DYNASCORE: DYNAmic Software COntroller to increase REsource utilization in mixed-critical systems

Angeliki Kritikakou, Thibaut Marty, Matthieu Roy

DYNASCORE: DYNAmic Software COntroller to increase REsource utilization in mixed-critical systems

Angeliki Kritikakou, University of Rennes 1-IRISA/INRIA
Thibaut Marty, University of Rennes 1-IRISA/INRIA
Matthieu Roy, LAAS-CNRS, University of Toulouse

In real-time mixed-critical systems, Worst-Case Execution Time analysis (WCET) is required to guarantee that timing constraints are respected —at least for high criticality tasks. However, the WCET is pessimistic compared to the real execution time, especially for multicore platforms. As WCET computation considers the worst-case scenario, it means that whenever a high criticality task accesses a shared resource in multicore platforms, it is considered that all cores use the same resource concurrently. This pessimism in WCET computation leads to a dramatic under utilization of the platform resources, or even failing to meet the timing constraints. In order to increase resource utilization while guaranteeing real-time guarantees for high criticality tasks, previous works proposed a run-time control system to monitor and decide when the interferences from low criticality tasks cannot be further tolerated. However, in the initial approaches, the points where the controller is executed were statically predefined. In this work, we propose a dynamic run-time control which adapts its observations to on-line temporal properties, increasing further the dynamism of the approach, and mitigating the unnecessary overhead implied by existing static approaches. Our dynamic adaptive approach allows to control the on-going execution of tasks based on run-time information, and increases further the gains in terms of resource utilization compared with static approaches.

CCS Concepts: •**Computer systems organization** → **Embedded systems; Real-time systems;** Reliability; •**Networks** → Network reliability;

Additional Key Words and Phrases: Dynamic management, Real-time systems, Run-time monitoring, Multicore systems, Safety, Resources Utilization

ACM Reference Format:
A. Kritikakou, T. Marty, and M. Roy. DYNASCORE: DYNAmic Software COntroller to increase REsource utilization in mixed-critical systems. *ACM Trans. Des. Autom. Electron. Syst.* V, N, Article A (January YYYY), 25 pages.
DOI: 0000001.0000001

1. INTRODUCTION

As system requirements increase and power dissipation issues of single-core systems have become a bottleneck, the chip market has moved towards multicore platforms [Singh et al. 2013]. Such systems provide massive computing power, and thus they can execute concurrently a higher volume of applications. Applications may have different properties and requirements, leading to *mixed-critical systems* [Vestal 2007]. A *mixed-critical system* runs applications with different levels of criticality. The criticality level depends partially on the consequences on the system when an application fails to meet its timing constraints. For instance, the Design Assurance Level (DAL)

This work is co-funded by the European Union under the HORIZON 2020 Framework Programme under grant agreement ICT-688131.

Author's addresses: A. Kritikakou and Thibaut Marty, University of Rennes 1-IRISA/INRIA, 35000, Rennes, France; M. Roy, LAAS-CNRS - University of Toulouse, Toulouse, France

DOI: 0000001.0000001

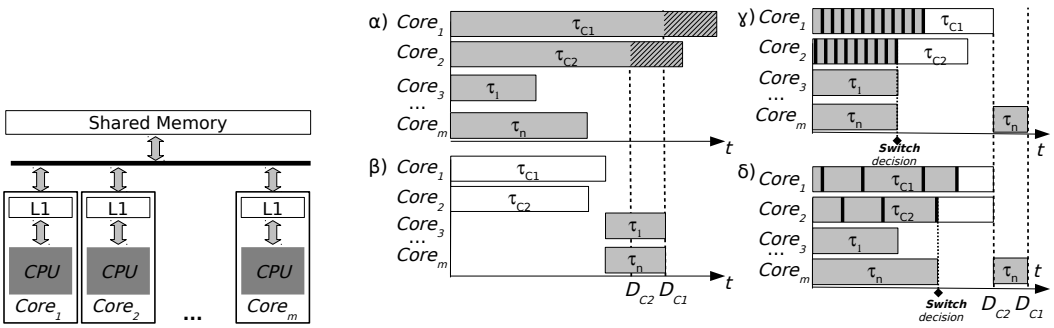
model [SAE 2010] defines *hard real-time* applications with high criticality levels A, B or C and *soft real-time* applications with low criticality levels D or E.

The applications with high criticality level require strict guarantees on their correct execution, i.e. correct functionality and providing responses on time. To ensure these strict guarantees, a safe estimation of the Worst-Case Execution Time (WCET) [Gatti 2013] has to be used. The WCET computation can be achieved through safe measurements or static analysis of the programs [Deverge and Puaut 2007]. The measurements on real hardware or simulator is always possible, but these methods cannot guarantee an exact computation of the worst case path [Deverge and Puaut 2007]. Static analysis relies on hardware models of the processors, which are becoming harder and harder to create [Heckmann et al. 2003] due to the complexity and the missing information of the system architecture, mainly for commercial reasons. Pessimism is introduced to the WCET due to 1) dynamic components with difficult-to-predict behaviour, such as cache memories and branch predictors, and 2) shared resources, where the concurrent accesses introduce timing variations, and thus the effects of possible task interferences have to be upper bounded by assuming full contention at each request. This safe but pessimistic WCET leads to over-allocating resources to high criticality applications and in the worst case, to considering the system as unschedulable.

Taking advantage of the difference in the criticality level of the mixed-critical applications, a possible approach is to develop a run-time controller [Kritikakou et al. 2014a; Kritikakou et al. 2014b] to mitigate this WCET pessimism. In these works, two types of WCET are considered: 1) $WCET_{iso}$ computed when high criticality tasks run only, and 2) $WCET_{max}$, computed with both high and low criticality tasks. When the $WCET_{max}$ for high criticality tasks is computed above the deadline, the system can still be schedulable for high criticality tasks if their $WCET_{iso}$ is below the deadline. Initially all tasks are launched. The controller verifies regularly at statically predefined points if the real occurring interferences lead to a deadline miss by computing the remaining part of the $WCET_{iso}$ (i.e. Remaining WCET, $RWCET_{iso}$). If not, the tasks continue their concurrent execution. Otherwise, the low criticality tasks must stop interfering with the high criticality tasks and are, thus, suspended. When the high criticality tasks finish, interferences from the low criticality tasks are re-allowed, and they resume their execution. In this way, the deadlines of the high criticality tasks are met and the utilisation of the system resources is increased, as low criticality tasks are allowed to run concurrently with high criticality tasks.

However, this static approach is executed at predefined points whose distance is driven by pessimistic WCET estimated at design-time and computed all tasks are allowed to run. Therefore, unnecessary control overhead is introduced affecting the execution time of the high criticality tasks. To address this issue, we propose a dynamic adaptive control mechanism which decides at run-time when the high criticality tasks should be monitored. The run-time controller is executed when it is really necessary reducing the execution time of the high criticality tasks and increasing the resources utilisation compared to the static approaches of [Kritikakou et al. 2014a; Kritikakou et al. 2014b]. The proposed dynamic approach further extends the gains of the static method, as it offers an irregular distribution of the points where the run-time control is executed, which is adapted during the execution of the system based on the monitoring information. More precisely, the contributions of this study on top of the static state of the art approaches are:

— The extension and the instrumentation of the theoretic results of [Kritikakou et al. 2014a] to implement an adaptive run-time version of our controller that is able to activate or inhibit observation points during execution.
— A new algorithm to run-time compute $RWCET_{iso}$ —the Remaining Worst Case Execution Time—, taking into account active and inactive points.



(a) Multi-core system (b) Mixed-critical schedules

Fig. 1. Mixed-critical systems implementation

— An adaptive mechanism to run-time compute the next active observation point.
— A implementation of the proposed dynamic version in a real multicore system
— An extended set of experiments and evaluation results of both the static and the dynamic approaches to show the gains of the proposed approach and to explore the design space of these two methods. The presented exploration study focuses on three main parameters: 1) slack in time, given by the high criticality tasks' deadline, 2) configuration granularities, and 3) low and high interferences. The exploration is performed under an application type that under-privileges the dynamic approach. We left as future work the exploration of different characteristics for the high criticality and low criticality applications.

The remaining of the paper is organized as follows: Section 2 provides a motivational example for the gains of the proposed method. Section 3 describes the target domain and the problem formulation. Section 4 describes the proposed dynamic adaptive control mechanism. Section 6 presents the implementations and the experimental results. Section 7 overviews the related work and Section 8 concludes this study.

2. MOTIVATIONAL EXAMPLE

Let us consider $n + 2$ tasks $\mathcal{T} = \{\tau_{\mathrm{C1}}, \tau_{\mathrm{C2}}, \tau_1, \ldots, \tau_n\}$ where τ_{C1} and τ_{C2} are periodic tasks of high criticality level (DAL A, B or C), period T_{C1} and T_{C2}, and deadline D_{C1} and D_{C2}; $(\tau_i)_{i=1..n}$ are n tasks of low criticality level (DAL D or E). The platform has m cores and each task is executed on a dedicated core.

Most of the existing approaches use one type of WCET to compute the $\boldsymbol{WCET}_{max}$ of τ_{C1} and of τ_{C2}, i.e. when all tasks are executed in parallel. Due to the pessimism in the WCET estimation, the $\boldsymbol{WCET}_{max}$ of τ_{C1} and of τ_{C2} are estimated above their deadlines. This is depicted in Fig. 1(b).α, where the gray box shows the $\boldsymbol{WCET}_{max}$ of τ_{C1} and τ_{C2} and the lined part indicates to which amount the deadlines are not met. The system is considered as not schedulable as the $\boldsymbol{WCET}_{max}$ is estimated above the high criticality tasks deadlines and the hard real-time constraints cannot be guaranteed.

A safe solution is the execution of high criticality tasks (one or more) in isolation, which means that only high criticality tasks are executed on the platform, eliminating the congestion from the low criticality tasks. The $\boldsymbol{WCET}_{iso}$ is significantly lower as no resource sharing and no conflicts occur from the low criticality tasks and the high criticality tasks can respect their deadlines. This is depicted in Fig. 1(b).β, where the white boxes show the $\boldsymbol{WCET}_{iso}$ of τ_{C1} and τ_{C2}, which are below their respective deadlines. When the high criticality tasks terminate, the low criticality tasks start their execution. The real-time constraints are met, but the system resources are under-utilized.

As the $WCET_{iso}$ of τ_{C1} and τ_{C2} is lower than their deadlines, the system can be schedulable at least for the high criticality tasks. The static version of [Kritikakou et al. 2014a; Kritikakou et al. 2014b] combines the previous two extremes to increase the resource utilization. In Fig. 1(b).γ, the gray box depicts the real execution time of the high criticality tasks with the low criticality tasks running in parallel, whereas the black parts show the execution time of the run-time controller regularly repeated at predefined points. When the controller decides to suspend the execution of the low criticality tasks (switch decision), the high criticality tasks are executed in isolation, which is depicted by the white box indicating the $RWCET_{iso}$ in Fig. 1(b).γ. As the $RWCET_{iso}$ is used, the high criticality tasks are guaranteed to meet their deadlines, whereas the partial parallel execution of the high criticality with the low criticality tasks increases the utilization of system resources. However, the monitoring points are statically defined without exploring the real interferences occurring during the execution, introducing unnecessary overhead which affects the execution time of the high criticality tasks and the time that the low criticality tasks can be executed in parallel.

In contrast, this work proposes an approach to reduce the unnecessary time spent in the regular execution of the controller. The dynamic version adapts the run-time control during the execution by deciding the next monitoring point to be activated based on actual information on run-time timing information. As depicted in Fig. 1(b).δ, the number of black marks (which show the execution of the controller) is now reduced and they are irregularly placed, whereas the low criticality tasks run for longer time in parallel with the high criticality tasks (the switching decision is taken later). Section 6.1 quantifies the obtained gains through experimental results.

3. TARGET DOMAIN AND PROBLEM FORMULATION

The platform target domain is a multi/many processor with m cores and r shared resources, whereas the application domain is a mixed-critical system consisting of a set of high criticality tasks and low criticality tasks. Therefore, the systems consisting only of high criticality tasks reside outside our application domain. The proposed approach guarantees the deadlines of the high criticality tasks, whereas improves the core utilization by taking advantage of the existence of low criticality tasks and executes them in parallel, whenever it is possible.

Our approach makes a distinction between two modes of execution: 1) the execution of only the high criticality tasks and 2) the execution of all tasks. The run-time controller switches between these two modes of execution to always achieve timely execution of high criticality tasks, and maximize resource usage when possible.

The input of our approach is the scheduling and assignment decisions over the m cores of the mixed-critical system. There are several techniques [Burns and Davis 2016] to perform scheduling and assignment for mixed critical systems. When the $WCET_{max}$ is estimated above the deadlines of the high criticality tasks (and thus the system is unschedulable as is), the scheduling is performed using high criticality tasks only. Then, the low criticality tasks are scheduled without modifying the scheduling decisions of the high criticality tasks.

In Fig. 2(a), the system is made of x independent tasks (first line). The tasks are partitioned into high criticality and low criticality tasks (line 2). The high criticality tasks $(\tau_{Ci})_{i=1..p}$ are characterized by: 1) High criticality level of DAL A, B or C, 2) Hard deadline $(D_{Ci})_{i=1..p}$, and 3) Period $(T_{Ci})_{i=1..p}$, which can also be 0 in case of a non periodic task. The low criticality tasks $(\tau_{\mathrm{i}})_{i=1..n}$ are characterized by: 1) Low criticality level of DAL D or E, and 2) Soft deadline $(D_i)_{i=1..n}$ or no deadline.

The applied scheduling and assignment (line 3) has as output have p groups of high criticality tasks and n groups of low criticality tasks maximum. Each group runs on a dedicated core (last line), with p less than m and $p+n$ less or equal to m.

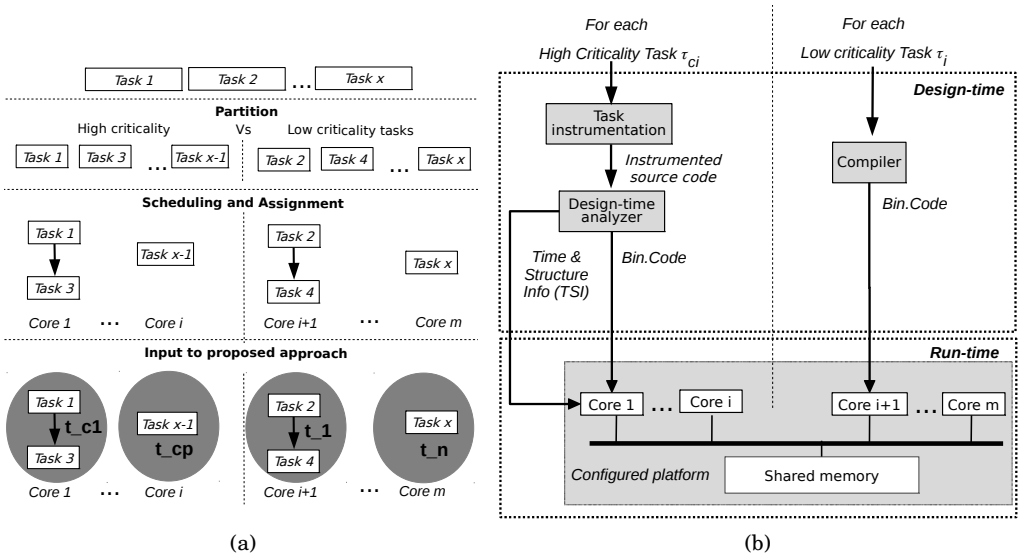


Fig. 2. a) Input and b) Overview of the proposed software methodology.

In the remaining text, the term "a high criticality task" refers to the *group* of high criticality tasks assigned to *a specific core*, whenever this is possible. For instance, the high criticality task running on Core 1 for Fig. 2(a) is the set of Task 1 and Task 3, because the scheduling and assignment step imposes that Task 1 and Task 3 are assigned to Core 1 and Task 3 starts execution after Task 1.

In this manuscript we focus on independent tasks and unified access memory architectures. The proposed approach could also be applied to parallel applications which are modelled as a set of tasks which exchange information through synchronization. In this latter case, the method to estimate the WCET used by the run-time control should include both the interferences and the synchronization costs. Extending our approach to non-uniform memory access (NUMA) architectures requires similar additional knowledge, i.e., the tools to estimate the partial WCET have to take into account the mapping of the tasks to the cores, memory mapping in the different available memory banks, as well as interferences in the grid that interconnects the cores.

4. PROPOSED APPROACH

The proposed methodology is based on two scenarios: the optimistic one, i.e. the maximum load, where low and high criticality tasks are executed at the same time, and the safe scenario, i.e. the isolation, where only high criticality tasks are executed. The approach combines a design-time analysis and a run-time control, as shown in Fig. 2(b).

The design-time analysis provides the information required to guarantee the timing behaviour of the high criticality tasks. Therefore, the high criticality tasks are instrumented with the code of the dynamic adaptive controller. No instrumentation is required for the low criticality tasks. Then, the required Time and Structure Information (TSI) is precomputed for the isolation scenario where no interferences occur by the low criticality tasks. The binary codes and the TSI are loaded onto the corresponding cores, as described by the scheduling given as input to our approach.

During execution, the all tasks run on the system. The dynamic controller is activated at a subset of the inserted monitoring points – depending on the real interferences – in order to manage the execution of the tasks over the platform meeting the

hard deadlines. It checks the timing behaviour of the high criticality tasks and, thus, indirectly estimates the interferences that actually occurred. To provide guarantees, it computes the remaining time required for the high criticality tasks to finish their execution in the isolation scenario, where no low-criticality task can interfere. If enough time still exists, the system continue to accept the interferences from the low criticality tasks, otherwise the core executing the high criticality task sends a request to a global master to suspend the low criticality tasks. The master keeps control of the total suspension/resume requests from the cores running high-criticality tasks. As the run-time control is distributed to each high criticality task, the proposed approach can scale up to many different cores sharing a single memory. For the master, the worst case is when all independent high criticality tasks demand for suspension and resume. Therefore, the master can receive in the worst case 2*i_ht requests, where i_ht is the number of independent high criticality tasks running on the platform.

To provide a better understanding of the steps of the design-time analysis and the run-time control, we use the toy example of Fig. 1(b) to introduce the basic ideas of each step. Let's assume that the τ_{C1} is given by the main function of Fig. 3(a), which consists of two sequential basic blocks ($\mathrm{B}_0, \mathrm{B}_1$), one basic block inside a loop (B_2) and one sequential basic block (B_3). The bounds of the loop are small only in order to facilitate the illustration of the principles.

4.1. Design-time analysis

4.1.1. Critical tasks instrumentation. Each high criticality task τ_{C} is represented by a set of Extended Control Flow Graphs (ECFGs) [Kritikakou et al. 2014a], where an ECFG is a control flow graph with monitoring points.

Definition 4.1 (*High criticality task* $(\tau_{Ci})_{i=1..p}$). A high criticality task $(\tau_{Ci})_{i=1..p}$ is a set of functions $\mathcal{S} = \{F_0, F_1, ..., F_l\}$, with F_0 the main function. Each function is represented by an ECFG.

Definition 4.2 (*ECFG*). An extended control flow graph (ECFG) is a control flow graph extended by adding monitoring points. An monitoring point is a position where the run-time control is executed, except *start* which is the initial point before starting the execution. 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,
   — $C \in \mathcal{C}$ represents the block of binary instructions of a condition statement,
   — $F_i \in \mathcal{F}$ represents the binary instructions of the function caller of a function F_i and links the node with the ECFG of the function F_i,
   — IN is the input node,
   — OUT is the output node.
   — every node $v \in V \setminus \{OUT, IN\}$ has one unique input monitoring point before the execution of the first binary instruction (the monitoring point is represented by a lower-case symbol);

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

In this work, the instrumentation of the high criticality tasks is performed by instantiating theoretical monitoring points of the ECFG of [Kritikakou et al. 2014a] with the code of the proposed dynamic adaptive control mechanism presented in Section 4.2.

Depending on the compiler optimization flag used, the creation of the ECFGs and the instrumentation can be applied in different abstraction layers. When no compiler optimizations are used, which is usually the case in critical systems due to requirements of keeping control of the design and passing the certifications, the ECFG and the in-

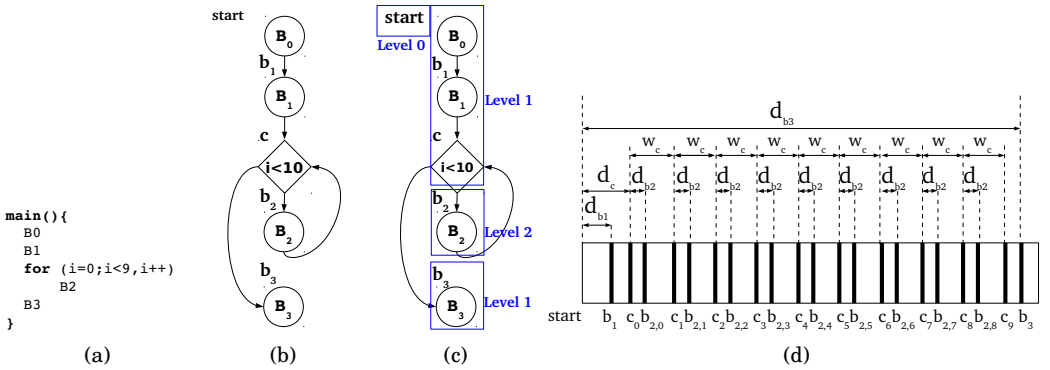


Fig. 3. a) Main function of τ_{C1}, b) ECFG, c) Structure information, and d) Timing information of τ_{C1}.

strumentation can be performed in the application code. When compiler optimizations are used, the ECFGs are created using the binary produced by the complier based on the grammar proposed by our theoretical paper of [Kritikakou et al. 2014a]. Therefore, compiler optimizations and transformations, e.g. loop unrolling, have been already applied. The instantiation of the run-time control occurs without modifying the binary code. Following the grammar rules, the instrumentation of the ECFGs can be achieved in different granularity levels, exploring the trade-offs in the ECFG complexity, the TSI information required to be stored and the time intervals between points.

The result of the instrumentation of τ_{C1} of Fig. 3(a) is the ECFG depicted at Fig. 3(b), where one monitoring point has been introduced to each basic block.

4.1.2. TSI computation. Our approach for the computation of TSI follows the approach described in [Kritikakou et al. 2014a; Kritikakou et al. 2014b]. The description of the structure and the time information is given below.

The *structure information* of the ECFG is required to distinguish between different visits of the same point during run-time execution. For instance, when during execution the point d of Fig. 3(b), which is the condition of the loop, is visited several times due to the loop structure. The structure information is the *level*(x), the *head*(x) point and the *type*(x) of an monitoring point x:

— The *level*(x) is: 1) 0, if x is the *start* point, 2) 1, if x is a sequential point between the IN and the OUT of an ECFG, 3) increased by 1 for each loop, if x is inside a loop.
— The *head* points show i) when a function has been called and ii) where a loop exists in each ECFG. The *head*(x) points are: 1) the *start* point, if x is a point of *level* 1 of the main function F_0, 2) the function caller, if x is a point of *level* 1 of the called function, 3) the condition of the loop, if x is inside a loop.
— The *type*(x) is : 1) F_ENTRY, if x is a function entry, 2) F_EXIT, if x is a function exit, i.e. the node where a function return to, 3) F_ENEX, if x is both a function entry and a function exit, i.e. the point x where the function returns is also a function caller, 4) $-$, otherwise.

In the Fig. 3(c), we depict the structure information for τ_{C1}, where the *start* point has level 0, the points b_1, c and b_3 have level 1 and the point b_2 has level 2 as it belongs inside the loop. The head point of b_1, c and b_3 monitoring points is *start*, whereas the head point of b_2 is the condition of the loop, i.e. the monitoring point c. As in this toy example no function call exists, the type is -.

The *time information* consists of pre-computing *partial RWCET*$_{iso}$ between monitoring points using the *RWCET*$_{iso}$ of a point x.

Definition 4.3 (***RWCET$_{iso}$(x) or RWCET$_{iso}$(x,j)***). ***RWCET$_{iso}$***(x) of a point x, or ***RWCET$_{iso}$***(x,j) when point x is inside a loop at the iteration j, is the ***WCET$_{iso}$*** from the point x up to the end of the task.

The computation of WCET can be performed by using any of the available WCET techniques (a survey is available in [Wilhelm et al. 2008]). Depending on which approach is used, it may be necessary to recompute the WCET values in case of modifications in the set of high criticality tasks. A static analysis based WCET estimation is agnostic to changes in co-executed high criticality tasks. On the contrary, measurement-based WCET estimations need re-evaluation. As soon as the set of high criticality tasks is extended by a new high criticality task, the WCET values have to be recomputed.

We compute three types of partial RWCET between points.

Definition 4.4 ($d_{\text{head}(x)-x}$). $d_{head(x)-x}$ is the partial ***RWCET$_{iso}$*** between a monitoring point x and its head point ***head***(x).

$$d_{head(x)-x} = \textbf{\textit{RWCET}}_{iso}(\textbf{\textit{head}}(x)) - \textbf{\textit{RWCET}}_{iso}(x)$$

Definition 4.5. $w_{head(x)}$ is the partial ***RWCET$_{iso}$*** between any two consecutive iterations j and $j+1$ of the ***head***(x), when ***head***(x) is the condition of a loop.

$$w_c = \textbf{\textit{RWCET}}_{iso}(c,j) - \textbf{\textit{RWCET}}_{iso}(c,j+1), \forall j \leq n$$

To guarantee that the high criticality tasks deadlines are always met, we must ensure that for each high criticality task, enough time is available to decide the suspension of the low criticality tasks at the next monitoring point. Hence, we compute the partial worst case execution time between any two consecutive points x, x' when both high criticality and low criticality tasks are concurrently executed. In this work, we have decided to keep only one value for this partial WCET, ***W$_{max}$***, that is the maximum value for each pair of consecutive points in a high criticality task, in order to store less information. There is a tradeoff between the potential pessimism introduced in this value (due to the asymmetry of the positions of observation points) and the amount of information required to be stored for each point, which is left for future exploration.

$$\textbf{\textit{W}}_{max} = max_{x,x'}(\textbf{\textit{RWCET}}_{max}(x) - \textbf{\textit{RWCET}}_{max}(x'))$$

Fig. 3(d) illustrates the time information derived during the analysis. This information includes the partial ***RWCET$_{iso}$*** from the initial point *start* up to each point, depicted in Fig. 3(d) by d_{b_1}, d_c, d_{b_3}, and from point c to point b_2 by d_{b_2}. For the loop, the partial ***RWCET$_{iso}$*** is computed between any two consecutive iterations, depicted by w_c.

4.2. Run-time control

The system starts running both high and low criticality tasks. Each core with a high criticality task has been instrumented at design-time with the code of the dynamic adaptive controller. Therefore, the controller of a high criticality task is executed locally and independently from the controller of the rest high criticality tasks. A master entity is running on a core running a low criticality task and it is responsible for stopping the interferences of the low criticality tasks with the high criticality tasks. In this work, stopping the interferences generated by low criticality tasks is implemented by pausing the execution of low criticality tasks. More involved strategies, like degraded execution of low criticality tasks or working with local data, are left for future work.

4.2.1. Dynamic adaptive local control mechanism. As stated in Section 4.1, the theoretical monitoring points are instantiated by the dynamic adaptive control mechanism. In contrast to static approaches, the proposed control is executed at each core at different

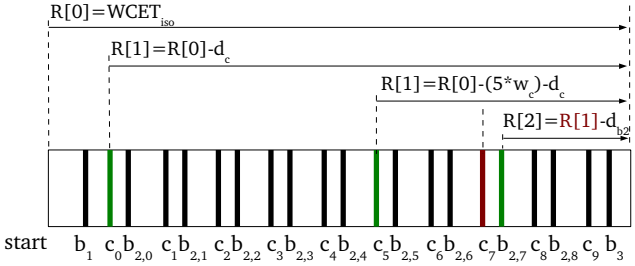


Fig. 4. Run-time execution of τ_{C1}

points which are a priori unknown and decided during the execution of the tasks. Therefore, not all the instrumented monitoring points are used at runtime.

Definition 4.6 (*Active point*). A point x is called an *active* point when:
— the execution changes from an ECFG to another, i.e. the *type* of the point x is F_ENTRY, or F_ENEX
— the point x is placed after a number of monitoring points without enabling the controller, which is given by the variable *points*.

The dynamic adaptive control mechanism is a piece of software as depicted in Algorithm 1, which is executed when we have not decided to eliminate the interferences of the low criticality tasks, i.e. the condition C_{RT} is valid. The control takes as input the precomputed data at design-time, i.e. the instrumented points and their TSI information. At each execution of the control mechanism, it takes as input the occurred point x, the variable $points$ and the values of the current iterators, in case the point x is inside a loop. Two variables are set to 0 before the system execution, i.e. the $counter$ and the $offset$. The $counter$ for the inactive points is increased by one each time we pass an instrumented point in the ECFG of the high criticality task. In Fig. 4, we see the run-time execution of τ_{C1} of Fig. 3 used to illustrate the design-time steps. Each time an instrumented monitoring point is passed (from monitoring point b_1 up to monitoring point b_3), the counter is increased. The control mechanism keeps track of the consecutive ECFG traversal by the high criticality task using the variable *offset*. Each time a function call occurs at a monitoring point, the *offset* is increased by the level of

ALGORITHM 1: Dynamic adaptive control mechanism.

Pre-computed data: $intrum.points$, TSI
Initialisation before system execution: $counter$=0, offset=0
Input: x, *points*, $iterator$
Output: Request to suspend or not the low criticality tasks
`/* High AND Low criticality tasks are running */`
if *(C_{RT})* **then**
  $counter$ ++;
  `/* Returning from a function call */`
  if *(*type[x] ==*F_EXIT*||*F_ENEX)* **then** *offset* = *offset* - *level*[x] ; `/* condition 1 */`
  `/* Active point */`
  if *(counter==points)*||(type[x] ==*F_ENTRY*||*F_ENEX)* **then** `/* condition 2 */`
    $RWCET_{iso}$(x)=Dynamic_computation_RWCET(x, $iterator$, *offset*);
    ET(x) = Monitoring_time();
    points = Next_active_point(*$RWCET_{iso}$*(x), *ET*(x));
    if *(points* $\leq$ 0*)* **then** Request_suspension();
    $counter$ = 0;
  end
  `/* Function call */`
  if *(*type[x] ==*F_ENTRY* || *F_ENEX)* **then** *offset* = *offset* + *level*[x] ; `/* condition 4 */`
end

the point x. In the case that a return from a function call occurs, the *offset* is decreased by the *level* of the point. After the update of the variable $offset$, the control mechanism checks if the point is an active one, using the variable $counter$ to identify when the number of inactive points has passed (given by the variable *points*), or if the point is a function entry point. For instance, if the variable *points* is initially calculated (before execution at point *start*) equal to 2, then the first active point is point c_0 in Fig. 4. If the point x is active, it: 1) computes the $\boldsymbol{RWCET}_{iso}$ at point x, 2) monitors the real execution time $\boldsymbol{ET}(x)$, 3) computes the next active point, and 4) when no more points can be skipped, it sends a request to the master for suspending the execution of the low criticality tasks. Then, the counter is initialized to zero. The next paragraphs describe in details the functions of the proposed run-time control mechanism.

Dynamic computation of $\boldsymbol{RWCET}_{iso}(x)$*:* When a point x is activated, the control mechanism calculates dynamically the $\boldsymbol{RWCET}_{iso}$ at the point x. In contrast to static approaches, the control mechanism is not aware a priori at which point it will be activated, as it depends on actual runtime properties. The corresponding algorithm is depicted in Algorithm 2. The algorithm uses precomputed data at design-time, i.e. the instrumented points and their TSI information, and takes as input the occurred point x, the values of the current iterators $iterator$ and the variable $offset$. Two variables are initialized before the system execution: the $last\_head$ of $\boldsymbol{level}(1)$ to the initial point *start*, and $R[0]$, the $\boldsymbol{RWCET}_{iso}$ at the level 0, i.e., the WCET of the critical task in isolation. This is depicted at the top of Fig. 4, where the remaining time before execution equals to the $\boldsymbol{WCET}_{iso}$. The algorithm uses the actual local level of the point x, ll, to compute the $\boldsymbol{RWCET}_{iso}$ at that level, $R[ll]$ —this value is equal to $\boldsymbol{RWCET}_{iso}(x)$. As stated earlier, the *offset* is updated when the execution changes from one ECFG to another. The *level* provides the nested position in the current ECFG. Therefore, the local level ll derives from the addition of the *offset* and the *level* of the point x.

The run-time calculation of $R[ll]$ of point x is performed in a recursive way using each time the $R[ll-1]$. For instance, in Fig. 4, the first active point is c_0. It has a level equal to 1 and as it belongs to the main function, therefore the $offset$ is 0 and the $ll = 1 + 0 = 1$. The $\boldsymbol{RWCET}_{iso}$ for point c_0 is given by $R[1]$, which is computed by $R[1] = R[0] - (0 * w_c) - d_c = R[0] - d_c$. Then, the second active point is c_5, i.e. the sixth execution of the condition, it has the same local level ll and the $R[1]$ is now computed by $R[1] = R[0] - (5 * w_c) - d_c$. Due to recursive computation, in case that at least one inactive point has passed ($condition\ 3$ is true) in the current ECFG, two problems may occur: 1) the $R[i]$ for each inactive point i has not been updated (as the point was inactive), and 2) the head of the point i is unknown. This occurs only for the points in the current ECFG, because the $offset$ is always up-to-date (a function entry is always an active point, and thus its $R[\boldsymbol{offset}]$ is always up-to-date). To illustrate this point, imagine that third active point is $b_{2,7}$ in Fig. 4 with a level of 2 and, thus, a ll equal to 2. To compute the $\boldsymbol{RWCET}_{iso}$ at this point, we require the value of $R[1]$. However, as the point c_7 was not active (red point in Fig. 4), the value $R[1]$ is not updated.

To address these issues, we find the head points of the levels ll up to the level equal to the $offset$. For the third active point $b_{2,7}$ of Fig. 4 means that the $last\_head$ of local level 1 is point c. Then, the $R[i]$ for these head points is updated using the information of the loop iterators $iterator[y]$, $d[y]$ and $w[y]$ of each head point y. To compute $R[i]$ for a head point y with local level i, we subtract from the remaining time of the head point of the previous local level $i-1$ ($R[i-1]$) the time passed up to now. This time derives by multiplying the $w[y]$ of the head point with local level i with the value of the loop iterator $iterator[y]$ and the $d[y]$ of the head point with local level i. In Fig. 4, the $\boldsymbol{RWCET}_{iso}$ of the head point c of the active point $b_{2,7}$ $R[1]$ is given by $R[1] = R[0] - (7 * w_c) - d_c$

After the update of the previous local levels (or if no inactive point has passed since the last execution), the $R[ll]$ of the active point x can be computed in a simi-

ALGORITHM 2: Dynamic computation $RWCET_{iso}(x)$

Pre-computed data: $intrum.points$, TSI
Initialisation before system execution: $last\_head[1]$=start, $R[0]$=$WCET_{iso}$
Input: x, $iterator$, $offset$
Output: $R[ll]$
$ll = offset + level[x]$
/* More than 1 inactive point */
if *(points* > 1*)* **then** /* condition 3 */
    /* Find the head points for the levels from $offset$ to $ll - 1$ */
    $last\_head[ll - 1]$ = $h[x]$;
    for *(i=ll-1;i>* offset*;i–)* **do** $last\_head[i - 1]$ = $h[last\_head[i]]$;
    /* Update $RWCET_{iso}$ of levels from $offset$ to $ll - 1$ */
    for *(i=* offset + 1*;i<ll;i++)* **do** $R[i]$ = $R[i - 1]$ - $(iterator[last\_head[i]])$ * $w[last\_head[i]]$ - $d[last\_head[i]]$;
end
/* Calculate $RWCET_{iso}$ of the point x with level ll */
$R[ll]$ = $R[ll - 1]$ - $(iterator[x])$ * $w[x]$) - $d[x]$;

lar way. For instance, the third active point $b_2, 7$ of Fig. 4 has a $RWCET_{iso}$ equal to $R[2] = R[1] - d_{b2}$, which is correctly computed as the $R[1]$ has been updated.

Monitoring time: To monitor the current execution time, we developed a set of low level functions that access the timing control registers of the target platform which provide access to the clock of the core. More information about the low level function for our multicore target platform are described in detail in Section 6.1.

Next active observation point: As proven in [Kritikakou et al. 2014a] for the static approach, the low criticality tasks are suspended when the *safety condition* given by Equation (1) does not hold.

$$\boldsymbol{ET}(x) + \boldsymbol{RWCET_{iso}}(x) + \boldsymbol{W_{max}} + t_{\mathrm{SW}} \leq D_{\mathrm{C}} \tag{1}$$

where $\boldsymbol{RWCET_{iso}}(x)$ is the remaining WCET from the point x until the end of the execution of τ_C when only high criticality tasks are executed on the platform, $\boldsymbol{W_{max}}$ is the maximum partial RWCET observed between two sequential points when all tasks are executed on the platform, t_{SW} is the overhead of the suspension of the low criticality tasks and $\boldsymbol{ET}(x)$ is the monitored current execution time of t_C at point x.

To extend Equation (1) to support our dynamic approach, the variable $\boldsymbol{W_{max}}$ has to be multiplied by the variable *points*, which is the number of inactive points up to the next activation of the run-time control mechanism:

$$\boldsymbol{ET}(x) + \boldsymbol{RWCET_{iso}}(x) + (\boldsymbol{points} * \boldsymbol{W_{max}}) + t_{\mathrm{SW}} \leq D_{\mathrm{C}} \tag{2}$$

THEOREM 4.7. *If $\boldsymbol{WCET_{iso}} \leq D_C$ for a high criticality tasks τ_C, then for any execution with the proposed adaptive control mechanism, τ_C always respects its deadline.*

PROOF. If only high criticality tasks are executed on the platform, by definition $\boldsymbol{WCET_{iso}} \leq D_C$. Let us assume that τ_C starts its execution with low criticality tasks until point p_{i+1}. For two consecutive points p_i and p_{i+1}, we have:

$$\boldsymbol{ET}(p_{i+1}) - \boldsymbol{ET}(p_i) \leq \boldsymbol{points} * \boldsymbol{W_{max}} \tag{3}$$

$$0 \leq \boldsymbol{RWCET_{iso}}(p_i) - \boldsymbol{RWCET_{iso}}(p_{i+1}) \tag{4}$$

Since the execution continues for all the tasks until p_{i+1}, it means that p_i fulfilled the safety condition of Equation (1):

$$\boldsymbol{ET}(p_i) + t_{\mathrm{SW}} + (\boldsymbol{points} * \boldsymbol{W_{max}}) + \boldsymbol{RWCET_{iso}}(p_i) \leq D_{\mathrm{C}}$$

The remaining execution from p_{i+1} is safe if τ_C is executed only with high criticality tasks deadline, therefore we have to show that:

$$\boldsymbol{ET}(p_{i+1}) + t_{\mathrm{SW}} + \boldsymbol{RWCET_{iso}}(p_{i+1}) \leq D_{\mathrm{C}}$$

ALGORITHM 3: Demonstration case study: gemm

```
# define PB_N 100
int A[PB_N][PB_N];
int B[PB_N][PB_N];
int C[PB_N][PB_N];
int alpha=32412;
int beta=2123;
begin int main()
    gemm();
    return EXIT_SUC;
end
```

```
begin int gemm()
    int i, j, k;
    for (i = 0; i < PB_N; i++) do
        for (j = 0; j < PB_N; j++) do
            C[i][j] *= beta;
            for (k = 0; k < PB_N; k++) do  C[i][j] += alpha * A[i][k] * B[k][j];
        end
    end
    return EXIT_SUCCESS;
end
```

Thanks to (3), $\boldsymbol{ET}(p_{i+1}) + t_{\mathrm{SW}} + \boldsymbol{RWCET}_{iso}(p_{i+1}) \leq \boldsymbol{ET}(p_i) + \boldsymbol{points} * \boldsymbol{W}_{max} + t_{\mathrm{SW}} + \boldsymbol{RWCET}_{iso}(p_{i+1})$. Because the safety condition (1) holds in p_i, we obtain: $\boldsymbol{ET}(p_{i+1}) + t_{\mathrm{SW}} + \boldsymbol{RWCET}_{iso}(p_{i+1}) \leq D_C + \boldsymbol{RWCET}_{iso}(p_{i+1}) - \boldsymbol{RWCET}_{iso}(p_i)$ Thanks to (4), $\boldsymbol{RWCET}_{iso}(p_{i+1}) - \boldsymbol{RWCET}_{iso}(p_i) \leq 0$, hence we conclude that $\boldsymbol{ET}(p_{i+1}) + t_{\mathrm{SW}} + \boldsymbol{RWCET}_{iso}(p_{i+1}) \leq D_{\mathrm{C}}$, i.e. the high criticality task terminates in time. □

Using Equation (2), we can compute at run-time the number of points that can be safely omitted until the run-time control has to be re-executed: *points*, the current number of consecutive inactive points, is defined by Equation (5):

$$\boldsymbol{points} = \frac{D_{\mathrm{C}} - (\boldsymbol{ET}(x) + \boldsymbol{RWCET}_{iso}(x) + t_{\mathrm{SW}})}{\boldsymbol{W}_{max}} \tag{5}$$

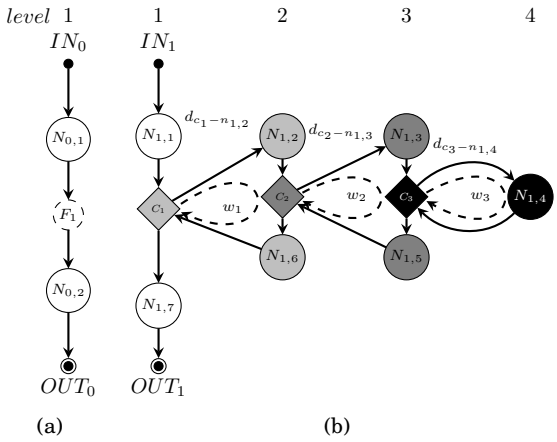
When $\boldsymbol{points} \leq 0$, the low criticality tasks must be suspended in the current active point, as no time remains for suspension in the next active point.

Request suspension of low criticality tasks: The implementation of suspension and resuming of the low criticality tasks is based on a set of interrupts and events to communicate between the cores. When a request for a suspension occurs, the core that runs the high criticality task and requires the suspension sends an interrupt to the core that runs the master entity. Similarly, to resume the execution of the low criticality tasks, the core notifies the master by sending another interrupt. The implementation on the target platform is described in detail in Section 6.1.

4.2.2. Global master control. The master collects the requests of the cores with high criticality tasks through interrupts. When an interrupt-request suspension is received by the master, the execution of the low criticality task is interrupted and the corresponding Interrupt Handling Routine (IHR) is executed. The IHR keeps track of active requests. Upon the first request, the IHR of the master sends an interrupt at each core with low criticality tasks and their IHR prohibits the execution of low criticality tasks, through an active polling mechanism. During isolation, the master updates the number of active requests. When a high criticality task has terminated, it sends a second interrupt-notification and the master reduces the active requests by 1. When all high criticality tasks have finished their execution, and no active requests exist, the low criticality tasks can resume their execution. This is enabled through a second interrupt to the cores with the low criticality tasks that terminates the active polling.

5. CASE STUDY

We use as a case study the *gemm* from Polybench benchmark suite [Pouchet et al. 2013], which is depicted in Algorithm 3. The ECFGs corresponding to the *main* function F_0 and the *gemm* function F_1 are depicted Fig. 5. Different colours 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. An observation point is associated to each block. By analysing the ECFGs we pre-compute the data



Obs. point x	*type* (x)	*level* (x)	w(x)	d(x)
$n_{0,1}$	-	1	0	0
f_1	F_ENTRY	1	0	$d_{start-\mathrm{f}_1}$
$n_{0,2}$	F_EXIT	1	0	$d_{start-\mathrm{n}_{0,2}}$
$n_{1,1}$	-	1	0	$d_{\mathrm{f}_1-\mathrm{n}_{1,1}}$
c_1	-	1	w_1	$d_{\mathrm{f}_1-\mathrm{c}_1}$
$n_{1,2}$	-	2	0	$d_{\mathrm{c}_1-\mathrm{n}_{1,2}}$
c_2	-	2	w_2	$d_{\mathrm{c}_1-\mathrm{c}_2}$
$n_{1,3}$	-	3	0	$d_{\mathrm{c}_2-\mathrm{n}_{1,3}}$
c_3	-	2	w_3	$d_{\mathrm{c}_2-\mathrm{c}_3}$
$n_{1,4}$	-	2	0	$d_{\mathrm{c}_3-\mathrm{n}_{1,4}}$
$n_{1,5}$	-	3	0	$d_{\mathrm{c}_2-\mathrm{n}_{1,5}}$
$n_{1,6}$	-	4	0	$d_{\mathrm{c}_1-\mathrm{n}_{1,6}}$
$n_{1,7}$	-	3	0	$d_{\mathrm{f}_1-\mathrm{n}_{1,7}}$

(c)

Fig. 5. ECFGs of the a) *main* function F_0 and b) *gemm* function F_1 and c) the TSI information.

Table I. $RWCET_{iso}$ run-time computation of level ll.

Active Point	(i,j,k)	Offset	$RWCET_{iso}$(x)	Update
start	-	0	$R[0] = RWCET_{iso}$	-
$n_{0,1}$	-	0	$R[1] = R[0] - 0$	-
f_1	-	0	-	offset
...				
c_3	(0,0,16)	1	$R[4] = R[3] - 16 * w_3 - d_{c_2-c_3}$	R[1],R[2],R[3]
...				
c_2	(0,8,-)	1	$R[3] = R[2] - 8 * w_2 - d_{c_1-c_2}$	R[1],R[2]
...				
$n_{1,7}$	(-,-,-)	1	-	offset

level, w, d, *type*, which are depicted in Fig. 5(c). Table I shows the behaviour of the dynamic approach in the run-time computation of the $RWCET_{iso}(x)$ for the following instrumented points (where (c_i, j) corresponds to the loop i at the iteration j): $n_{0,1}$, f_1, $n_{1,1}$, $(c_1, 0)$, $n_{1,2}$, $(c_2, 0)$, $n_{1,3}$, $(c_3, 0)$, $n_{1,4}$, ..., $(c_3, 16)$, $n_{1,4}$, ..., $(c_3, 31)$, $n_{1,5}$, $(c_2, 8)$, ..., $n_{1,6}$, $(c_1, 20)$, ..., $n_{1,7}$, $n_{0,2}$. The static approach behaviour for the same sequence (all instrumented points are activated) can be found in [Kritikakou et al. 2014a].

6. EVALUATION RESULTS

6.1. Implementation

We are targeting a real multi-core COTS platform, i.e. the TMS320C6678 chip (TMS in short) of Texas Instrument [Texas Instruments 2013]. The platform is composed of 8 TMS320C66x DSP processors clocked at 1 GHz, which can issue up to 8 instructions in one clock cycle. Each core contains 32 KB level 1 program memory (L1P), 32 KB data memory (L1D), and 512 KB level 2 memory (L2) that may contain instructions and data, which can be configured as cache, SRAM or a hybrid. The level 3 memory (L3) of 4 MB on-chip SRAM and the external DDR3 of 512 MB memory is shared among the cores. The cores and the hardware modules are connected via the TeraNet on-chip network. During the experiments, the compiler optimization flag is set to -O0, i.e. no optimizations, in order to have a one-to-one translation of the C code to binary code.

We appropriately configured the TMS platform and implemented the low-level functions to support two main mechanisms used by our software controller to handle the execution of the tasks: the time monitoring of the on-going execution and the suspension/resuming of the low criticality tasks. We reuse the bare-metal library of [Durrieu et al. 2014] that provides a set of timing functions to read the current clock by access-

ing the control registers TCSL and TCSH of the local core clock, which runs at the core's frequency. As the system should start the execution when all tasks have been loaded to the cores, this library also provides a synchronization scheme to ensure that cores start at the same time. When an active observation point occurs, the run-time control uses the functions to read the real execution time of the system. The suspension and the resume of the low criticality tasks is implemented using the event and interrupt mechanisms of the TMS. The bare-metal library is extended with a set of functions that (1) configure the events and the interrupts of the TMS, (2) allow the use of the events by providing software setting, clearing and monitoring mechanisms for the events, (3) keep suspended or resume the low criticality tasks.

6.2. Experimental setup

We consider in our case study the *gemm* as the high criticality tasks executed on core 1 and core 2. The *gemm* benchmark has been selected due to the regularity and the symmetry of each structure, which favours the static approach and under-privileges the dynamic approach. In this way, the experimental results provide a lower bound on the gains of the dynamic approach. A set of loop and data dominated low criticality tasks executed on the remaining cores, which consist of infinitive loops that perform read and write accesses to the memory. The aim of the low criticality tasks in the experimental section is to insert interferences to the high criticality tasks and their number is modified during the experiments in order to tune the number of possible interferences. We present the experimental results for: i) one low criticality task τ_1, in parallel with the high criticality tasks, which is the hardest case as it provides the smallest slack between the $RWCET_{iso}$ and $RWCET_{max}$ (Section 6.4.1) and ii) six low criticality tasks $\tau_1, \ldots, \tau_6$ in parallel with the high criticality tasks, which provides the highest number of interferences (Section 6.4.2).

The memory configuration for all the cores is the following:

— the L1P, L1D, and L2 are configured as SRAMs for better predictability
— the stack, data and code sections (`.stack`, `.data`, `.text` ...) are allocated in the L2 SRAM. The input / output of the code are allocated in the DDR. This means that the arrays A, B, and C of *gemm* are placed in the DDR. In this configuration, conflicts occur in the shared resources among the tasks executed over the platform.

For this system, we tune several parameters to explore the behaviour of our approach:

(1) the deadline of the high criticality tasks D_C, i.e. from tight deadlines close to the WCET of the high criticality tasks in isolation up to more relaxed deadlines,
(2) the number of cores that run low criticality tasks, i.e. from 1 core up to 6 cores,
(3) the granularity of run-time control:
    — *coarse-grained*, monitoring points at the head points of nested level 1 (HP1),
    — *medium-grained*, points at the head points of nested levels 1 and 2 (HP2),
    — *fine-grained*, points at the head points of all nested levels (HP3).

6.3. Design-time analysis

The data required to be pre-computed for the high criticality tasks are the $RWCET_{iso}$, the $d(x)$ and the $w(x)$, and the W_{max} between any two sequential observation points.

For the calculation of the WCET, no existing static WCET analysis tool models our platform. Hence, we calculated the $RWCET_{iso}$, the partial RWCET for the w and the d based on safe measurements on the real platform using the local timer of the cores that run high criticality tasks. To increase the reliability of the measurements, we have repeated our experiments a high number of times and we have maintained the maximum observed value for $RWCET_{iso}$ and the minimum observed value for the partial RWCET for the w and the d. The minimum provides guarantees, as it is the value that is subtracted from the $RWCET_{iso}$ at run-time. If the maximum value would be used,

Table II. Monitoring time overhead (in cycles)

	Read timer	Light control	Full control	Suspend/Restart
Max.time (Cycles)	70	150	1551	200

the high criticality task will be considered to finish earlier. In addition, we increase the maximum observed value and decrease the minimum observed value by 10%. We should stress that during the experiments we observed a low variation of the different samples. For instance, for the isolated execution of the high criticality tasks, we observe a variation of: 1) 0.16% for the execution time of the high criticality task and 2) 3.5% for the W_{max} for the HP1 points, 3.8% for the HP2 and 0.26% for the HP3.

To compute $RWCET_{iso}$ for each high criticality task, we run only the high criticality tasks on the platform. We read the local clock of the core just before the execution of the task and we subtract this value from the time obtained from the local clock at the end of the task execution. To compute $d(x)$ and $w(x)$ for each granularity of our dynamic controller for a point x, we read the local clock of the core at each point. To reduce the overhead introduced by our run-time measurements, we perform the timing measurements for one point at a time. For the $d(x)$ computation, we use the minimum time observed between the head point and the point x. For the computation of the $w(x)$, we use the minimum time observed between any two consecutive loop iterations.

We apply the same technique to compute W_{max}, with the high criticality tasks executed in parallel with low criticality tasks. For each different number of low criticality tasks and for granularity of our run-time control, we have computed the corresponding W_{max} using the local clock of the cores that run the high criticality tasks.

6.4. Run-time control

Table II depicts the maximum time overhead of our run-time control for reading the local timer, updating the offset (light control), computing the $RWCET_{iso}(x)$ and verifying that the low criticality tasks can continue their execution (full control) and performing the suspension/resume of the low criticality tasks.

We consider the notion of relative gain of our methodology compared with the existing static approach of [Kritikakou et al. 2014a; Kritikakou et al. 2014b]. Our comparison with the static approach includes also the comparison of the dynamic version with the execution of only the high criticality tasks, as the experiments of [Kritikakou et al. 2014a; Kritikakou et al. 2014b] have shown significant gain of the static approach with respect to the isolated execution. The relative gain make sense for the deadlines where the interferences of the low criticality tasks cannot be accepted, i.e. switching occurs and the low criticality tasks are suspended. When no switching occurs, both approaches have enough slack to accept the interferences of the low criticality tasks.

Definition 6.1 (*Relative gain*). Let t_{dyn} denote the execution time of the low criticality task with the dynamic approach methodology and t_{sta} the execution time of the low criticality task using the static approach. The *relative gain* of the methodology is:

$$(t_{dyn} - t_{sta})/t_{sta}$$

The execution time of the low criticality tasks for both static and dynamic approaches includes the time up to the switching decision, where all tasks are executed, and the time after the termination of the high criticality task, where the low criticality tasks are resumed until the deadline. We also provide the execution time of the high criticality tasks to compare between the two methods. This execution time is not only affected by the overhead of the controller, but also from the interferences allowed to occur at run-time by the low criticality tasks. However, the execution time of the high criticality tasks provides a metric for the gains of the proposed approach in larger deadlines,

Table III. Execution time and switching time with one low criticality task in parallel (ms).

Deadlines	HP1				HP2				HP3			
	Static		Dynamic		Static		Dynamic		Static		Dynamic	
	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW
625	624.69	0.0003336	624.30	0.000465	624.73	79.32	624.25	81.90	624.11	10.71	622.69	15.51
630	629.39	107.31	628.33	126.92	629.79	163.28	626.64	185.31	629.69	22.88	628.16	33.50
635	634.73	259.31	633.83	259.71	632.81	243.93	631.59	283.00	634.35	35.37	633.67	51.79
640	639.56	367.81	639.52	369.84	639.26	365.58	639.98	445.54	639.67	47.62	638.62	70.05
650	649.79	613.51	642.19	633.15	649.58	533.01	645.62	567.34	649.19	72.50	649.13	104.19
700	655.18	-	653.84	-	659.50	-	657.87	-	693.13	198.14	698.37	275.11
800	655.17	-	653.04	-	659.49	-	657.98	-	798.27	450.59	798.17	603.79
900	655.03	-	653.01	-	659.54	-	657.93	-	894.03	702.71	897.14	-
1,000	655.12	-	652.10	-	659.52	-	657.70	-	998.63	995.57	897.65	-
1,100	655.17	-	653.12	-	659.30	-	657.98	-	1,041.63	-	897.54	-

where both static and dynamic approaches allow the low criticality tasks to always be executed in parallel with the high criticality tasks.

6.4.1. Smallest slack: 1 low criticality task. Table III shows the execution time of τ_{C1} and the time when the switch occurs (SW) for several deadlines and granularities for both the static and the dynamic approaches. The results highlight the effect of the dynamic approach. The switching to the isolated execution of only the high criticality tasks occurs latter as less overhead is introduced by our controller. The more time we spend in the parallel execution of high and low criticality tasks, the longer is the execution time of τ_1, and thus the gain. In addition, τ_{C1} finishes earlier in the dynamic approach, because by removing the overhead of the controller more time is left for the execution of the high criticality task. We also observe that the proposed controller is capable of deciding in a smaller deadline than the static controller that the low criticality tasks can always be executed in parallel with the high criticality tasks, increasing the gain as the low criticality tasks are not suspended compared with the static approach.

Fig. 6 presents the behaviour of the relative gain of our dynamic approach compared with the static approach for the different configurations of our controller and

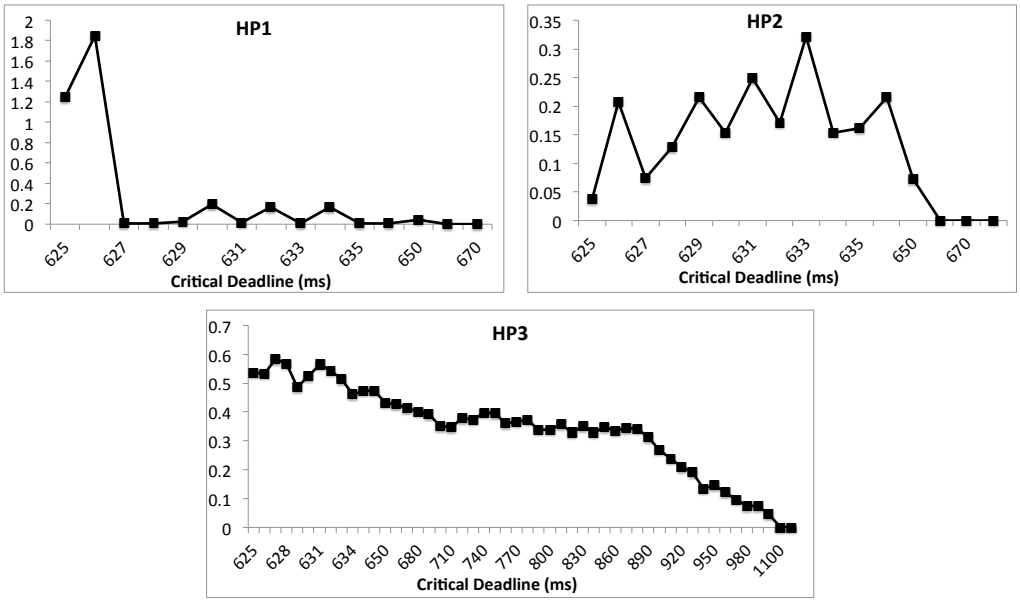


Fig. 6. Relative gain for HP1, HP2, and HP3 configurations

Table IV. Number of active points.

	HP1		HP2		HP3	
Deadlines	Static	Dynamic	Static	Dynamic	Static	Dynamic
625	1	1	2,235	95	21,417	56
630	22	26	4,309	108	45,727	63
635	56	41	6,034	109	70,698	67
640	73	46	9,103	138	95,277	70
650	121	59	13,495	130	145,332	69
700	128	11	16,512	12	399,376	73
800	128	4	16,512	5	911,971	78
900	128	3	16,512	3	1,423,362	23
1000	128	2	16,512	2	1,937,353	10
1100	128	2	16,512	2	2,113,664	7

the different deadlines. From the experiments, we observe that our approach achieves significant relative gains on top of the static approach:

— For the configurations of HP1 and HP2, the improvements of the proposed dynamic approach appear at deadlines which are close to the execution time in isolation. From the Fig. 6(a) and Fig. 6(b), the gain is observed for deadlines from 625 ms up to 650 ms, since after the deadline 650 ms both the dynamic and the static approaches decide that there is no need to suspend the low criticality tasks. For the HP1 configuration we observe a maximum gain of 184.69%, which is the highest gain observed in the three configurations. This occurs for the deadline of 626 ms, with switching times SW equal to 10.03 ms for the static case and 30.65 ms for the dynamic. This occurs because the configuration of HP1 has a large W_{max} between two concurrent observation points, and thus the gain is significantly larger when the dynamic approach passes over observation points. For the configuration of HP2, the highest observed gain is 32% for the deadline of 633 ms.
— For the configuration of HP3 the gains of the dynamic approach are also presented in relaxed deadlines, because the dynamic approach is able to decide in earlier deadlines the parallel execution of the high criticality tasks and the low criticality tasks. Therefore, we observe the aims of our approach up to the deadline 1,000 ms. The highest gain occurs at 627 ms and it is 58.21%.

Table IV shows the number of times the controller has been executed during the execution of the high criticality tasks for several deadlines and granularities for both the static and the dynamic approaches. The results highlight that the proposed approach highly decreases the number of the points that have been activated and thus the overhead of the run-time control mechanisms. For the configuration of HP1 we observe that the number of active points is in the same magnitude for both the dynamic and the static approach. However, the proposed dynamic approach decreases the number of the active points and at the same time it places them closer to the point where the suspension of the low criticality tasks must occur, reducing the number of the active points when the deadline is closer to the deadline where all the tasks can be executed in parallel. In this way, it avoids unnecessary execution of the controller, allowing to further execute the high criticality tasks. Therefore, the point of switching between scenarios arrives later, increasing the gain, as described in the previous paragraphs. For the configurations of HP2 and HP3, the number of active points is reduced from 2 to 4 orders of magnitude for HP2 and 3 to 6 orders of magnitude for HP3.

To further explore the behaviour of the proposed dynamic and the static approach Fig. 7 presents the number of active points where the run-time control has been executed in logarithmic scale for several deadlines and configurations of our controller.

From the experiments, we observe that:

— The static approach in all the configurations (HP1, HP2 and HP3) increases the number of observation points where the run-time controller has been executed with an increase in the critical deadline up to the point where both the high criticality

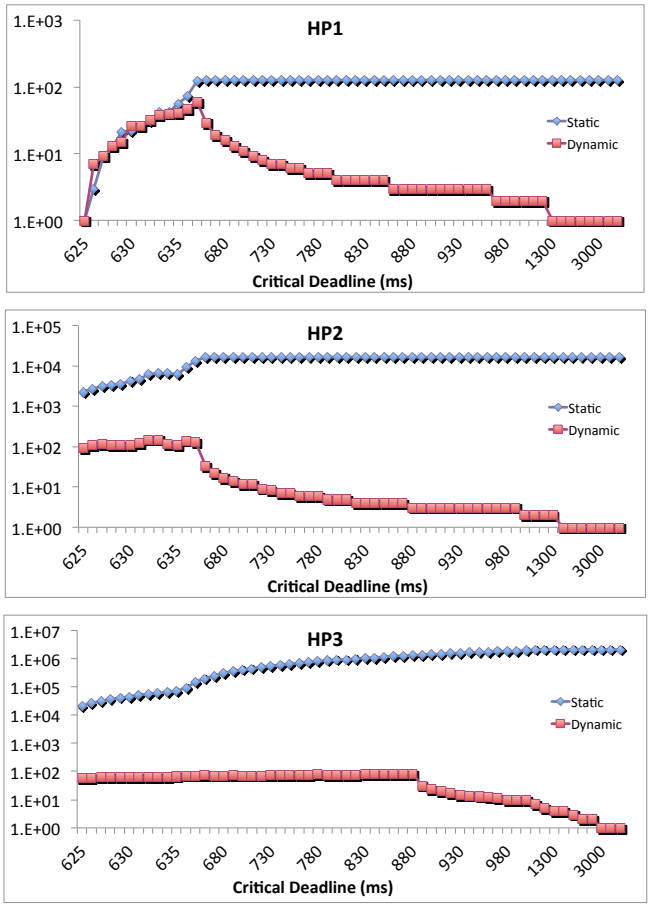


Fig. 7. Number of active points for HP1, HP2, and HP3 configurations

tasks and the low criticality tasks are executed in parallel and the controller is executed at all the observation points.

— The dynamic approach has less active points compared to the static approach for all the configurations. The dynamic computation of the next active point ignores the unnecessary observation points, which adds additional overhead, whereas it moves the verification of the safety condition closer to the actual time when it should occur.

— For the deadlines before the decision of running all the tasks in parallel without suspension (660 ms for HP1 and HP2 and 890 ms for HP3), the dynamic approach has its highest number of points activated. For the HP1 and HP2 configuration, the highest number of active points is observed nearby the deadline that allows parallel execution of all tasks and it is 59 for the HP1 configuration (650 ms) and 138 for the HP2 configuration (640 ms). For the static approach at the same deadlines, we observe 121 active points for the HP1 configuration and 9,103 for the HP2 configuration. For the HP3 configuration, the highest number of active points is 83 and it is observed nearby the deadline that allows parallel execution of all tasks (860 ms). In the same deadline, the static approach executes 1,220,182 times the controller.

— For the deadlines where parallel execution of all the tasks is possible as decided by our dynamic controller, the number of active points is significantly decreased,

as described in the right part of the curves of Fig. 7. As the deadlines are further relaxed, the number of active points is decreased and at the end it converges to only one execution of the proposed dynamic controller which is capable of deciding that all the tasks can run in parallel. In contrast, the static approach executes the control mechanisms at all the statically predefined points.

6.4.2. Highest interferences: maximum number of low criticality tasks. In this section we present the results of the proposed dynamic approach when all the cores are active, i.e. we have six low criticality tasks in parallel and thus the number of congestions over the shared resources is highly increased and we explore the behaviour of our approach for larger deadlines compared with the case presented in section 6.4.1.

Table V shows the execution time of τ_{C1} and the time when the switch occurs (SW) for several deadlines and granularities for both the static and the dynamic approaches. We observe that the dynamic approach decides in earlier deadline than the static approach that all task can be safely executed in parallel for the configuration of HP1 and HP2. A similar behaviour is expected for HP3, when we explore the deadlines with a smaller step, as shown by the results in section 6.4.1.

Fig. 8 presents the behaviour of the relative gain of our dynamic approach compared with the static approach. We observe that for the configurations of HP1 we have a gain of 10.69%, for the HP2 a gain of 55.75% and for HP3 a gain of 14.64%. Compared to the results of the previous section, in the case of the highest interferences the HP2 provides better gains than HP1. This occurs because the w_{HP1} is much higher than the w_{HP2} in the case of the smallest slack. Therefore, the time between two consecutive monitoring points is larger, reducing the possible inactivation of observation points (few points placed far away). In contrast, the w_{HP2} is lower, so it allows an exploration of the gains without too much overhead, whereas w_{HP3} is too small (lot of points placed very close one to the other). Similarly to section 6.4.1, the results highlight that the proposed approach highly decreases the number of the points that have been activated and thus the overhead of the run-time control mechanisms.

Table VI shows the number of times the controller has been executed during the execution of the high criticality tasks for several deadlines and granularities for both the static and the dynamic approaches. For the configuration of HP1 we observe that the number of active points is reduced in almost all the cases by one order of magnitude. The dynamic approach with the configuration of HP1 has the largest number of active points just before the deadline for which it decides that all the tasks can be executed in parallel. Then, the number of active points is monotonically decreasing. For the configuration of HP2, we observe that the number of active points for the proposed dynamic approach are decreased from 2 to 4 orders of magnitude. For the configuration of HP3, we observe that the number of active points for the proposed dynamic approach are decreased from 3 to 6 orders of magnitude, as the deadline is further relaxed.

Fig. 9 presents, in logarithmic scale, the number of active points where the run-time control has been executed, for several deadlines and different configurations of the controller. We still observe that the static approach requires a higher number of active

Table V. Execution time and switching time with six low criticality task in parallel (ms).

Deadlines	HP1				HP2				HP3			
	Static		Dynamic		Static		Dynamic		Static		Dynamic	
	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW	$t_{\tau_{C1}}$	SW
650	643.08	24.12	640.26	24.73	648.11	36.63	642.10	36.68	645.07	35.75	642.11	36.52
750	742.48	142.97	742.40	146.19	743.41	146.19	749.13	151.35	749.33	148.71	749.73	157.88
950	931.63	403.91	913.49	404.02	943.54	433.47	949.600	441.35	948.55	423.24	947.99	426.79
1,500	1,492.54	1,196.04	1,492.35	1,197.08	1,494.68	1,210.3	1,496.89	1,212.12	1,493.54	1,198.07	1,492,80	1,200.49
2,100	2,085.73	2,028.30	2,085.14	-	2,099.44	2,052.23	2,095.76	-	2,099.63	2,035.55	2,096.71	2,041.75
2,300	2,192.60	-	2,085.67	-	2,203.35	-	2,095.87	-	2,269.17	-	2,135.22	-

observation points in all configurations (HP1, HP2 and HP3), whereas the dynamic technique reduces the number of active points when the deadline is relaxed. For the HP1 configuration, the highest number of active points is 9 and this maximum is attained at 1,500 ms and 2,100 ms. For the HP2 configuration the highest number of active observation points is 87 at 2,000 ms, whereas for the HP3 configuration, the highest number of active points is 147 at 850 ms. In the same deadlines for the HP1 configuration, the static approach executes 68 and 117 times the controller, for the HP2 configuration 13,952 times and for the HP3 configuration, 188,078 times.

6.4.3. Different granularity comparison and further discussion. From the obtained results, we observe that the relative gain of HP2 with high interferences is higher that the HP2 gain with low interferences, whereas the opposite occurs for the HP3 gain. We should stress that in the experiments with low interferences one task runs in parallel, and, thus, the computation of W_{max} is less pessimistic, whereas less interferences occur during the execution. In the experiments with high interferences, the deadlines where tasks can be executed in parallel are significantly larger. For the HP2 gain in low interferences, the W_{max} is closer to the real time and we have a medium-grained instrumentation. The dynamic version provides less gains compared with HP2 with high interferences, where W_{max} is pessimistic. When we go to a more fine-grained instrumentation (HP3), the experiments with low interferences provide more gains as the activated points in static are too many compared with the HP2 experiments. But, when more interferences occur, the W_{max} of the internal loop is higher, so the pessimism added reduces the gains of the dynamic approach.

The comparison between different granularities provides significant insight in the two approaches. The granularity decides the time interval between two consecutive observation points. Therefore, a more coarse-grain granularity in the static approach can be seen as equivalent to more fine-grained dynamic approach. However, the two approaches are not equivalent. The static coarse-grained approach has few observa-

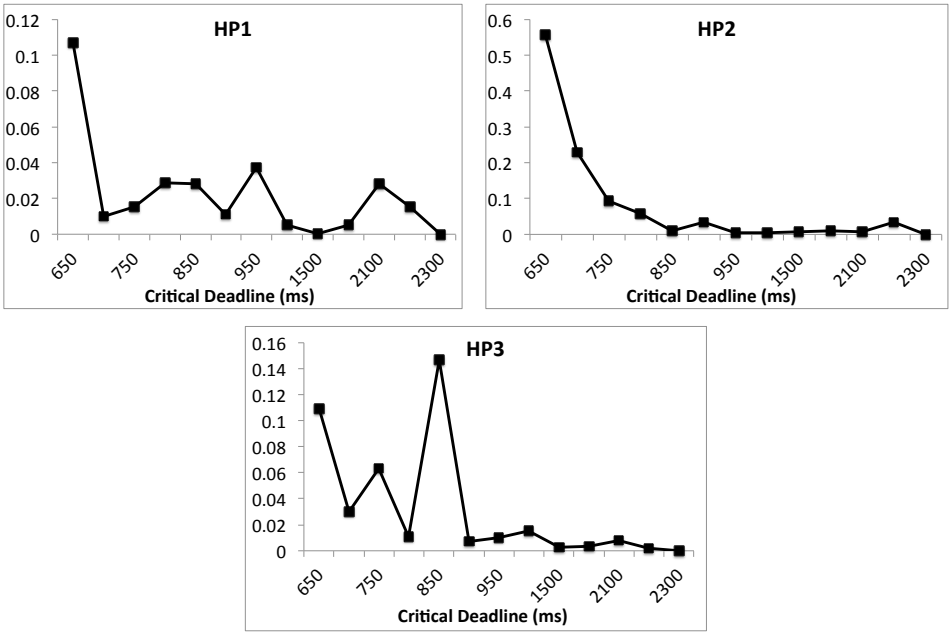


Fig. 8. Relative gain for HP1, HP2, and HP3 configurations

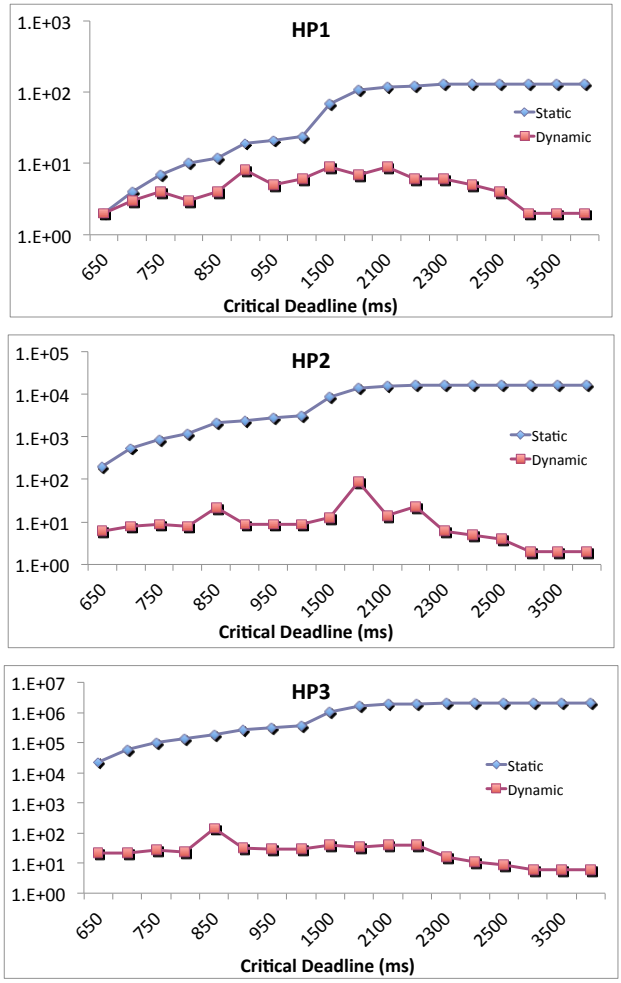


Fig. 9. Number of active points for HP1, HP2, and HP3 configurations

tion points placed regularly in large time intervals, whereas the dynamic approach may have also few points, but they are placed in irregular intervals and closer to the point where the interferences cannot be tolerated. Therefore, the dynamic fine-grained approach is useful in cases where the static coarse-grained approach cannot provide any gains. Here is a motivational example: a code with a three level nested loop, where the outer loops have few iterations and the inner loop is dominant. The coarse-grained static approach will suspend the low criticality tasks too early, as it cannot explore

Table VI. Number of active points.

	HP1		HP2		HP3	
Deadlines	Static	Dynamic	Static	Dynamic	Static	Dynamic
650	2	2	199	6	21,592	21
750	7	4	848	9	96,778	26
950	21	5	2,740	9	315,648	28
1,500	73	46	9,103	138	95,277	70
2,100	68	9	8,697	13	1,071,851	38
2,300	128	6	16,512	6	2,113,664	16

Table VII. Time comparison of different granularities.

Smallest slack						
Switching occurs		No		Yes		When
Original	New	D	High Criticality*	D	Low Criticality*	
S: HP1	D: HP2		0		+ 21,13%	∼
S: HP1	D: HP3	1,100	+ 39%	640	- 80.75%	∼
S: HP2	D: HP3		+ 39%		- 80.72%	∼
Highest Interferences						
Switching occurs		No		Yes		When
Original	New	D	High Criticality*	D	Low Criticality*	
S: HP1	D: HP2		-4%		+ 5.86%	earlier
S: HP1	D: HP3	2,300	- 2.62%	750	+ 10.43%	∼
S: HP2	D: HP3		- 3.9%		+ 8%	∼

* + %: New has longer exec. time, - %: New has shorter exec. time

finer granularity. By using a more fine-grained version, the inner loop can be explored. And by using the dynamic version the unnecessary points of static approaches can be avoided pushing the verification closer to the limit of tolerating interferences.

The decision of which configuration and approach to use is a multidimensional trade-off problem with axes, such as: 1) granularity configuration (affecting the overhead and, thus, the execution time of high criticality task and the step between points, and thus the gain), 2) the available slack (affects the decision of when the switching occurs), 3) the time when switching decision occurs (affects the execution time of the high criticality task due to the interferences of the low criticality), 4) the application (affects the partial RWCET (d and w), and 5) the tasks executed in parallel with the high criticality task (affects the w_{max} and the interferences). We expect that the experimental results of this work provide insight with respect to the different design parameters (static Vs dynamic, coarse-medium-fined grained configurations) and problem constraints (deadline, low and high interferences). Further work is the exploration of impact of the characteristics of the high criticality applications and the low criticality applications.

Table VII compares the gains in execution time of the high and low criticality tasks for different configurations. The comparison is given by $(new - original)/original$ for a deadline that both approaches (original, new) have a parallel execution of all tasks (column “No”), a deadline where both approaches suspend the low criticality tasks (column “Yes”). In column “When” informs which approach explores better the slack.

For the smallest slack possible, from the comparison of a large time interval in the static method (S: HP1) with a medium interval in the dynamic method (D: HP2), we observe that when switching occurs, the D: HP2 behaves generally better that the S: HP1 approach, because of the smaller time interval of D: HP2, which explores better the internal loop. We observe that in the deadline close to the deadline where all tasks can be executed without suspension, the S: HP1 provides more gain. This is because of two main facts. In the smallest slack, the W_{max} is computed between two consecutive points of the external loop and when only one core executes low criticality tasks. Hence, the inserted pessimism of W_{max} is the lowest possible and the run-time computation of the $RWCET_{iso}$ is closer to the actual remaining time. The D: HP2 proceeds in irregular steps, multiples of a smaller time interval W_{max}. The W_{max} is computed between two consecutive points of the second loop. The inserted pessimism is now multiplied by the number of inactive points each time. Therefore, it can happen than for specific deadlines (close to multiplies of the W_{max} of S: HP1) the S: HP1 regular large steps reach close to the case where all task can be concurrently executed. In similar way, for more fine-grained dynamic approach, the coarse-grain static still provides more gain. For larger deadlines, where no switching occurs, the coarse-grained static provides better gains. However, this does not reduce the usefulness of the D: HP3, as in many cases depending on the application, the configuration of HP1 cannot provide gains. We should state that changing from S: HP1 to S: HP3 has also a significant gain reduction

of 87%. However, the D: HP3 approach explores better the available time slack offered by to the deadline compared to S: HP3.

For the highest interferences and the deadlines where switching occurs, the execution time of the high criticality tasks is similar with the coarse-grain static approach, but it provides a gain in the low criticality tasks execution from 5.86% up to 10.43%. For larger deadlines, the execution time of the high criticality task is has a reduction from 2.62% up to 4%. These observations occur because in the case of the highest interferences, the W_{max} of the external loop is computed in the worst case, where all other cores are executing tasks. Therefore, the inserted pessimism in the computation of W_{max} is the largest possible and leads to early suspension of the low criticality tasks.

7. RELATED WORK

The main domains for run-time control of tasks execution and scheduling on multicores are average execution and worst execution. Approaches such as [Pricopi and Mitra 2014] on task scheduling and [Shafique et al. 2015] on run-time task mapping mechanisms belong to the former case. Two main task models are used in worst case domain, i.e. tasks with same and with different criticality. Approaches such as [Chen 2016] belong to the former case, where they consider the WCET of the tasks and focus on task scheduling. Our approach belongs to the second category as it focuses on mixed-critical systems. This section presents existing approaches for implementing mixed-critical systems, with respect to run-time execution and task scheduling. A detailed survey on real-time systems is available in [Burns and Davis 2016].

Task scheduling: The initial mixed-criticality scheduling work has mainly addressed uni-processor platforms (e.g. [Baruah et al. 2010; Burns and Baruah 2013]), the results of which are not directly applicable in multicore platforms. In multicore platforms, several approaches exist that assume that the task set is schedulable at least at the high criticality level, i.e. the WCET is estimated lower than the deadlines. In [Brandenburg and Anderson 2007], 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 approach, assuming that the hard real-time tasks are statically schedulable. When a core finishes its execution before the estimated WCET, this time is reallocated to non hard real-time tasks. The two level mixed-criticality scheduling of [Anderson et al. 2009; Mollison et al. 2010] schedules tasks of different criticality levels with different scheduling approaches. The lowest criticality tasks are allowed to be executed when no higher criticality task is running, i.e. the critical tasks are executed in isolation. To support the approach, the LITMUS^{RT} offers several mixed-critical scheduling policies [Herman et al. 2012]. Other approaches associate several WCETs to each task and consider a scheduling scenario per criticality level [Li and Baruah 2012; Baruah et al. 2013; Pathan 2012]. The hisher the criticality level, the larger and safer the WCETs [Burns and Baruah 2011]. Initially, all tasks are assigned their low criticality WCET. At run-time, they observe if the tasks have signaled termination at the pre-defined position given by the value of the low criticality WCET. If no signal termination has been received, switching to higher criticality WCET occurs.The tasks with lower criticality levels are dropped [Baruah et al. 2012] or allowed to be reexecuted when the system returns to the low criticality mode [Fleming and Burns 2013]. In [Li and Baruah 2012; Baruah et al. 2013] describe a generalization of the preemptive uni-processor algorithm EDF with Virtual Deadlines (EDF-VD) to multiprocessor platforms. In [Baruah and Fohler 2011] mixed-critical systems use time-triggered scheduling where WCET estimates are allowed to overrun. A monitor run-time detects overruns and switches schedules based on a set of pre-computed schedules. In [Giannopoulou et al. 2013], a time-triggered scheduling enables only tasks of the same criticality level to be executed concurrently achieving timing isolation on the core level.

Run-time control: Several approaches propose resources reallocation based on information derived from monitoring their utilization, e.g. the memory accesses. In [Nowotsch et al. 2013] *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 monitor observes at run-time the task resource usages and suspends the task that overtakes the allocated capacity. In [Nowotsch and Paulitsch 2013] allows dynamic changes in the resource partitioning, when resources are underutilized. In [Yun et al. 2012; 2013] memory accesses are reserved for critical tasks. A run-time controller regulates the accesses to the shared memory and ensures temporal isolation among tasks. An off-line profiling technique has been proposed in [Mancuso et al. 2013] which finds the most frequently accessed memory pages in a task. This information is used to modify the variables' position in the shared caches in order to reduce the interferences. The hardware approach of [D. Lo and Suh 2014] allows the monitoring only when enough slack time exists guaranteeing that the monitoring does not impact the meeting of the real-time constraints of the tasks. If no slack exists, a dropping operation minimizes the monitoring overhead.

Our approach is orthogonal to the aforementioned approaches: it considers two types of WCETs depending on how the critical task is executed on the platform (maximum load or isolation), and not on the WCET reliability, it is applicable in cases where the complete system is considered as unschedulable, e.g. the WCET of low and high criticality tasks are estimated above the deadlines, it is based on monitoring the ongoing execution time of the critical task, and not the accesses to the shared resources.

8. CONCLUSIONS

In this work, we present our dynamic software methodology and implementation in a multi-core system for improving resources utilization by increasing task parallelism, while guaranteeing the real-time response of the high criticality tasks. At design-time analysis, the high criticality tasks are instrumented and analysis is applied to compute the structure and timing information. At run-time, an adaptive controller is activated at dynamically computed points, where it computes the remaining WCET of the high criticality tasks and decides whether the low criticality tasks should be suspended. The conducted experiments show a significant gain of our dynamic approach, compared to safe isolation approaches and static run-time control approach, while always guaranteeing hard real-time constraint of the high criticality tasks.

REFERENCES

J. H. Anderson, S. K. Baruah, and B. B. Brandenburg. 2009. Multicore Operating-System Support for Mixed Criticality. In *WMC*.

S.K. Baruah, V. Bonifaci, G. D'Angelo, Haohan L., A. Marchetti-Spaccamela, N. Megow, and L. Stougie. 2012. Scheduling Real-Time Mixed-Criticality Jobs. *Trans. Computers* 61, 8 (2012), 1140–1152.

S.K. Baruah, B. Chattopadhyay, H. Li, and I. Shin. 2013. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems* (2013), 1–36.

S.K. Baruah, L. Haohan, and L. Stougie. 2010. Towards the Design of Certifiable Mixed-criticality Systems. In *RTAS*. 13–22.

S. K. Baruah and G. Fohler. 2011. Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems. In *RTSS*. 3–12.

B.B. Brandenburg and J.H. Anderson. 2007. Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors. In *ECRTS*. 61–70.

A. Burns and B. Baruah. 2013. Towards A More Practical Model for Mixed Criticality Systems. In *RTSS*.

A. Burns and S.K. Baruah. 2011. Timing Faults and Mixed Criticality Systems. In *Dependable and Historic Computing (Lecture Notes in Computer Science)*, Vol. 6875. 147–166.

A. Burns and R. Davis. 2016. *Mixed Criticality Systems - A Review*. Technical Report. Dep. Computer Science, Univ.York, UK.

J. J. Chen. 2016. Partitioned Multiprocessor Fixed-Priority Scheduling of Sporadic Real-Time Tasks. *ECRTS* 00 (2016), 251–261.

T. Chen D. Lo, M. Ismail and G. E. Suh. 2014. Slack-Aware Opportunistic Monitoring for Real-Time Systems. In *RRTAS*.

J. F. Deverge and I. Puaut. 2007. Safe measurement-based WCET estimation. In *WCET*, Vol. 1.

G. Durrieu, M. Faugre, S. Girbal, D. G. Prez, C. Pagetti, and W. Puffitsch. 2014. Predictable Flight Management System Implementation on a Multicore Processor. In *ERTS*.

T. Fleming and A. Burns. 2013. Extending Mixed Criticality Scheduling. In *RTSS*.

M. Gatti. 2013. Development and certification of Avionics Platforms on Multi-Core processors. In *Tutorial Mixed-Criticality Systems: Design and Certification Challenges, ESWeek*.

G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. 2013. Scheduling of Mixed-criticality Applications on Resource-sharing Multicore Systems. In *EMSOFT*. IEEE, Piscataway, NJ, USA, Article 17, 15 pages.

R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. 2003. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE* 91, 7 (2003), 1038–1054.

J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. 2012. RTOS Support for Multicore Mixed-Criticality Systems. In *RTASum*. 197–208.

A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, and M. Roy. 2014a. Run-time Control to Increase Task Parallelism in Mixed-Critical Systems. In *ECRTS*.

A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez. 2014b. Distributed Run-time WCET Controller for Concurrent Critical Tasks in Mixed-critical Systems. In *RTNS*. 139–148.

L. Haohan Li and S. Baruah. 2012. Global Mixed-Criticality Scheduling on Multiprocessors. In *ECRTS*. 166–175.

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

M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. 2010. Mixed-Criticality Real-Time Scheduling for Multicore Systems.. In *CIT*. 1864–1871.

J. Nowotsch and M. Paulitsch. 2013. Quality of Service Capabilities for Hard Real-time Applications on Multi-core Processors. In *RTNS*. 151–160.

J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. 2013. *Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement*. Technical Report. Univ. Augsburg, Germany.

R. M. Pathan. 2012. Schedulability Analysis of Mixed-Criticality Systems on Multiprocessors. In *ECRTS*. 309–320.

Luis-Noel Pouchet and others. 2013. PolyBenchmarks Benchmark Suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/. (2013).

M. Pricopi and T. Mitra. 2014. Task Scheduling on Adaptive Multi-Core. *IEEE Trans. Computers* 63, 10 (2014), 2590–2603. DOI:http://dx.doi.org/10.1109/TC.2013.115

SAE. 2010. Aerospace Recommended Practices 4754a - Development of Civil Aircraft and Systems. (2010). SAE.

M. Shafique, D. Gnad, S. Garg, and J. Henkel. 2015. Variability-aware Dark Silicon Management in On-chip Many-core Systems. In *DATE*. EDA Consortium, San Jose, CA, USA, 387–392.

A.K. Singh, M. Shafique, A. Kumar, and J. Henkel. 2013. Mapping on multi/many-core systems: Survey of current and emerging trends. In *DAC*. 1–10.

Texas Instruments. 2013. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor*. Technical Report SPRS691D. TI.

S. Vestal. 2007. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *RTSS*. 239–243.

R. Wilhelm, J. Engblom, A. Ermedahl, and others. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *TECS* 7, 3 (2008).

H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 2012. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *ECRTS*. 299–308.

H. Yun, G Yao, Rodolfo Pellizzoni, M. Caccamo, and L. Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*. 55–64.

Received January 2017; revised xxx xxx; accepted xxx xxx