methodology consists of the design-time analysis to pre-compute the required data and the task scenario control applied at run-time, as shown in Fig. 2. A brief description of the general idea is detailed in [9], whereas in this paper, we focus on the design, the formal description and the proof of the proposed methodology. We also give evaluation results based on simulations. The proposed run-time control can be implemented in the target system as a software or as a hardware component, which is our future work.

The remaining of the paper is organized as follows: Section II presents the proof of our guarantee on the critical task real-time response. Section III formally presents the design-time analysis. Section IV describes and proves the run-time control. Section V demonstrates how the proposed methodology is applied in a detailed case study and presents several experimental results. Section VI presents the related work on mixed-critical systems. Section VII concludes this study.

II. GUARANTEE OF CRITICAL TASK REAL-TIME RESPONSE

The proposed methodology initially executes the maximum load scenario and at each observation point of τ_C checks whether the low criticality tasks should be suspended. Hence, the following statement should be proved: "The switching from the maximum load scenario to the isolation scenario guarantees that the critical task meets the hard real-time deadline D_{C}."

The task set scenario switching occurs when the following safety condition does not hold:

$$\mathsf{RWCET}_{\mathsf{iso}}(x) + \mathsf{RWCET}_{\mathsf{max,PTP}} + t_{\mathrm{RT}} \leq D_{\mathrm{C}} - \mathsf{ET}(x) \quad (1)$$

where $\mathsf{RWCET}_{\mathsf{iso}}(x)$ is the remaining WCET of τ_C at an observation point x in the isolation scenario, $\mathsf{RWCET}_{\mathsf{max,PTP}}$ is the WCET until the next observation point, t_{RT} is the total time of the proposed run-time control mechanism and $\mathsf{ET}(x)$ is the real execution time of τ_C until point x. The t_{RT} is the sum of: 1) t_{Mon} (the overhead to monitor the real execution time), 2) t_{Cnt} (the WCET of the run-time control), and 3) t_{SW} (the WCET overhead due to scenario switching).

Theorem 1. *If $\mathsf{WCET}_{\mathsf{iso}} \leq D_C$, then for any execution with the proposed run-time control, τ_C always respects its deadline.*

Proof: If τ_C is executed in isolation, by definition $\mathsf{WCET}_{\mathsf{iso}} \leq D_C$. Let us assume that τ_C starts its execution in the maximum load scenario until point p_{i+1}. For two consecutive observation points p_i and p_{i+1}, we have:

$$\mathsf{ET}(p_{i+1}) - \mathsf{ET}(p_i) \leq \mathsf{RWCET}_{\mathsf{max,PTP}} \quad (2)$$

$$0 \leq \mathsf{RWCET}_{\mathsf{iso}}(p_i) - \mathsf{RWCET}_{\mathsf{iso}}(p_{i+1}) \quad (3)$$

Since the execution continues in the maximum load scenario until p_{i+1}, it means that p_i fulfilled the safety condition 1:

$$\mathsf{ET}(p_i) + t_{\mathrm{RT}} + \mathsf{RWCET}_{\mathsf{max,PTP}} + \mathsf{RWCET}_{\mathsf{iso}}(p_i) \leq D_{\mathrm{C}}$$

The remaining execution from p_{i+1} is safe if τ_C can be switched in the isolation scenario and ends before its deadline, therefore we have to show that:

$$\mathsf{ET}(p_{i+1}) + t_{\mathrm{RT}} + \mathsf{RWCET}_{\mathsf{iso}}(p_{i+1}) \leq D_{\mathrm{C}}$$

Thanks to (2), $\mathsf{ET}(p_{i+1}) + t_{\mathrm{RT}} + \mathsf{RWCET}_{\mathsf{iso}}(p_{i+1}) \leq \mathsf{ET}(p_i) + \mathsf{RWCET}_{\mathsf{max,PTP}} + t_{\mathrm{RT}} + \mathsf{RWCET}_{\mathsf{iso}}(p_{i+1})$. Because the safety condition (1) holds in p_i, we obtain: $\mathsf{ET}(p_{i+1}) + t_{\mathrm{RT}} + \mathsf{RWCET}_{\mathsf{iso}}(p_{i+1}) \leq D_C + \mathsf{RWCET}_{\mathsf{iso}}(p_{i+1}) - \mathsf{RWCET}_{\mathsf{iso}}(p_i)$ Thanks to (3), $\mathsf{RWCET}_{\mathsf{iso}}(p_{i+1}) - \mathsf{RWCET}_{\mathsf{iso}}(p_i) \leq 0$, hence we conclude that $\mathsf{ET}(p_{i+1}) + t_{\mathrm{RT}} + \mathsf{RWCET}_{\mathsf{iso}}(p_{i+1}) \leq D_C$, i.e., the critical task terminates in time. ∎

III. Design-time analysis

This section describes the design-time analysis of the critical task and the computation of parameters used by the run-time control mechanism.

A. Extended Control Flow Graph Representation

A graph grammar is proposed to model the critical task τ_C considered under study. The critical task τ_C is described by the syntax of Table I, which covers a wide range of applications. From the binary code of the critical task τ_C [10], we create a set of *control flow graphs* (CFGs), where we insert observation points. The CFGs obtained from the abstract syntax of Table I and compiled without optimizations are covered by the proposed grammar.

TABLE I. Application model syntax

<table>
<tr><th colspan="3">Syntax rules</th></tr>
<tr><td>term</td><td>::=</td><td><constant> | <variable></td></tr>
<tr><td>expr</td><td>::=</td><td><term> | <term> <operator> <term> | <unary-expr></td></tr>
<tr><td>unary-expr</td><td>::=</td><td><variable> <unary-operator> | <unary-operator> <variable></td></tr>
<tr><td>cond-expr</td><td>::=</td><td><expr> <conditional-operator> <expr></td></tr>
<tr><td>assignment</td><td>::=</td><td><unary-expr> <assignment-operator> <expr></td></tr>
<tr><td>instruction</td><td>::=</td><td><assignment> | <unary-expr> | <>;</td></tr>
<tr><td>stat</td><td>::=</td><td><instruction> | <stat>;<stat> | if (<cond-expr>) then <stat1> else <stat2> | for (expr1; cond-expr; expr2) <stat> | <function-call> ;</td></tr>
<tr><td>function-call</td><td>::=</td><td><return-type> functionName(<parameter-list>) <stat> return <expr-return>;</td></tr>
<tr><td>program</td><td>::=</td><td><function-call></td></tr>
</table>

Definition 1 (Critical task τ_C)**.** *A* critical task τ_C *is a set of functions* $\mathcal{S} = \{F_0, F_1, ..., F_n\}$*, with* F_0 *the main function. Each* function *is represented by an Extended CFG (ECFG).*

Definition 2 (Observation point)**.** *An observation point is a check point in an ECFG where the run-time control is executed. A special observation point named* start *is defined before the starting of the execution.*

Definition 3 (Extended Control Flow Graph)**.** *An ECFG is a CFG annotated with observation points. The ECFG of a function* F *is a directed graph* $G = (V, E)$*, consisting of*

1) A finite set of nodes V *composed of 5 disjoint sub-sets* $V = \mathcal{N} \cup \mathcal{C} \cup \mathcal{F} \cup \{IN\} \cup \{OUT\}$ *where,*

- $N \in \mathcal{N}$ *represents a binary instruction or a block of binary instructions (Fig. 3(a)),*
- $C \in \mathcal{C}$ *represents the block of binary instructions of a condition statement (Fig. 3(b)),*
- $F_i \in \mathcal{F}$ *represents the binary instructions of calling a function* F_i *and links the node with the ECFG of the function* F_i *(Fig. 3(c)),*
- IN *is the input node with an observation point* in *(Fig. 3(d)),*
- OUT *is the output node with an observation point* out *(Fig. 3(e)).*

Every node $v \in V$ *corresponds to a terminal node of our grammar with one unique input observation point before the execution of the first binary instruction. We name the terminal nodes with upper letters and the observation point with lower letter based on the node type.*

2) a finite set of edges $E \subseteq V \times V$ *representing the control flow between nodes.*

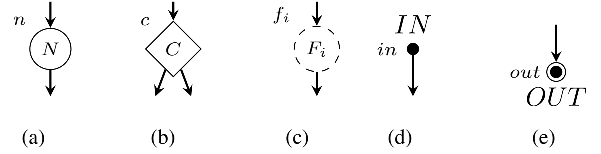

Fig. 3. Schematic representation of terminal grammar nodes.

Definition 4 (ECFG syntax)**.** *The ECFG syntax is defined by the following graph grammar, where the terminal nodes are the nodes of ECFG and the non-terminal node is* B*. The grammar rules are:*

- *A function* F_i *has exactly one input node and one output node, with* B *a non-terminal node of the grammar (Fig. 4),*
- *A non-terminal node* B *is derived as (Fig. 5):*
  1) *An empty node (Fig. 5(b)),*
  2) *A single node* N *(Fig. 5(c)),*
  3) *A sequential component, i.e. the concatenation of non-terminal nodes (Fig. 5(d)),*
  4) *An if-then-else component, i.e. the concatenation of a* C *conditional node with two mutually executed paths that end to the same non-terminal node (Fig. 5(e)),*
  5) *A loop component, i.e. the concatenation of a loop condition* C *with two mutually executed paths, one with the empty node and one with the repetition of the loop kernel (Fig. 5(f)),*
  6) *A function call node* F_i *(Fig. 5(g)).*
- *During every derivation, the possibly multiple incoming links of* B *are "glued" to the upper block, and the outgoing link is glued to the lower block, with lower and upper referring to the graphical representation of Fig. 5*

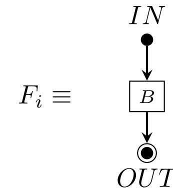

Fig. 4. Representation of function F_i.

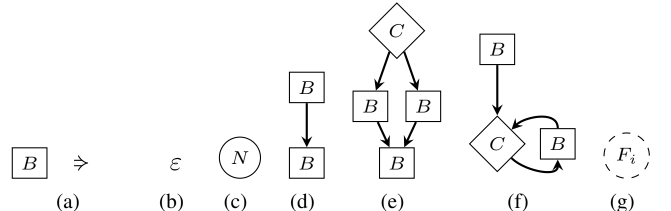

Fig. 5. Schematic representation of rewritting grammar rules

Definition 5 (Disjoint graphs)**.** *Two ECFG graphs $G_i = (V_i, E_i)$ and $G_j = (V_j, E_j)$ are disjoint if $V_i \cap V_j = \emptyset$ and $E_i \cap E_j = \emptyset$.*

The function call nodes link the disjoint ECFGs associated to the different functions. The Fig. 6 describes a task composed of two functions $\mathcal{S} = \{F_0, F_1\}$. In Fig 6(a), F_1 is sequential, while in Fig. 6(b) F_1 is recursive.

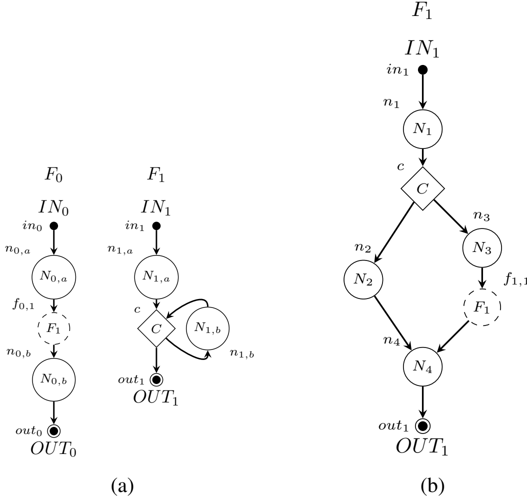

Fig. 6. Disjoints ECFGs associated to $\mathcal{S} = \{F_0, F_1\}$

Definition 6 (Walk)**.** *A walk $W = (IN, v_1, \ldots, v_n, OUT)$ in an ECFG is defined as a finite list of nodes such that $(v_i, v_{i+1}) \in E$, $(IN, v_1) \in E$ and $(v_n, OUT) \in E$.*

A walk is defined on a unique ECFG, while at run-time a complete execution over the different ECFGs is observed.

Definition 7 (Execution of τ_C)**.** *Let us consider the task $\mathcal{S} = \{F_0, ..., F_n\}$ and the associated ECFGs $G_0,\ldots G_n$. An execution P is a list of nodes $\in V_0 \cup \ldots \cup V_n$ obtained by inserting walks over the ECFGs. An execution is obtained inductively as follows:*

- *Initially, P=W_o, where W_o is a walk over the ECFG of F_0, $W_o = (IN_0, v_1, \ldots, OUT_0)$.*
- *Let P=$(IN_0, \ldots, v_i, v_{i+1}, \ldots, OUT_0)$. If $v_i = F_k$ and $v_{i+1} \neq IN_k$, then P is rewritten as $P = (IN_0, \ldots, v_i, W_{F_k}, v_{i+1}, \ldots, OUT_0)$ where W_{F_k} is a walk over G_k. The* entry function point *is the point of node v_i. The* exit function point *is the point of node v_{i+1}. The exit and entry points are sequential in ECFG of v_i. The type of a point x is denoted by* type$(x) \in \{F_ENTRY, F_EXIT, F_ENEX, -\}$*, where F_ENEX refers to the case where a point is both an entry and an exit point.*

Following an execution, we obtain the sequence of visited nodes of the ECFGs and thus the sequence of the observation points. An observation point may be visited several times due to the loop components and the function calls. Hence, we define the nested level of an observation point to distinguish between different visits of the same point.

Definition 8 (Nested level)**.** *The nested level* level *(x) of an observation point x in an ECFG is defined based on the grammar syntax as:*

- *Initialization point: The level of the initialization point start is 0,* level(start)=0.
- *Input node: The level of the input point in is 1,* level(in)=1.
- *Output node: The level of the output point out is 1,* level(out)=1.
- *Sequential component: The levels of two sequential points b_1 and b_2 are equal,* level(b_1)=level(b_2).
- *If-then-else component: The levels of all the points c, b_t, b_f and b_o are equal, i.e.* level(c)=level(b_t) =level(b_f)=level(b_o).
- *Loop component: The level of the initial block b_i is equal to the level of the conditional node c, i.e.* level(b_i)=level(c). *The level of the loop body point b is equal to the level of the loop condition c incremented by one,* level(b)=level(c)+1.

As the nested levels are design-time determined per ECFG, we define the head points to deal with the real execution of the critical task. The head points show when a function has been called and where a loop exists in each ECFG.

Definition 9 (Head point)**.** *Let us consider the task $\mathcal{S} = \{F_0, ..., F_n\}$ with the associated ECFGs $G_0,\ldots G_n$ and an execution $P = (IN_0, v_1, \ldots, OUT_0)$. The head points are defined as follows:*

- head(in_0)=start
- *let us assume that we computed* head(v_i).
  - *if $v_{i+1} = IN_k$, then* $\text{head}(v_{i+1}) = v_i$
  - *if $v_i = OUT_k$, then* $\text{head}(v_{i+1}) = \text{head}(\text{head}(v_i))$
  - *if $(v_i, v_{i+1}) \in E_k$ (i.e. are in the same ECFG):*
    - *if v_i is the condition of a loop,* $\text{head}(v_{i+1}) = v_i$
    - *otherwise* $\text{head}(v_{i+1}) = \text{head}(v_i)$

For instance, in Fig. 6(a) the P=$(IN_0, N_{0,a}, F_1, IN_1, N_{1,a}, C, N_{1,b}, C, OUT_1, N_{0,b}, OUT_0)$ is an execution of the task. In a similar way, an execution over a recursive function in Fig. 6(b) is $P = (IN_0, N_{0,a}, F_1, IN_1, N_1, C, N_3, F_1, IN_1, N_1, C, N_2, N_4, OUT_1, N_4, OUT_1, N_{0,b}, OUT_0)$. Table II provides the nested levels and the head points for the observation points of Fig. 6 for these executions.

B. Remaining WCET analysis

The remaining WCET of the critical task heavily depends on the real execution of the critical task. Hence, at design-time, we process the ECFGs and compute partial WCETs using safe static WCET analysis, similar to [5], but extended and adapted to our methodology. The WCET is computed by writing an Integer Linear Programming (ILP) formulation to express the program execution time as the combination of the individual times of the grammar components weighted by their execution counts. This expression is maximized to find the WCET, with

TABLE II. NESTED LEVEL AND HEAD POINTS OF FIG. 6

Observation point x	*level* (x)	*type* (x)	*head* (x)	
Initialization				
start	0	-	-	
F_0				
in_0	1	-	*start*	
$n_{0,a}$	1	-	*start*	
$f_{0,1}$	1	F_ENTRY	*start*	
$n_{0,b}$	1	F_EXIT	*start*	
out_0	1	-	*start*	
F_1 Fig. 6(a)				
in_1	1	-	$f_{0,1}$	
$n_{1,a}$	1	-	$f_{0,1}$	
c	1	-	$f_{0,1}$	
$n_{1,b}$	2	-	c	
out_1	1	-	$f_{0,1}$	
F_1 Fig. 6(b)				
			1st visit	2nd visit
in_1	1	-	$f_{0,1}$	$f_{1,1}$
n_1	1	-	$f_{0,1}$	$f_{1,1}$
c	1	-	$f_{0,1}$	$f_{1,1}$
n_3	2	-	c	
$f_{1,1}$	2	F_ENTRY	c	
n_2	1	-	$f_{1,1}$	-
n_4	1	F_EXIT	$f_{1,1}$	$f_{0,1}$
out_1	1	-	$f_{1,1}$	$f_{0,1}$

a number of constraints that reflect flow facts, e.g. loop bounds and unfeasible paths.

Our WCET analysis is based on computing the remaining WCET from one observation point x until the end of the program, $\mathsf{RWCET_y}(x)$, where $y \in \{\text{iso,max}\}$. When the $\mathsf{RWCET_{max}}(x)$ is computed, we consider that interferences occur from the parallel tasks, whereas when the $\mathsf{RWCET_{iso}}(x)$ is computed no interferences are taken into account. When point x is the entry of the critical task, i.e. *start* of F_0, $\mathsf{RWCET_y}(\textit{start})$ is the total $\mathsf{WCET}_{\tau_{i,y}}$. When point x is inside the ECFG, we compute the remaining WCET by using constraints to prohibit the execution of all the blocks which do not belong to any path from point x until the end of the ECFG.

Using the remaining WCET analysis of an observation point x, we can compute remaining WCETs between an observation point and its head point.

Definition 10 ($d_{head(x)-x}$)**.** *$d_{\text{head}(x)-x}$ is the maximum time from* head(x) *to x.*

$$d_{\text{head}(x)-x} = \mathit{RWCET_{iso}}(\text{head}(x)) - \mathit{RWCET_{iso}}(x)$$

Definition 11 (Loop component)**.** *When $d_{\text{head}(x)-x}$ is computed inside a loop component with conditional node c and n number of iterations, no time variation exists between iterations.*

$$d_{c-x} = \mathit{RWCET_{iso}}(c, j) - \mathit{RWCET_{iso}}(c, j), \forall j \le n$$

w_c is the time between any two consecutive iterations j and $j+1$ of the conditional node c.

$$w_c = \mathit{RWCET_{iso}}(c, j) - \mathit{RWCET_{iso}}(c, j+1), \forall j \le n$$

When a function is called from different points, the paths to the different points may result to several partial remaining WCETs. The existence of several paths may introduce time variability in the computation of $d_{head(x)-x}$ of the function points x with level 1. For instance, point n_1 in Fig. 6(a) may have different values $d_{f_{0,1}-n_1}$ and $d_{f_{1,1}-n_1}$. A trade-off exists between storing these time-variations or a unique partial information that permits the computation of a local upper bound for these points, i.e. the minimum remaining time observed from any common function call. For instance, in Fig. 6(b), the OTAWA tool computes $\mathsf{RWCET_{iso}}(f_{0,1}) = 50$, $\mathsf{RWCET_{iso}}(n_1) = 48$, $\mathsf{RWCET_{iso}}(f_{1,1}) = 45$, $\mathsf{RWCET_{iso}}(n_1) = 40$. By using the minimum time, we store $d_{head(n_1)-n_1} = 2$ and at run-time we obtain $\mathsf{RWCET_{iso}}(f_{0,1}) = 50$, $\mathsf{RWCET_{iso}}(n_1) = 48$, $\mathsf{RWCET_{iso}}(f_{1,1}) = 45$, $\mathsf{RWCET_{iso}}(n_1) = 43$.

To guarantee that the critical task deadline is always met, we must ensure that enough time is available to perform the scenario switch at the next observation point. Hence, we apply our remaining WCET analysis to compute the $\mathsf{RWCET_{max,PTP}}$ between any two consecutive observation points of the critical task in the maximum load scenario.

$$\mathsf{RWCET_{max,PTP}} = max_{x,x'}(\mathsf{RWCET_{max}}(x) - \mathsf{RWCET_{max}}(x'))$$

To support the run-time $\mathsf{RWCET_{iso}}(x)$ computation, we store in memory the *level*(x) and $d_{head(x)-x}$ for each point x. If x is the head point of a loop component, the w_x is also stored. If x is a function call entry or exit point, the type of the point is stored as well. An example of the data stored after the design-time analysis is given in Table III of Section V-A for our case study.

IV. RUN-TIME CONTROL

At each observation point x, the safety condition of Section II decides the switching between scenarios. As the $\mathsf{RWCET_{iso}}(x)$ is modified at each observation point, we propose a low-overhead algorithm to run-time compute this value by efficiently reusing the $\mathsf{RWCET_{iso}}$ of the head points. In Section IV-A we describe and formally prove the basic version of our algorithm applied when the critical task consists of a single function, i.e. $\mathcal{S} = \{F_0\}$. In Section IV-B we present and formally prove our extended version applied when the critical task consists of a finite set of functions, i.e. $\mathcal{S} = \{F_0, \ldots, F_n\}$.

A. Basic version

1) Algorithm description: The run-time computation of $\mathsf{RWCET_{iso}}(x)$ is depicted in Alg. 1. The pre-computed data are stored in the memory as constant arrays: *level* for the level and d and w for the partial WCETs. The algorithm maintains two local values o_level (for the previous observed level) and ll for the local level. For the basic version, ll is always equal to *level*(x). At run-time, the algorithm stores in array $last_point$ the last observed point and in array R the computed $\mathsf{RWCET_{iso}}$ per level.

ALGORITHM 1: Basic version.

Pre-computed data: level, w, d
Input: x
Data: $o_level = 0$, ll = *level*$[x]$, last_point[0]=*start*, *RWCET*$_{iso}$[0]=*WCET*$_{iso}$
Output: *RWCET*$_{iso}(x) = R[ll]$
if $o_level < ll$ **then** /* condition 1 */
  $R[ll] = R[ll-1] - d[x]$
else
  if *(last_point[level] == x)* **then** /* condition 2 */
    $R[ll] = R[ll] - w[x]$
  else
    $R[ll] = R[ll-1] - d[x]$
last_point[ll]= x
o_level=ll

Three cases exist during the algorithm execution:

- Case 1: The current observation level o_level is less than the local level ll. In this case the ECFG is traversed in a forward direction, because the observed point x has larger level than the previously observed point. This case occurs when we enter a loop. The remaining WCET is given by the remaining WCET of the head point c minus the time from the head point to the observation point x, $d[x] = d_{c-x}$.
- Case 2: $o_level \geq ll$ and $last_point[ll] = x$. The observation point x is revisited. In this case, the ECFG is traversed in a backward direction due to the head point of a loop component, i.e. the condition. The remaining time of this level is reduced by $w[x] = w_c$.
- Case 3: otherwise. The ECFG is traversed in forward direction as the observed point is placed in a sequence with the previously observed point. This case occurs in the sequential component and the if-then-else component of our grammar. The remaining time is the remaining time of the head point c minus $d[x] = d_{c-x}$.

2) Algorithm proof: The termination of the algorithm is ensured, as the task execution terminates and a finite number of observed points exist. If m points have been inserted to the critical task, each executed at most l times, due to the bounded number of loop iterations, the algorithm is executed at most n times, where n is the total number of visits over the observed points, i.e. $n \leq \sum_{i=0}^{m} l_i$.

We prove the correctness of the Alg. 1 given the graph grammar defined in Section III-A. We show that the result of the algorithm is the $\mathsf{RWCET_{iso}}(x)$ of point x. The proposed algorithm is proved through a structural induction on the grammar rules. Initially, the algorithm is proved for the basic case. Then, starting from the basic case and applying m grammar rules, we reach our induction hypothesis. At that point, the induction step derives from rewriting a non-terminal block B using the grammar components depicted in Fig. 5.

Basic case (Fig. 7(a)): The function F_0 has a non-terminal block B. By definition 8, *level* (in_0) = *level* (out_0) = *level* (b) = 1. By definition 9, *head* (in_0) = *head* (out_0) = *head* (b) = *start*. By definition 7, there is a unique execution $P = (IN_0, B, OUT_0)$. We have moreover $R[0] = \mathsf{RWCET_{iso}}(start) = \mathsf{WCET_{iso}}$. We apply the algorithm on the ECFG.

Point in_0 is the input of the function. Since o_level=0, the *condition 1* of Alg. 1 is true:

$$\begin{aligned} R[1] &= R[0] - d_{start-in_0} = R[0] - (WCET_{iso} - RWCET_{iso}(in_0)) \\ &= RWCET_{iso}(in_0) \\ o_level &= 1 \\ last_point[1] &= in_0 \end{aligned}$$

Point b is just before the execution of the block. Since o_level=1 and the previously observed point is in_0, the *condition 1* of the run-time control algorithm is false. As the $last_point[1]$ is the point in_0 and we observe the point b, the *condition 2* is false:

$$\begin{aligned} R[1] &= R[0] - d_{start-b} = R[0] - (WCET_{iso} - RWCET_{iso}(b)) \\ &= RWCET_{iso}(b) \\ o_level &= 1 \\ last_point[1] &= b \end{aligned}$$

Point out_0 is the point after the execution of all instructions of the block, but for this basic case, the algorithm did not visit any other observation point. The variables are $o_level = 1$, $last_point[1] = b$. Therefore, the *condition 1* and the *condition 2* are false.

$$\begin{aligned} R[1] &= R[0] - d_{start-out} = R[0] - (WCET_{iso} - RWCET_{iso}(out_0)) \\ &= RWCET_{iso}(out_0) = 0 \\ o_level &= 1 \\ last_point[1] &= out_0 \end{aligned}$$

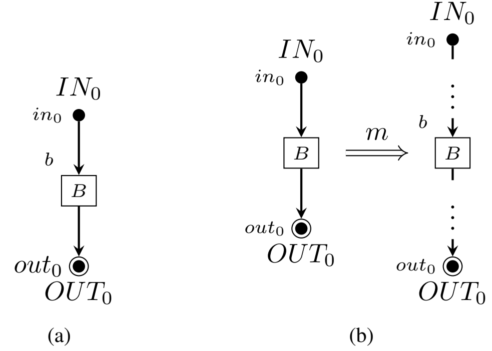

Fig. 7. a) Basic case and b) Structural induction hypothesis.

Induction step: We first define the induction *hypothesis (Fig. 7(b))*. By applying m grammar rules starting from the basic case, we obtain an execution starting from IN_0 and ending at OUT_0 with at least one non-terminal block (we are somewhere in the ECFG creation process), i.e. P=$(IN_0, \ldots, B, \ldots, OUT_0)$. We apply the algorithm until the observation point b, with *level*$(b) = l$ and *head*$(b) = h$, and we compute $R[l] = \mathsf{RWCET_{iso}}(b)$. The run-time execution has already visited the point h, which is the head point of b, with level $l-1$. More precisely, we know that $R[l-1] = \mathsf{RWCET_{iso}}(h)$, o_level=l and $last_point[l] = b$.

A step (Fig. 8) is done as follows: The non-terminal block of B in the execution P is rewritten by applying the grammar rules.

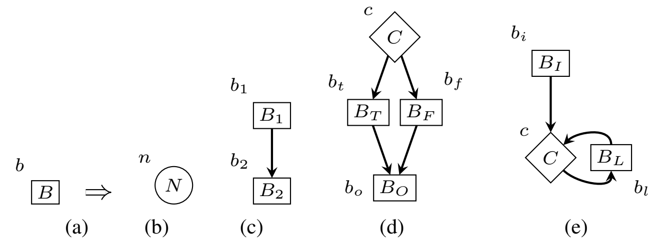

Fig. 8. Structural induction step.

Single node N (Fig. 8(b)): The block B becomes node N and the execution becomes P=$(IN_0, \ldots, N, \ldots, OUT_0)$. By definition 8, *level* (b) = *level* (n) = l and by definition 9, *head* (n) = *head* (b) = h. We now apply the algorithm. Since N replaces B, $R[l] = R[l-1] - (\mathsf{RWCET_{iso}}(h) - \mathsf{RWCET_{iso}}(n)) = \mathsf{RWCET_{iso}}(n)$, o_level=l and $last_point[l]$=n.

Sequential component of two blocks B_1 and B_2 (Fig. 8(c)): The block B becomes two blocks B_1 and B_2, while the execution becomes P=$(IN_0, \ldots, B_1, B_2, \ldots, OUT_0)$. By definition 8, *level* (b_1) = *level* (b_2) = *level* (b) = l and by definition 9, *head* (b_1) = *head* (b_2) = *head* (b) = h. We now apply the algorithm. At point b_1, we have $R[l] = R[l-1] -$

$(\mathsf{RWCET_{iso}}(h) - \mathsf{RWCET_{iso}}(b_1)) = \mathsf{RWCET_{iso}}(b_1)$, o_level=l and $last_point[l]$=b_1. At point b_2, the previously observed point is b_1 which has level l and thus the *condition 1* and *condition 2* are false. Therefore, we have:

$$\begin{aligned} R[l] &= R[l-1] - d_{h-b_2} \\ &= R[l-1] - (RWCET_{iso}(h) - RWCET_{iso}(b_2)) \geq RWCET_{iso}(b_2) \\ o_level &= l \\ last_point[l] &= b_2 \end{aligned}$$

If-then-else condition component (Fig. 8(d)): The block B becomes a conditional node C, two mutually executed blocks B_T and B_F and an output block B_O. The execution becomes P=$(IN_0, \ldots, C, B_T, B_O, \ldots, OUT_0)$ when the condition C is true, or P=$(IN_0, \ldots, C, B_F, B_O, \ldots, OUT_0)$ when the condition C is false. The point b is equal to the point c. Based on definition 8, the level remains same between observation points of the condition node C, the mutually executed blocks B_T and B_F and the output block B. Based on definition 9, all points have the same head point. For point b_f and b_t the previously observed point is c and for b_o is b_f or b_t, depending on the value of the condition C. The *condition* 1 is always false.

For all points $b_p \in \{b_t, b_f, b_o\}$, we have:

$$\begin{aligned} R[l] &= R[l-1] - d_{h-b_p} \\ &= RWCET_{iso}(h) - (RWCET_{iso}(h) - RWCET_{iso}(b_p)) \\ &= RWCET_{iso}(b_p) \\ o_level &= l \\ last_point[l] &= b_p \end{aligned}$$

Loop component (Fig. 8(e)): The block B becomes an initial block B_I, a condition node C and a loop body block B_L. We also know the maximal number of loop iterations p. The execution becomes P=$(IN_0, \ldots, B_I, (C, B_L)^k, C, \ldots, OUT_0)$ with $k \leq p$. By definition 8, *level* (b_i) = *level* (c) = *level* (b) = l and *level* (b_l)=$l+1$. By definition 9, *head* (b_i) = *head* (c) = *head* (b) = h and *head* (b_l)=c. We now apply the algorithm.

Since B_I replaces B, at point b_i we have:

$$\begin{aligned} R[l] &= RWCET_{iso}(b_i) \\ o_level &= l \\ last_point[l] &= b_i \end{aligned}$$

The first time C is visited (iteration 0), the *condition 1* and the *condition 2* are false.

$$\begin{aligned} R[l] &= R[l-1] - d_{h-c} = R[l-1] - (RWCET_{iso}(h) - RWCET_{iso}(c)) \\ &= RWCET_{iso}(c) \\ o_level &= l \\ last_point[l] &= c \end{aligned}$$

The j-th time c is visited (iteration j), the previously observed point is b_l, so o_level=$l+1$. Therefore, the *condition 1* is false and since the previously observed point of level l is c, the *condition 2* is now true.

$$\begin{aligned} R[l] &= R[l] - w_c = RWCET_{iso}(c) - j \times w_c = RWCET_{iso}(c, j) \\ o_level &= l \\ last_point[l] &= c \end{aligned}$$

When b_l is visited (C was true) at the j-th iteration, the *condition 1* is true.

$$\begin{aligned} R[l+1] &= R[l] - d_{c-b_l} \\ &= R[l] - (RWCET_{iso}(c, j) - RWCET_{iso}(b_l, j)) \\ &= RWCET_{iso}(c, j) - (RWCET_{iso}(c, j) - RWCET_{iso}(b_l, j)) \\ &= RWCET_{iso}(b_{l,j}) \\ o_level &= l+1 \\ last_point[l+1] &= b_l \end{aligned}$$

B. Extended version

1) Algorithm description: The extension of the basic version of the proposed run-time algorithm for the critical task described by a set of ECFGs is depicted in Alg. 2.

ALGORITHM 2: Extended version.

```
Pre-computed data: level, w, d, type
Input: x
Data: o_level = 0, ll = level[x], last_point[0]=start, R[0]=WCET_iso, offset = 0
Output: RWCET_iso(x) = R[ll]
if (type[x] ==F_EXIT or F_ENEX) then                     /* condition 4 */
    o_level-=1
    offset -= level[x]
ll = offset + level[x]
Instructions of basic version [Alg. 1]
if  (level[x] ==F_ENTRY or F_ENEX) then                  /* condition 3 */
    offset += level[x]
```

As the function calls link the ECFGs, we need to know at run-time when the execution moves to another ECFG. Therefore, we have to mark the entry and the exit points of a function call and, thus, the $type(x)$ of the points is also stored in memory. Locally, a data *offset* computes the accumulated nested level up to the last observed function entry point. The local level ll depends on this *offset* and the nested levels of the observation point. Note that now the *condition 1* of the basic version becomes also true in the case we enter in a function after the function call node. In the extended version, two additional cases exist during the algorithm execution:

- Case 1: When a function entry point is observed (*condition 3* is true), we increase the offset by the level of the entry point.
- Case 2: When an exit point is observed (*condition 4* is true), we decrease the offset by the level of the entry point. In that case, o_level is decreased by 1 to indicate that we encountered an exit point.

From definition 8, the level of the exit point is equal to the level of the entry point. Hence, if the offset is equal to zero, we have returned in the main function F_0.

2) Algorithm proof: The termination of the algorithm is ensured due to the finite number of loop iterations and function calls. We prove the correctness of the extended algorithm by applying a structural induction on the grammar rules in a similar way with the basic version, as depicted in Fig. 7. The basic case of the extended version remains the same (Fig. 7(a)), as no function call has occurred yet.

Induction step: We first define the induction *hypothesis (Fig. 7(b))*. After applying m grammar rules, we obtain an execution that has started from IN_0 and ends at OUT_0 with at least one non-terminal node, i.e. P=$(IN_0, \ldots, B, V, \ldots, OUT_0)$. We know for point b that $level(b) = l$, $offset = m$, $o_level = m + l$, $R[m + l] = \mathsf{RWCET_{iso}}(b)$. The head point of b is point h and $R[m + l - 1]$=$\mathsf{RWCET_{iso}}(h)$. We need also the successor V of B in the execution, which can be either a terminal or non terminal block. We also know that $level(v) = l$ and $head(v) = h$ since B is a non-terminal block, and thus it cannot be a loop head or a function call.

Step (Fig. 9): The non-terminal block B is rewritten based on the grammar components. For the grammar rules except the function call, the proof of the basic version remains valid.

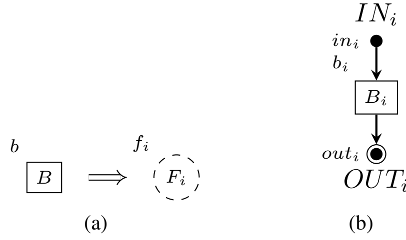

Fig. 9. Structural induction step to function call node.

The main difference occurs when the non-terminal block B becomes a function call node F_i, where the F_i is described by an ECFG. Let the block B become the node F_i and the execution become P=($IN_0, \ldots, F_i, IN_i, B_i, OUT_i, V, \ldots, OUT_0$).

1) Since f_i replaces b, we have $R[m+l] = \mathsf{RWCET_{iso}}(f_i)$. Moreover, the *condition 3* is true, thus *offset* $= m+l$ and *last_point*$[m+l] = f_i$.
2) For $b_k \in \{in_i, b_i, out_i\}$, we have *level*$(b_k) = 1$ and the *condition 3* and *condition 4* are false. The computation is similar in these points, we obtain

$$\begin{aligned} R[m+l+1] &= R[m+l] - d_{f_i - b_k} \\ &= RWCET_{iso}(f_i) - (RWCET_{iso}(f_i) - RWCET_{iso}(b_k)) \\ &= RWCET_{iso}(b_k) \\ o_level &= m{+}l{+}1 \\ last_point[m{+}l{+}1] &= b_k \end{aligned}$$

3) The successor V of B is the exit point of the function call F_i. In v, the condition *C4* is true, meaning that *offset* $= m+l-l = m$ and *o_level* $= m+l$. This implies that *condition 1* and *condition 2* are false.

$$\begin{aligned} R[m+l] &= R[m+l-1] - d_{h-v} \\ &= RWCET_{iso}(h) - (RWCET_{iso}(h) - RWCET_{iso}(v)) \\ &= RWCET_{iso}(v) \\ o_level &= m{+}l \\ last_point[m{+}l] &= v \end{aligned}$$

V. Results

A. Demonstration case study

We use as a case study the lu from Polybench benchmark suite [11], which is depicted in Alg. 3.

ALGORITHM 3: Demonstration case study.

```
# define PB_N 100
int A[PB_N][PB_N];
begin int main()
    lu();
    return EXIT_SUC;

begin int lu()
    int i, j, k;
    for (k = 0; k < PB_N; k++) do
        for (j = k + 1; j < PB_N; j++) do
            A[k][j] = A[k][j] / A[k][k];
        for (i = k + 1; i < PB_N; i++) do
            for (j = k + 1; j < PB_N; j++) do
                A[i][j] -= A[i][k] * A[k][j];
    return EXIT_SUCCESS;
```

1) Design-time analysis: The ECFGs corresponding to the main function F_0 and the lu function F_1 are depicted Fig. 10. Different colors are used to distinguish the nested loops: the first loop is depicted with light gray, the second with gray, the third with black, whereas the white corresponds to no loop. Fig. 11 depicts the corresponding assembly operations and the blocks. An observation point exists per block. By analyzing the ECFGs we pre-compute the data *level*, w, d, *type*, which are depicted in Table III.

The assembly instruction and the grouping into blocks is depicted in Fig. 11.

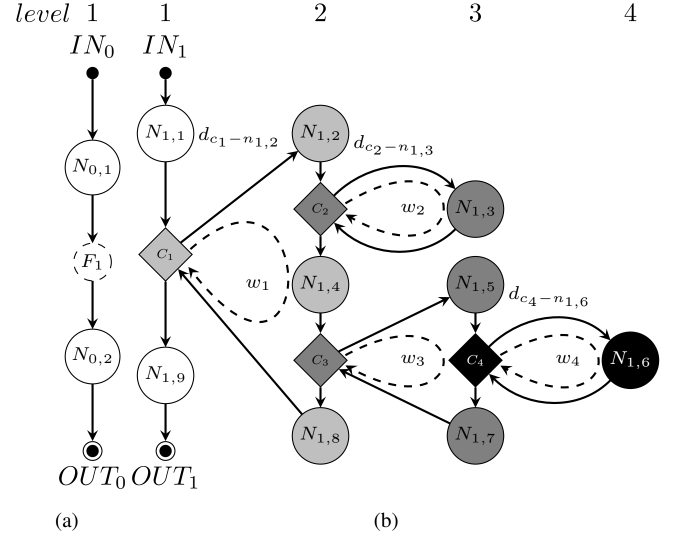

Fig. 10. ECFGs of the main function F_0 and the lu function F_1 with the $level$, the w of loop head points and some d between the loop head point and the first observation point.

TABLE III. Design-time analysis results for main and lu.

Obs.point x	*type* (x)	*level* (x)	w(x)	d(x)
$n_{0,1}$	-	1	0	0
f_1	F_ENTRY	1	0	$d_{start-f_1}$
$n_{0,2}$	F_EXIT	1	0	$d_{start-n_{0,2}}$
$n_{1,1}$	-	1	0	0
c_1	-	1	w_1	$d_{f_1-c_1}$
$n_{1,2}$	-	2	0	$d_{c_1-n_{1,2}}$
c_2	-	2	w_2	$d_{c_1-c_2}$
$n_{1,3}$	-	3	0	$d_{c_2-n_{1,3}}$
$n_{1,4}$	-	2	0	$d_{c_1-n_{1,4}}$
c_3	-	2	w_3	$d_{c_1-c_3}$
$n_{1,5}$	-	3	0	$d_{c_3-n_{1,5}}$
c_4	-	3	w_4	$d_{c_3-c_4}$
$n_{1,6}$	-	4	0	$d_{c_4-n_{1,6}}$
$n_{1,7}$	-	3	0	$d_{c_3-n_{1,7}}$
$n_{1,8}$	-	2	0	$d_{c_1-n_{1,8}}$
$n_{1,9}$	-	1	0	$d_{f_1-n_{1,9}}$

2) Run-time control: We apply the extended version of the proposed run-time algorithm (Alg. 2) on the execution $P = (IN_0, N_{0,1}, F_1, IN_1, N_{1,1}, C_1, N_{1,2}, C_2, N_{1,3}, C_2, N_{1,4}, C_3, N_{1,5}, C_4, N_{1,6}, C_4, N_{1,7}, C_3, N_{1,8}, C_1, N_{1,9}, OUT_1, N_{0,2}, OUT_0)$. The obtained results are depicted in Table IV. The table, following the order of the execution, shows the observation points, the value of the each $condition$ of the run-time computation algorithm, how the $\mathsf{RWCET_{iso}}(x)$ is computed and which is the last point $last_point[ll]$ that modified the $\mathsf{RWCET_{iso}}(x)$.

B. Experimental results

We applied our methodology for several benchmarks from Polybenchmark suite [11] through simulation to obtain preliminary results of our method. We explore several options in the sets of observation points, i.e. Head Points (HP) in: i) nested level 1, ii) nested level 1 and 2, iii) nested level 1, 2 and 3, and iv) nested level 1,2 and 3 and Sequential Points (SP) inside level 3, considering the same deadline per benchmark. We compute the gain of our approach, i.e. the percentage of the execution time of the critical tack in maximum load scenario, and the overhead introduced by our

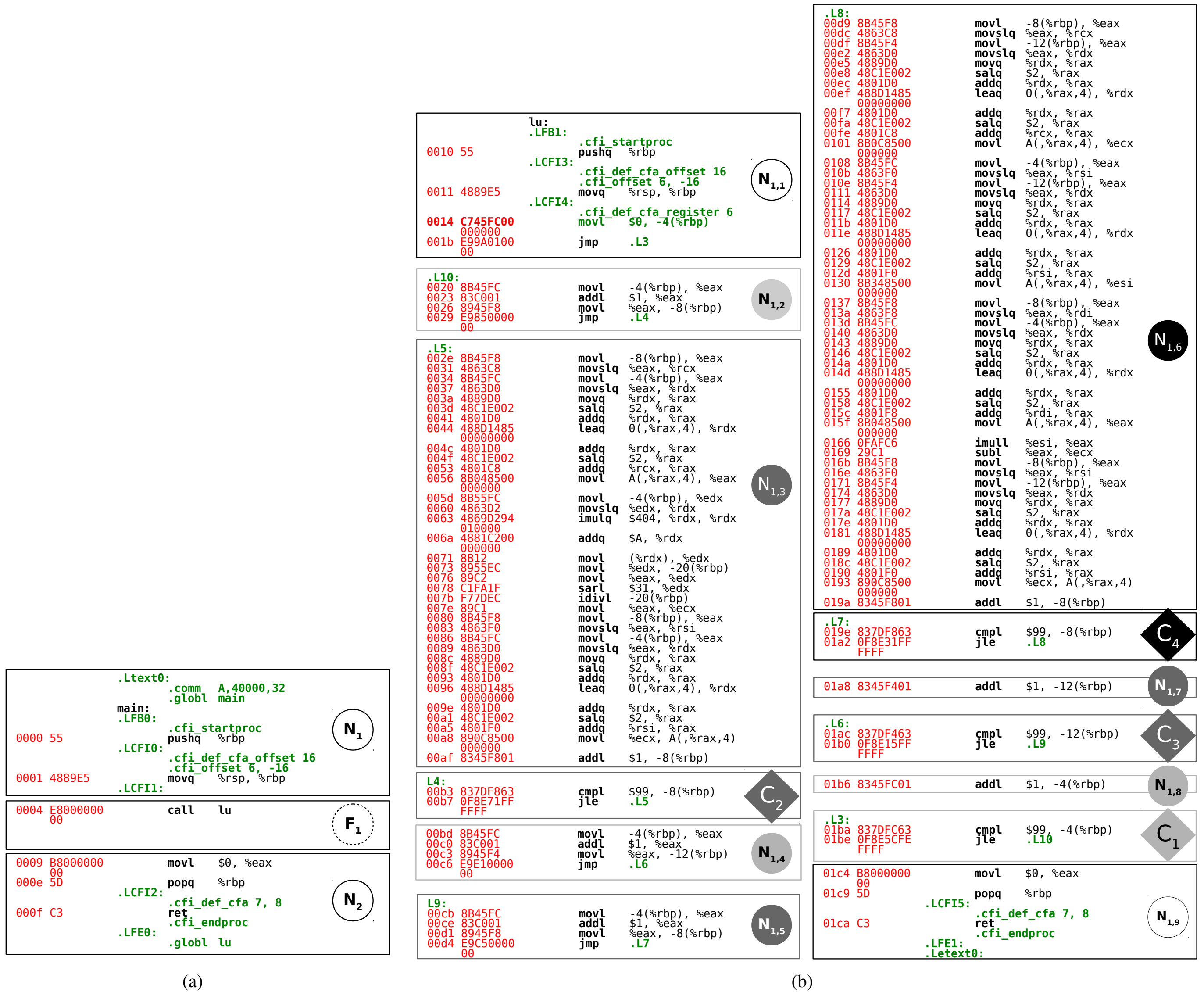

Fig. 11. Assembly instructions and the grouping to nodes for a) main function F_0 and b) lu function F_1.

run-time control. The results are given by the number of observation points executed before the switching to isolation scenario over the total number of observation points introduced in the benchmark. Our simulation has as input the ECFG, the partial RWCETs and the information about the observation points and the execution. We considered an ∼0.10-0.32% difference between the real execution time and the WCET in isolation. The results are depicted in Table V.

Based on the obtained results, a trade-off exists between the number of observation points introduced in the ECFGs and the gain of the proposed approach, which is affected by the shape of the ECFGs. By using few points, e.g. HP in L1, the switching of scenarios may occur too early giving lower than the possible gains, e.g. 25,00% for benchmark bicg. This behavior occurs due to large distance between two sequential observation points. It may occur in the case where inside the loops of level 1 several nested loops exists with high number of iterations. On the other hand, by using too many points, i.e. in all loop head points and in the internal loop kernel, the time between two consecutive points may be similar to the execution time of the controller. Then, a significant time is dedicated to control execution pushing further in time the task execution, resulting to an early scenario switch and to a lower gain of the proposed approach, e.g. 51,90% for benchmark bicg. When the execution time of the critical task between two sequential points is quite larger than the controller overhead, but still not too large to prohibit the exploration of the potentials of our approach, we can achieve significant gains, e.g. 78,68% for benchmark bicg.

VI. Related Work

This section briefly presents the different approaches on the mixed-critical systems, whereas a detailed survey on the mixed-criticality research up to now is available in [12].

A. Task scheduling

The majority of mixed-criticality scheduling work has been mainly addressed for uni-processor platforms (e.g. [2],

TABLE IV. $RWCET_{iso}$ COMPUTATION OF LEVEL l AT RUN-TIME.

Obs. Point	*condition* 1	2	3	4	Offset	$RWCET_{iso}$(x)	Last point[ll]	Obs. level
Init.	x	x	x	x	0	$R[0] = RWCET_{iso}$	LP[0]=0	0
$n_{0,1}$	1	x	0	0	0	$R[1] = R[0] - 0$	LP[1]=$n_{0,1}$	1
f_1	0	0	1	0	0	$R[1] = R[0] - d_{start-f_1}$	LP[1]=f_1	1
$n_{1,1}$	1	x	0	0	1	$R[2] = R[1] - 0$	LP[2]=$n_{1,1}$	2
c_1	0	0	0	0	1	$R[2] = R[1] - d_{n_{1,1}-c_1}$	LP[2]=c_1	2
$n_{1,2}$	1	x	0	0	1	$R[3] = R[2] - d_{c_1-n_{1,2}}$	LP[3]=$n_{1,2}$	3
c_2	0	0	0	0	1	$R[3] = R[2] - d_{c_1-c_2}$	LP[3]=c_2	3
$n_{1,3}$	1	x	0	0	1	$R[4] = R[3] - d_{c_2-n_{1,3}}$	LP[4]=$n_{1,3}$	4
c_2	0	1	0	0	1	$R[3] = R[3] - w_2$	LP[3]=c_2	3
$n_{1,4}$	0	0	0	0	1	$R[3] = R[2] - d_{c_1-n_{1,4}}$	LP[3]=$n_{1,4}$	3
c_3	0	0	0	0	1	$R[3] = R[2] - d_{c_1-c_3}$	LP[3]=c_3	3
$n_{1,5}$	1	x	0	0	1	$R[4] = R[3] - d_{c_3-n_{1,5}}$	LP[4]=$n_{1,5}$	4
c_4	0	0	0	0	1	$R[4] = R[3] - d_{c_3-c_4}$	LP[4]=c_4	4
$n_{1,6}$	1	x	0	0	1	$R[5] = R[4] - d_{c_4-n_{1,6}}$	LP[5]=$n_{1,6}$	5
c_4	0	1	0	0	1	$R[4] = R[4] - w_4$	LP[4]=c_4	4
$n_{1,7}$	0	0	0	0	1	$R[4] = R[3] - d_{c_3-n_{1,7}}$	LP[4]=$n_{1,7}$	4
c_3	0	1	0	0	1	$R[3] = R[3] - w_3$	LP[3]=c_3	3
$n_{1,8}$	0	0	0	0	1	$R[3] = R[2] - d_{c_1-n_{1,8}}$	LP[3]=$n_{1,8}$	3
c_1	0	1	0	0	1	$R[2] = R[2] - w_1$	LP[2]=c_1	2
$n_{1,9}$	0	0	0	0	1	$R[2] = R[1] - d_{n_{1,1}-n_{1,9}}$	LP[2]=$n_{1,9}$	2
$n_{0,2}$	0	0	0	1	0	$R[1] = R[0] - d_{start-n_{0,2}}$	LP[1]=$n_{0,2}$	1

TABLE V. EXPERIMENTAL RESULTS OF OUR METHODOLOGY.

Algorithm	Points Position	Number of points	% max scenario	Tot.overhead (tu)
Basic version				
$LU, N = 16$	HP L1	16	50	18
	HP L1,L2	256	69.53	358
	HP L1-3	4,096	19.33	1,586
$LU, N = 32$	HP L1	32	43.75	30
	HP L1,L2	1,024	67.77	1,390
	HP L1-3	32,768	29.04	19,030
$LU, N = 128$	HP L1	128	25.78	68
	HP L1,L2	16,384	85.81	28,120
$bicg$, NY=NX=16	HP L1	32	25.00	18
	HP L1,L2	272	78.68	430
	HP L1,L2 & SP	578	51.90	602
$bicg$, NY=NX=32	HP L1	64	18.75	26
	HP L1,L2	1,056	37.31	790
	HP L1,L2 & SP	2,178	17.54	766
$bicg$, NY=NX=64	HP L1	128	5.47	16
	HP L1,L2	4,160	11.35	946
	HP L1,L2 & SP	8,450	43.79	7,402
Extended version				
$trmm$, $NI = 8$	HP L1	11	27.27	8
	HP L1,L2	83	63.85	108
	HP L1-3	659	43.86	580
	HP L1-3 & SP	1,316	36.93	974
$trmm$, $NI = 16$	HP L1	19	21.05	10
	HP L1,L2	291	56.70	332
	HP L1-3	4,643	23.48	2,182
	HP L1-3 & SP	9,284	15.53	2,886
$gesummv$, $NI = 8$	HP L1	11	18.18	6
	HP L1,L2	83	31.33	54
	HP L1,L2 & SP	164	48.78	162
$gesummv$, $NI = 16$	HP L1	19	36.84	16
	HP L1,L2	291	40.55	238
	HP L1,L2 & SP	580	51.73	602

HP: Head Points, SP: Sequential Points, tu: time units

[13], [14], [15]), which is not directly applicable in multicore platforms. In the latter, shared resources exist and time compositionality cannot be ensured [16], as the WCET analysis cannot be applied independently per task.

In multicore platforms, several approaches exist that assume that the task set is schedulable at least at the high criticality level. For instance, in [17], both hard real-time and soft real-time tasks are scheduled using an Earliest Deadline First for Hard real-time, Soft real-time and Best effort tasks (EDF-HSB) approach with the assumption that the hard real-time tasks are statically schedulable. When time slack occurs at run-time, it is reallocated to non hard real-time tasks. Another example is the two level mixed-criticality scheduling for multicore platforms proposed in [6] and extended in [7], where the tasks are scheduled based on the WCET of their criticality level and the time slack is reallocated to lower criticality levels. The tasks of different criticality levels are scheduled with different appropriate scheduling approaches. The tasks with the lowest criticality level are allowed to be executed when no higher criticality task is running, i.e. in the critical tasks are executed in isolation. In addition, several mixed-critical scheduling policies have been implemented in the LITMUS^{RT} framework [18].

Less pessimistic approaches, such as [8], [19], [20], use several WCETs per task during task scheduling. Initially, all tasks are assigned their low criticality WCET, which is a less pessimistic bound on WCET given by designers. This WCET derives from a set of test cases [21] and is the maximum execution time observed during execution of the system on maximum load scenario. The proposed algorithms in [8], [19] describe a generalization of the preemptive uniprocessor algorithm EDF with Virtual Deadlines (EDF-VD) to multiprocessor platforms. At run-time, they observe if the tasks have signaled termination at their low criticality WCET. If no signal termination exists by that time, the criticality level of the tasks is increased and the tasks with lower criticality levels are dropped. This occurs because a scenario of higher criticality is now considered and the completion of jobs of lower criticality becomes irrelevant for the new scenario [22]. Further extensions of similar methods are presented in [23] which avoid the abandoning the low criticality tasks during high criticality mode and return to the low criticality mode after the high criticality mode has been terminated. In [20] a Mixed-criticality Scheduling on Multiprocessor (MSM) algorithm is proposed which uses a global fixed priority based approach. When the switching of criticality level occurs, the low criticality tasks are dropped. The approach presented in [24] considers mixed-critical systems and time-triggered paradigm where WCET estimates may be overrun. A run-time monitor is in charge of detecting these overruns and switching on a schedule selected from a set of pre-computed schedules.

Existing approaches explore efficient ways to address mixed-criticality systems in multi/many-core systems, but under the assumption that the WCETs of at least the high criticality level remains below the deadlines. The methodology proposed in this paper improves the resources utilization while guaranteeing the critical task real-time response, when the WCET of the critical task in maximum load scenario is estimated above the deadline, whereas in isolation scenario respects its deadline.

B. Run-time control implementation

Several approaches exist that reallocate the resources based on information derived from monitoring their utilization, e.g. the memory accesses. For instance, in [25] *interference-sensitive* WCETs are computed based on a preliminary analysis of the resource usage of tasks. The shared resources are off-line partitioned among tasks. A run-time monitoring device observes the resource usage of each task and suspends the task that overtakes the allocated capacity. In [26] the approach is extended by allowing safe dynamic changes in the resource partitioning, when resources are underutilized. In [27] an approach has been developed to reserve memory accesses for critical tasks. A run-time controller has been implemented which regulates the accesses to the shared memory and ensures temporal isolation among tasks. An off-line profiling technique

has been proposed in [28] which finds the most frequently accessed memory pages in a task. Then, this information is used to modify the variables' position in the shared caches in order to reduce the interferences.

In contrast, our approach is based on monitoring the real execution time of the critical task and decides the suspension of the low criticality task by run-time computing the remaining WCET of the critical task in isolation.

VII. Conclusion & Future work

In this work, we present a methodology to improve the resources utilization by increasing the task parallelism, while guaranteeing the real-time response of the critical task. At design-time analysis, the critical task is described by a set of ECFGs and partial WCET analysis is applied to compute the required data for the run-time part. At run-time, a low-overhead controller computes the remaining WCET of the critical task and decides the switching between maximum load scenario and isolation scenario.

As future directions, we plan to develop a methodology to decide the position of the observation points over the ECFGs and to implement and evaluate the proposed methodologies to a real mutlicore system. Potentially a similar profiling approach with [28] could be also used to experimentally identify the position of the observation points. In addition, we consider the extension of the proposed approach to several high criticality tasks and several criticality levels. In this context, we plan to explore several strategies on task scheduling, where the proposed methodology could be potentially combined with several approaches presented in the related work.

References

[1] A. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *DAC*, pp. 1–10, ACM/EDAC/IEEE, 2013.

[2] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS*, (USA), pp. 239–243, IEEE, 2007.

[3] SAE, "Aerospace recommended practices 4754a - development of civil aircraft and systems," 2010. SAE.

[4] M. Gatti, "Development and certification of avionics platforms on multi-core processors," in *Tutorial Mixed-Criticality Systems: Design and Certification Challenges, ESWeek*, (Montreal, Canada), 2013.

[5] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[6] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg, "Multicore operating-system support for mixed criticality," in *WMC*, April 2009.

[7] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems.," in *CIT*, pp. 1864–1871, 2010.

[8] H. Li and S. Baruah, "Global mixed-criticality scheduling on multiprocessors," in *ECRTS*, pp. 166–175, 2012.

[9] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, M. Roy, and F. Vargas, "Monitoring on-line timing information to support mixed-critical workloads," in *WiP RTSS*, 2013.

[10] K. D. Cooper, T. J. Harvey, and T. Waterman, "Building a control-flow graph from scheduled assembly code," Tech. Rep. TR02-399, Rice University, 2002.

[11] L.-N. Pouchet *et al.*, "Polybenchmarks benchmark suite." http://www.cse.ohio-state.edu/~pouchet/software/polybench/, 2013.

[12] A. Burns and R. Davis, "Mixed criticality systems - a review," tech. rep., Department of Computer Science, University of York, York, UK, 2014.

[13] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *ECRTS*, (USA), pp. 147–155, IEEE, 2008.

[14] S. Baruah, L. Haohan, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *RTAS*, (USA), pp. 13–22, IEEE, 2010.

[15] A. Burns and B. Baruah, "Towards a more practical model for mixed criticality systems," in *RTSS*, 2013.

[16] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, S. Wegener, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," *Ingénieurs de l'Automobile*, vol. 807, pp. 36–42, September 2010.

[17] B. Brandenburg and J. Anderson, "Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors," in *ECRTS*, (USA), pp. 61–70, IEEE, 2007.

[18] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "Rtos support for multicore mixed-criticality systems," in *RTAS*, pp. 197–208, 2012.

[19] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, pp. 1–36, 2013.

[20] R. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *ECRTS*, (USA), pp. 309–320, IEEE, 2012.

[21] A. Burns and S. Baruah, "Timing faults and mixed criticality systems," in *Dependable and Historic Computing* (C. Jones and J. Lloyd, eds.), vol. 6875 of *Lecture Notes in Computer Science*, pp. 147–166, Springer Berlin Heidelberg, 2011.

[22] S. Baruah, V. Bonifaci, G. D'Angelo, H. L., A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *Trans. Computers*, vol. 61, no. 8, pp. 1140–1152, 2012.

[23] T. Fleming and A. Burns, "Extending mixed criticality scheduling," in *RTSS*, 2013.

[24] S. K. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *RTSS*, pp. 3–12, 2011.

[25] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement," Tech. Rep. 2013-10, University of Augsburg, Germany, 2013.

[26] J. Nowotsch and M. Paulitsch, "Quality of service capabilities for hard real-time applications on multi-core processors," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, (New York, NY, USA), pp. 151–160, ACM, 2013.

[27] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *RTAS*, pp. 55–64, 2013.

[28] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *RTAS*, pp. 45–54, 2013.